Méthodologie de partitionnement logiciel/matériel pour plateformes reconfigurables dynamiquement

Karim Ben Chehida

UNIVERSITÉ DE NICE-SOPHIA ANTIPOLIS - UFR Sciences
Ecole Doctorale de Sciences & Technologies de l'Information et de la Communication

THÈSE

pour obtenir le titre de

DOCTEUR EN SCIENCES
DE L'UNIVERSITE DE NICE-SOPHIA ANTIPOLIS

Discipline (ou spécialité) : Automatique, traitement du signal et des images

présentée par :
AUTEUR : *Karim BEN CHEHIDA*

Méthodologie de Partitionnement Logiciel/Matériel pour Plateformes Reconfigurables Dynamiquement

---

Thèse dirigée par : ***Michel AUGUIN***

Soutenue le : 30 Novembre 2004

Devant la commission d'examen composée de :

M. Olivier SENTIEYS	Professeur, IRISA, Université Rennes 1	Rapporteur
M. Lionel TORRES	Professeur, LIRMM, Université Montpellier II	Rapporteur
M. Michel AUGUIN	Directeur de recherche CNRS, I3S, UNSA	Directeur de thèse
M. Charles ANDRÉ	Professeur, I3S, UNSA	Examinateur
M. Yvon TRINQUET	Professeur, IRCCyN, Université de Nantes	Examinateur
M. Christian GAMRAT	Ingénieur (CEA - LIST)	Invité
M. Philippe KAJFASZ	Responsable du laboratoire architectures avancées (THALES)	Invité

Table des Matières

Introduction

L'ÉMERGENCE de la technologie des microprocesseurs a constitué un tournant technologique important qui a permis aux systèmes électroniques embarqués de devenir une réalité très vite perceptible. Les développements technologiques ont rendu ces microprocesseurs plus flexibles et moins chers. De ce fait, les systèmes embarqués (SE) à base de microprocesseurs ont été introduits dans de nombreux domaines d'application tels que les appareils électroménagers, l'automobile, l'avionique, les équipements réseaux, les terminaux de communication sans-fils, les systèmes multimedia, les systèmes de contrôle industriels, etc. L'utilisation des SE est en train de suivre une courbe exponentielle et on attend même que leurs ventes dépassent en revenus celles des processeurs de PC à usage général. Il est intéressant de constater que nous utilisons actuellement, sans nous en rendre compte, plus d'une douzaine de processeurs embarqués dans notre vie courante. Par exemple, l'électronique embarquée dans le secteur automobile est en train de connaître une grande progression (elle était d'environ 5% des ventes en 2002, Figure 1.1) puisque dans les nouvelles générations de voitures on peut trouver plus de 60 processeurs embarqués qui contrôlent une multitude de fonctions pour améliorer les conditions de conduite telles que l'injection, l'anti-blocage des roues (ABS), l'anti-patinage, l'authentification du conducteur.

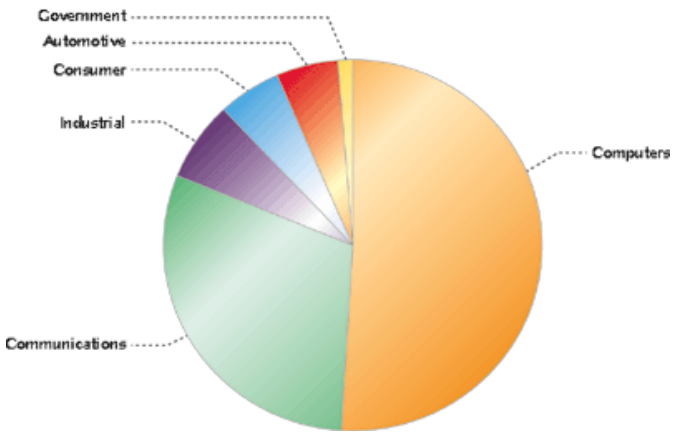


Figure 1.1: Parts de marché des ventes de microprocesseurs (source: Embedded.com, article du 18-12-02)

Les contraintes associées à ce type d'applications sont aussi diverses que variables dans leur criticité. En premier lieu, la plupart de ces applications sont implantées dans des appareils compacts et portables imposant des contraintes sur leurs degrés d'intégration (taille du composant) ainsi que sur leurs consommations d'énergie. En deuxième lieu, le temps de mise sur le marché est devenu un paramètre crucial dans la phase de conception. Avec une concurrence au niveau mondial, la durée typique du cycle de vie d'un produit est descendue à 9 mois pour l'électronique grand public et seulement 18 mois pour les applications en télécommunication, devenant égale ou même plus petite que le temps de conception du produit. Les industriels sont alors confrontés à des fenêtres de marché de plus en plus étroites, une introduction tardive sur le marché réduit considérablement les revenus du produit. En troisième lieu, ces systèmes sont sujets à des contraintes temps réel souvent rigoureuses nécessitant de grandes capacités de calcul et complexifiant ainsi leur implémentation. Enfin, ces systèmes sont de nature hétérogène et combinent:

- des parties analogiques et d'autres numériques.
- du traitement de données à haut niveau et du contrôle (même les données traitées sont souvent de nature et de taille différentes).
- des parties synchrones et d'autres asynchrones.
- des fonctions à contraintes temps réel strictes et d'autres à contraintes temps réel souples.

Prenons l'exemple de l'UMTS[1] qui a pour but d'intégrer des standards de communication de génération actuelle (GSM[2], DECT[3], IS-95[4], Wi-Fi[5]) et de lui adjoindre de nouvelles possibilités de télécommunication ayant des débits plus élevés, avec des capacités multimedia. Une chaîne typique d'émission d'un signal par le biais d'un tel système est comparable à celle modélisée par la figure 1.2.

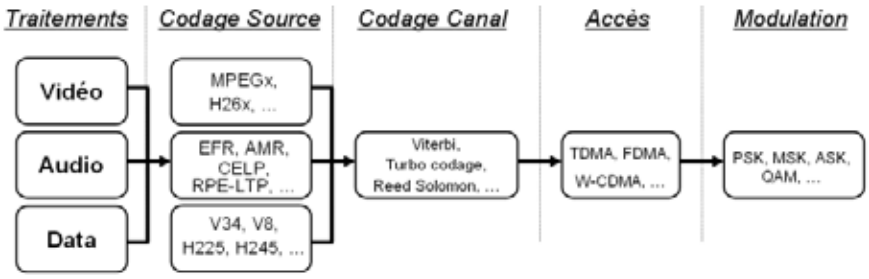


Figure 1.2: Synoptique d'un système d'émission de troisième génération

1. UMTS : Universal Mobile Telecommunication System
2. GSM : Global System for Mobile communications
3. DECT : Digital Enhanced Cordless Telecommunications
4. IS-95 : Interim Standard-95
5. Wi-Fi : Wireless Fidelity

Outre les contraintes déjà citées, cette application en impose une autre née de la définition même du standard. En effet, la norme est sans cesse en train d'évoluer en intégrant de nouvelles applications ce qui a pour conséquence que les terminaux multimedia mobiles doivent être flexibles pour supporter ces évolutions.

Ces applications et les standards qui les définissent rejettent la complexité sur le concepteur qui doit faire des choix sur l'architecture et sur le partitionnement des différentes parties de l'application sur cette architecture.

L'architecture des *SE* a bénéficié de progrès technologiques énormes (une capacité d'intégration de plus de 200 millions de transistors annoncée pour 2005) permettant d'intégrer des systèmes complets sur une seule puce (systèmes sur puce ou *SoC*[1]). Afin de réduire l'effort de conception et contourner des coûts exhorbitants d'ingénierie non récurrente *NRE*[2] (le coût d'un masque variant de $500K à $1M), on privilégie de plus en plus la réutilisation de composants d'une génération à une autre. Cette réutilisation peut être au niveau des sous-blocs du *SoC* et on parle alors de conception par assemblage d'*IP*[3], ou au niveau du *SoC* lui même et on parle alors de conception basée sur une plateforme[4] (ce qui revient à factoriser l'effort de conception pour une classe d'applications).

La figure 1.3 donne l'exemple de la plateforme OMAP2 de Texas Instruments dédiée au standards de communication 3G. Elle est conçue en technologie 90nm et est composée essentiellement d'un processeur ARM11, d'un DSP C55x, d'un accélérateur graphique 2D/3D, d'un accélérateur vidéo, d'interconnexions, de mémoires et de différents périphériques.

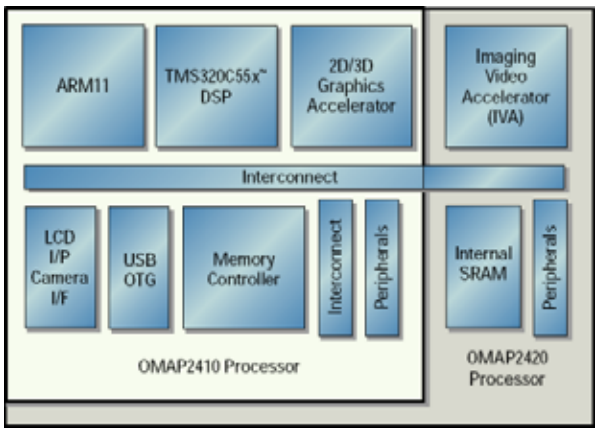


Figure 1.3: La plateforme OMAP2 de Texas Instruments

1. SoC : Systems-on-a-Chip
2. NRE : Non-Recurring Engineering
3. IP : Intellectual Property
4. Platform based design

Les *SoC* sont essentiellement hétérogènes du fait de l'hétérogénéité des applications qu'ils implémentent. Ils combinent typiquement des parties logicielles (des processeurs à usage général *GPP*[1], des processeurs à usage spécifique *ASIP*[2], des processeurs de traitement du signal *DSP*[3]...) et d'autres matérielles (des modules spécifiques *ASIC*[4]).

Une implantation logicielle d'une partie de l'application a l'avantage de lui procurer une flexibilité liée à la possibilité de reprogrammation, contrairement à une implantation matérielle figée mais qui a l'avantage de satisfaire plus facilement les contraintes de performances.

Un autre facteur important dans l'évolution de ces systèmes est l'apparition de nouvelles architectures ayant la flexibilité du logiciel et la performance du matériel, basées sur la programmation de circuits matériels tels que les *FPGA*[5]. Ces propriétés attrayantes et le rythme soutenu des progrès qu'a connu cette technologie (au niveau taille, intégration et performances) ont changé le rôle jusque là joué par ces composants et leur ont permis de passer du rôle de prototypage d'*ASIC* ou de '*glue logic*' au rôle d'unités de calcul alternatives.

On parle de plus en plus de systèmes (ou plateformes) reconfigurables qui intègrent sur un même substrat un ou plusieurs cœurs de processeurs et une matrice programmable (ex: *Virtex-2 Pro* [68], *Virtex-4 FX* [69] de Xilinx et *Excalibur* d'Altera [70]). Par ailleurs, tout un champ technologique émerge actuellement dans le domaine de la reconfiguration dynamique (*RTR*[6]). Cette technologie permet de modifier, en cours d'exécution, partiellement ou complètement la configuration (donc la fonctionnalité) du circuit.

Ceci introduit une nouvelle dimension au problème de conception, en élargissant encore l'ensemble des choix d'intégration possibles. Le concepteur se retrouve donc face à des choix d'implantations logicielles (spécifiques ou génériques) et matérielles (figées ou reconfigurables) pour les différentes parties de l'application. Des approches de conception conjointe logicielle/matérielle (*co-design*) ont été proposées afin d'aider à rechercher une adéquation application/architecture satisfaisant les nombreuses contraintes de conception (coût, performance, surface, consommation, flexibilité...). Cependant, ces choix sont généralement basés sur l'expérience des concepteurs. Pour les prochaines générations de systèmes, cette approche empirique n'est plus envisageable à terme, en effet, la complexité croissante nécessite de faire appel à des méthodes et outils d'aide à la prise de décisions (*CAD*[7]). Des outils automatiques

1. GPP : General Purpose Processor
2. ASIP : Application Specific Integrated Processor
3. DSP : Digital Signal Processor
4. ASIC : Application Specific Integrated Circuit
5. FPGA : Field Programmable Gate Array
6. RTR : Run Time Reconfiguration
7. CAD : Computer Aided Design

existent déjà et sont (plus ou moins) performants pour les niveaux d'abstraction assez bas où la séparation est claire entre le matériel et le logiciel. Pour des niveaux plus hauts où la distinction entre ces deux parties ne peut et ne doit pas être faite encore, ces outils manquent, principalement ceux qui offrent un flot complet à partir de la spécification au niveau système de l'application jusqu'à son raffinement vers les outils de niveau RTL.

Pour les systèmes reconfigurables, ce type de méthodologie cohérente de conception fait également défaut. Il est donc nécessaire d'étendre ou de repenser les approches de conception actuelles afin de les adapter aux possibilités offertes par les technologies matérielles programmables.

Cette thèse propose une méthode automatique de partitionnement logiciel/matériel qui cible des systèmes mixtes logiciel et matériel reconfigurable dynamiquement.

CONTRIBUTION

Le travail élaboré et présenté dans ce mémoire a été initié dans le cadre du projet RNTL *EPICURE* (Environnement de Partitionnement et de Co-développement adapté aux processeurs à architecture REconfigurables) qui a regroupé autour du laboratoire *I3S*, le laboratoire *LESTER* de l'Université de Bretagne Sud, le *CEA/List*, *Thales Communications* et *Esterel Technologies*. Ce projet exploratoire a démarré en février 2001 pour une durée de 2 ans.

Son but était d'apporter des solutions au développement d'applications basées sur une plateforme reconfigurable constituée d'un processeur connecté à une unité reconfigurable dynamiquement à travers une interface intelligente. Pour ce faire, un environnement a été développé intégrant la méthode et les outils logiciels permettant d'estimer les caractéristiques de réalisation des fonctionnalités de l'application et d'effectuer le partitionnement logiciel/matériel de manière automatique. En entrée de la méthode, les spécifications sont développées à l'aide du formalisme graphique SyncCharts basé sur le langage synchrone *Esterel*. Une étape d'estimation architecturale permet une exploration de l'espace de conception et alimente ensuite des librairies logicielles et matérielles qui servent d'entrée à la méthode de partitionnement développée. Après partitionnement, les codes implantés sur le processeur (C, Java) et sur la structure reconfigurable peuvent être générés.

La méthode et l'outil de partitionnement développés dans cette étude sont originaux en plusieurs points. D'abord, peu d'études ont été menées dans le domaine, en partie parce que le pro-

blème de conception conjointe appliquée aux architectures reconfigurables dynamiquement n'a émergé que récemment (on détaillera les méthodes de partitionnement rencontrées dans l'état de l'art). Ensuite, la méthode prend en compte les spécificités de l'architecture reconfigurable en ajoutant au partitionnement spatial (ou affectation) classique qui consiste à affecter les différentes parties de l'application aux composants de l'architecture, une étape de partitionnement temporel afin de distribuer dans le temps les différentes configurations implantées successivement sur la partie reconfigurable. L'évaluation des performances est réalisée par une étape d'ordonnancement qui prend en compte les temps de communication et ceux dus aux changements de configurations.

PLAN DU MÉMOIRE

Le mémoire est structuré en trois parties qui s'intéressent successivement à l'état de l'art, au partitionnement et à la validation de la méthodologie.

Dans la première partie de ce mémoire, nous exposons les principales méthodes de partitionnement logiciel/matériel développées dans la littérature et nous les comparons sur la base du type de modélisation, de l'application d'une part, et de l'architecture cible d'autre part. La deuxième partie est consacrée à la description de notre méthodologie de partitionnement logiel/matériel basée sur un algorithme génétique. Nous développons dans un premier temps le modèle d'application et le modèle d'architecture choisis, ensuite nous présentons la méthode proprement dite. La validation de l'approche proposée fait l'objet de la troisième partie où sont exposés les résultats de partitionnement pour deux applications test: une caméra intelligente qui réalise l'étiquetage et le suivi d'objet en mouvement, et une application en robotique sous-marine relative à la cartographie des fonds marins.

Finalement, en conclusion, nous effectuons une synthèse des contributions de ce travail de recherche et proposons différentes perspectives possibles.

PARTIE 1

Etat de l'art sur le partitionnement logiciel / matériel

Sommaire

Objet

Une méthodologie de co-conception logicielle/matérielle permet de concevoir conjointement les parties logicielles et matérielles du système tout en vérifiant les contraintes imposées, et en réduisant le temps de conception et le temps de mise sur marché. La co-conception logicielle/matérielle de systèmes embarqués peut être le plus simplement représentée par un diagramme classique en ***Y*** illustré par la figure 1.1:

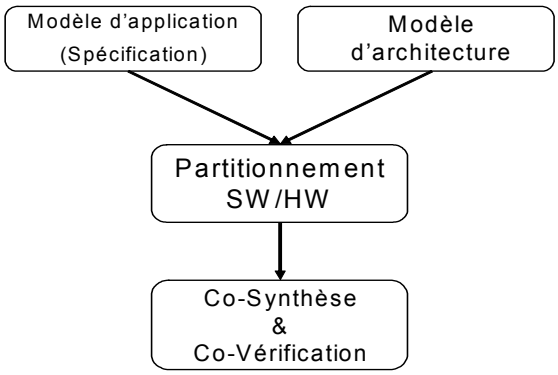


FIGURE 1.1. Représentation en Y de la procédure de co-conception logicielle/matérielle

Dans cette partie, nous exposons les méthodes rencontrées dans la littérature traitant le problème de co-conception logicielle/matérielle. Pour cela nous reprenons les trois premières étapes du diagramme en ***Y*** en nous limitant à l'étude des systèmes reconfigurables dans l'étape de sélection de l'architecture. Nous ne considérons pas l'étape de synthèse qui possède déjà plusieurs outils éprouvés et performants.

Il faut rappeler que le problème de conception est un problème de définition du système. Ceci implique généralement la définition d'un ou de plusieurs modèles du système et de raffiner ces modèles jusqu'à ce que la fonctionnalité désirée soit obtenue tout en vérifiant un ensemble de contraintes. La conception est de ce fait en étroite liaison avec la modélisation.

CHAPITRE 1

Modélisation de l'application

Introduction

Le partitionnement opère sur un modèle de l'application (ou une spécification). Le choix du formalisme pour décrire l'application a donc un fort impact sur la qualité des résultats [1].

Ces modèles de description ont en général été développés pour spécifier un comportement ou une structure et/ou vérifier des propriétés. D. Gajski et F. Vahid les définissent dans [2] comme étant un moyen clair permettant de saisir la fonctionnalité désirée d'un système. Etant donné une séquence de valeurs en entrée, une spécification permet de déterminer ce que devrait être la réponse du système.

Les modèles d'application offrent différents niveaux d'abstraction. La spécification peut être abstraite ou constructive (exécutable). Les modèles abstraits définissent un ensemble d'assertions relatives aux propriétés du système comme sa fonctionnalité, sa charge de travail ou ses dimensions physiques, sans exécuter la spécification. Alors que les modèles exécutables représentent l'application elle même ou un modèle de performance à événements discrets de l'application. Ces modèles sont appelés parfois simulations (terme surtout

approprié quand le modèle exécutable est clairement distinct du système qu'il modélise).

Les modèles exécutables sont construits à partir d'un modèle de calcul (MoC[1]), qui représente l'ensemble des lois qui régissent l'interaction des composants dans le modèle. Plus généralement, c'est un ensemble de règles abstraites (sémantique) qui représente l'ossature permettant au concepteur de créer des modèles. Les modèles de calcul les plus intéressants pour modéliser des systèmes embarqués doivent être en mesure de gérer la concurrence, la réactivité et le temps. En effet, les systèmes embarqués consistent typiquement en un ensemble de composants opérant simultanément et ayant de multiples sources de stimuli simultanés. De plus, ils opèrent dans un environnement réel où le temps de réponse à leurs stimuli est aussi important que l'exactitude de la réponse.

Le choix du modèle de calcul dépend fortement du type de modèle à construire. Par exemple, pour un système purement de calcul qui transforme un bloc fini de données en un autre bloc fini, la sémantique impérative commune des langages de programmation comme C, C++, Java et Matlab serait adéquate. Pour modéliser un système mécanique, la sémantique doit être capable de gérer la concurrence et la continuité temporelle, dans ce cas un modèle de calcul à temps continu comme celui trouvé dans Simulink, Saber, ADS et VHDL-AMS est plus approprié.

Une différence essentielle entre les différents modèles de calcul est donc leur représentation du temps : temps continu, temps discret, temps à ordre partiel, etc. Plusieurs autres manières de les distinguer existent selon par exemple, le type de l'application cible.

Nous considérons dans la suite deux grandes familles de modèles de calcul, une orientée contrôle et une autre orientée traitement. Nous entendons par contrôle tout ce qui est régi par le temps en terme d'évènements discrets, et par traitement tout ce qui est régi par les données elles mêmes. Nous exposons quelques uns de leurs avantages et inconvénients pour présenter ensuite quelques modèles hybrides construits à partir des modèles classiques et finir par citer quelques environnements de modélisation multi-paradigmes.

1. MoC : Model of Computation

1. Modèles orientés traitement

1. 1. Modèles à base de langages impératifs :

Le langage C est le langage le plus utilisé dans la pratique pour exprimer les comportements des systèmes embarqués. D'autres langages impératifs gagnent du terrain comme C++ ou Java du fait de la réutilisation potentielle apportée par le concept d'objet. Ces langages ne permettent pas de révéler un parallélisme potentiel aisément exploitable par les méthodes de conception du fait de leur nature séquentielle. Il existe des compilateurs permettant d'extraire le parallélisme au niveau instruction (ILP[1]), donc ayant une granularité très fine. Ces compilateurs ne sont cependant pas capables d'exploiter le parallélisme à gros grain qui est celui considéré par le concepteur dans une phase initiale de partitionnement ou de conception système. Un langage de type C (qui peut être étendu pour représenter par exemple des processus concurrents) est ainsi utilisé dans les approches [3, 4, 5, 6, 7, 8, 9]. D'autres approches comme [10] et [11] utilisent l'environnement Simulink de Matlab comme langage de spécification.

1. 2. Modèles à base de graphes de flots :

Dans ces modèles, ce sont les données qui fixent l'exécution.

1. Graphes de flot de données

Les graphes de flot de données (DFG[2]) ont été initialement utilisés pour la synthèse de haut niveau [12] et la compilation logicielle [13]. Ils sont formés de nœuds, qui représentent les opérations, reliés par des arcs orientés qui représentent les dépendances (ou chemins) de données. On n'exécute un nœud que lorsque toutes ses données en entrée sont disponibles. Les nœuds peuvent avoir des granularités différentes qui varient de la granularité de la simple instruction à celle de la fonction.

Les approches présentées dans [14, 15, 16, 17] utilisent un modèle de graphe de flots de données avec des nœuds de forte granularité.

2. Graphes de flots mixtes de données et de contrôles

Les graphes de flots de données et de contrôles (CDFG[3]) [18] sont très appropriés pour décrire les manipulations de données, les communications ainsi

1. ILP : Instruction Level Parallelism
2. DFG : Data Flow Graph
3. CDFG : Control Data Flow Graph

que de simples contrôles locaux. Le contrôle est enfoui dans le flot de données au moyen de nœuds de branchement et de fusion et souvent au moyen de structures de boucles spéciales. L'expression du flot de contrôle est souvent limitée et n'est pas en mesure d'exprimer aisément les synchronisations (hors synchronisation par échange de données), ou les interruptions.

On parle aussi de graphes de flots de données et de contrôle hiérarchiques HCDFG[1] [19, 20], où le niveau de la hiérarchie le plus bas est le DFG avec des nœuds élémentaires de traitement et des liens de communication. A un niveau plus haut on retrouve les CDFG avec des nœuds test (*if, case, loop...*) et des appels à des DFG. Au niveau le plus haut, le HCDFG peut contenir des nœuds test, d'autres HCDFG et des CDFG.

1. 3. Processus communicants concurrents :

Les modèles basés sur ces processus permettent de décrire le parallélisme d'une application tout en étant indépendant d'une implémentation logicielle ou matérielle. Ces modèles sont bien adaptés à la description de systèmes de télécommunication.

1. Le modèle basé sur les réseaux de processus de Kahn

C'est un modèle de processus concurrents qui s'échangent des données au travers des liens de communication point à point bufferisés (FIFO de taille infinie).

Dans ce modèle [21], les arcs représentent des séquences de données (*tokens*), et les entités représentent les fonctions qui transforment les séquences d'entrée en séquences de sortie. Pour assurer le déterminisme, quelques restrictions sur ces fonctions ont été introduites comme les lectures bloquantes (condition de monotonicité). Ces réseaux de processus sont faiblement couplés et donc relativement aisés à paralléliser ou à distribuer. Ils peuvent être efficacement implémentés aussi bien en logiciel qu'en matériel et donc les options d'implémentation restent ouvertes.

Ce modèle est assez souple mais il a l'inconvénient d'être difficile à ordonnancer (ordonnancement dynamique en général). Une autre limitation de ce modèle est dans la spécification de la logique de contrôle, de la réactivité (l'interaction avec un utilisateur par exemple).

Dans [22], les auteurs présentent un modèle de calcul qui étend le modèle basé sur les processus de Kahn avec la notion de sélection de canal afin de gérer les évènements non-déterministes.

1. HCDFG : Hierarchical Control Data Graph

2. Le modèle SDF[1]

Le modèle SDF est un sous ensemble restreint des réseaux de processus de Kahn, où l'ordonnancement des exécutions, séquentielles ou parallèles, est calculé statiquement. Cette propriété fait de SDF un formalisme de spécification extrêmement utile pour le matériel et pour le logiciel temps-réel embarqué. Plusieurs outils commerciaux offrent des environnements de modélisation (de systèmes de traitements du signal en général) basés sur le modèle SDF, comme les outils *SPW* [23] de Cadence/Coware et *DSP Station* [24] de Mentor Graphics.

1. 4. Processus communicants séquentiels :

Dans le modèle basé sur les processus séquentiels communicants (CSP[2]) développé dans [25], les entités représentent des processus concurrents implémentés sous la forme de *threads* Java. Ces processus communiquent par des actions atomiques et instantanées appelés *rendez-vous* (ou parfois, passage synchrone de message). Comme exemples de modèles avec *rendez-vous* on peut citer le modèle CSP de Hoare [26] et le modèle CCS de Milner [27]. Ce modèle de calcul a été utilisé dans de nombreux langages de programmation comme Lotos et Occam.

Les modèles de calcul basés sur les processus communicants concurrents et les processus séquentiels communicants possèdent un modèle de temps abstrait dit à ordre partiel. Cette notion de temps à ordre partiel est une sorte de contrainte imposée par la causalité.

2. Modèles orientés contrôle

2. 1. Les réseaux de Petri avec notion de temps :

Pour bénéficier de la puissance des réseaux de Petri dans le domaine des applications temps-réel, différentes extensions ont été introduites aux réseaux de Petri classiques [28] en particulier pour prendre en compte la notion de temps. Ces modèles restent identiques en forme, mais diffèrent par la sémantique des opérations ou par leur manière d'annoter le temps.

Un réseau de Petri ordinaire décrit une relation de causalité entre les évènements, ce qui les ordonne dans le temps. Le temps est pris en compte de

1. SDF : Synchronous DataFlow
2. CSP : Communicating Sequential Processes

manière qualitative. Plusieurs modèles fondés sur les réseaux de Petri permettent de prendre en compte le temps de manière quantitative en associant des durées (des temporisations) aux *transitions* ou aux *places* comme les réseaux de Petri temporisés [29, 30], les réseaux de Petri temporels [31] et les réseaux de Petri stochastiques [32, 33].

Les réseaux de Petri temporisés ainsi que les réseaux de Petri stochastiques sont plutôt utilisés dans la communauté analyse de performances pour construire des modèles réalistes afin d'évaluer, par simulation, des systèmes à évènements discrets. Les réseaux temporels, eux, sont utilisés pour la construction de graphes de couverture des états accessibles du système et la vérification formelle.

Une des faiblesses de ce modèle est la quasi-absence de commodités pour décrire l'aspect communication de données. Quelques modèles fondés sur les réseaux de Petri ont permis d'obtenir une description compacte des parties traitements et données. Ces modèles, dits de haut niveau, attachent une partie des données aux *jetons*. Parmi ces modèles on cite les réseaux de Petri colorés [34], les réseaux Prédicat-Transitions et les réseaux à objet.

2. 2. Les machines à états finis :

Les entités de ce modèle représentent les états, et les connexions représentent les transitions entre états. L'exécution est une succession strictement ordonnée d'états et de transitions. Les machines à états finis (FSM[1]) sont des modèles bien adaptés pour exprimer la logique de contrôle et pour construire des modèles de modes (systèmes avec des modes d'opération distincts, où le comportement est différent pour chaque mode : intéressant pour exprimer différent modes de reconfiguration dans un système reconfigurable dynamiquement). Les modèles à base de FSM se prêtent bien à une analyse formelle et peuvent, de ce fait, être utilisés pour vérifier l'absence de comportements inattendus du système.

L'une des faiblesses de ce modèle réside dans le fait que le nombre d'états peut devenir rapidement excessivement grand en fonction de la complexité des systèmes.

2. 3. Modèles réactifs synchrones :

Le modèles Réactif Synchrone (RS) a été très utilisé pour la co-conception logicielle/matérielle. Parmi les points forts des langages basés sur le modèle RS, on cite leur syntaxe simple à utiliser et leur sémantique succincte et for-

1. FSM: Finite State Machine

melle (ce qui a permis de développer des outils de simulation et de vérification efficaces). Dans le modèle RS, les entités représentent des relations entre les valeurs en entrées et celles en sorties à chaque *réaction* du système. Comme exemple de langages basés sur le modèle de calcul RS, on cite *Esterel* [35], *Signal* [36], *Lustre* [37] et *Argos* [38].

Les modèles RS se prêtent parfaitement pour décrire des applications comprenant de la logique de contrôle complexe et concurrente. Grâce à l'hypothèse de forte synchronisation qu'offrent ces modèles, ils peuvent cibler des applications temps-réel avec contrainte de sûreté. Cependant, à cause de cette forte synchronisation, quelques applications peuvent être sur-spécifiées, même avec un langage RS multi-horloges comme *Signal*, limitant ainsi les alternatives d'implémentation. Par ailleurs, disposer d'une horloge unique et totalement synchronisée (*Esterel* ou *Lustre*) couvrant la totalité des nœuds d'un système temps-réel distribué peut être extrêmement coûteux ou même simplement infaisable.

2. 4. Graphes de tâches :

Pour tenir compte des évolutions des SoC qui intègrent généralement plusieurs processeurs et au moins un système d'exploitation, différentes approches considèrent un modèle de type graphe de tâches avec un ordonnancement statique en ligne préemptif, par exemple [39, 40, 41, 42]. Les tâches sont généralement de forte granularité et traitées comme des boites noires avec des conditions de précédance. C'est le système d'exploitation qui gère l'exécution des tâches en utilisant ses propres structures de contrôle.

3. Modèles hybrides

Une approche de modélisation unifiée cherche à définir un modèle de calcul concurrent qui servirait à modéliser toutes les composantes d'un système. Ceci pourrait être possible en combinant tous les modèles précédents, mais une telle combinaison serait extrêmement complexe, difficile à définir et à utiliser (notons par exemple l'expérience *VCC*[1] de Cadence), et les outils de synthèse et de simulation seraient aussi difficile à concevoir.

Plusieurs modèles hybrides ont vu le jour, essayant de profiter des avantages de certains modèles de calcul pour masquer les inconvénients des autres. Parmi ces modèles ont peut citer:

1. VCC: Virtual Component Co-design

1. Les StateCharts et les CFSM

Les StateCharts [43] comme les CFSM[1] [44] peuvent être vu comme la combinaison des machines à états finis avec le modèle réactif synchrone. Les CFSM ont l'avantage d'être localement synchrones, globalement asynchrones ce qui simplifie grandement le partitionnement entre modules logiciels et matériels. L'outil *POLIS* [44] de co-conception logicielle/matérielle élaboré à l'Université de Berkeley est basé sur le modèle CFSM. Cet outil utilise *Esterel* pour décrire chaque CFSM, puis les interconnecte pour former un réseau hiérarchique de machines d'états finis réactives. Cet outil à été la base d'un autre outil, commercial cette fois-ci, de co-conception logicielle/matérielle appelé *VCC* qui a été commercialisé par Cadence.

2. Le modèle SDL

Le modèle SDL[2] peut être vu comme la combinaison des machines à états finis avec les réseaux de processus communicants [45]. C'est un formalisme graphique, basé sur les processus communicants qui permet de décrire le parallélisme d'une application tout en étant indépendant d'une implémentation logicielle ou matérielle. Chaque processus est une machine à états finis étendue par des variables. Le formalisme SDL est adapté à la description des protocoles de télécommunication. Parmi les approches de conception utilisant le modèle SDL, on peut citer [46, 11].

3. Le modèle PSM

Le modèle PSM[3] associe les machines d'états finis et les processus séquentiels communicants (CSP). Ce modèle s'exprime dans le langage SpecCharts, qui consiste en une extension de VHDL. L'approche SpecSyn détaillée dans [47] propose un environnement de conception au niveau système utilisant le modèle PSM. Un autre langage issu du modèle PSM est le langage SpecC. Ce langage se base sur un réseau hiérarchique de comportements et de canaux. C'est surtout une extension du C-ANSI pour la conception matérielle en intégrant les concepts de temps, de concurrence, de synchronisation.

4. Modèle MTG

Le modèle MTG[4] développé dans [48] est basé sur un modèle de comportement bas niveau sous forme de CDFG géré à plus haut niveau par un modèle de réseaux de Petri temporisés. Dans [48], les auteurs développent les points clés de ce modèle qui comprennent la hiérarchie, la synchronisation, la com-

1. CFSM : Co-design Finite State Machine
2. SDL : Specification and Description Langage
3. PSM : Program State Machine
4. MTG : Multi-Thread Graph

munication de données et la prise en compte des contraintes de temps. Ce modèle est la base d'une méthodologie de co-conception qui arrive jusqu'à la synthèse de RTOS.

5. Modèles au niveau transactionnel

- SystemC

SystemC est une librairie de classes construite à base du standard C++ et d'un noyau de simulation par événements [49]. Cette librairie de classes permet de modéliser des systèmes logiciels/matériels avec la possibilité de décrire la concurrence, une notion du temps et quelques types de données matériels spécifiques. Plusieurs outils commerciaux utilisent SystemC dans leurs flots de conception, parmi lesquels on trouve Cocentric de *Synopsys* et ConvergenSC de *CoWare*.

- SystemVerilog

SystemVerilog est un nouveau langage considéré comme une extension du langage de description matériel Verilog afin de supporter de plus hauts niveaux d'abstraction pour la modélisation et la vérification [50].

Un article intéressant faisant le point sur les langages SystemVerilog et SystemC a fait l'objet d'une session spéciale à DATE 2004 [51].

6. Modèles orientés objet

UML est le langage de modélisation orienté objet émergeant actuellement et utilisé principalement pour le développement de systèmes logiciels [52]. Il consiste en un ensemble de blocs de base, de règles qui dictent l'utilisation et l'arrangement de ces blocs et de mécanismes qui améliorent la qualité des modèles UML. Ce langage est en train de gagner l'attention de la communauté s'intéressant au logiciel embarqué qui le considère comme étant une solution possible pour travailler à un niveau d'abstraction plus élevé.

Plusieurs travaux ont été réalisés ces dernières années pour étudier la possibilité d'utiliser UML comme langage de base pour la spécification des systèmes embarqués [53, 54], ou plus particulièrement des plateformes embarquées [55].

L'approche détaillée dans [56] combine les modèles UML et SDL. La spécification au plus haut niveau est réalisée en UML et le comportement de chaque module est spécifié en SDL.

7. Divers autres modèles

L'approche développée dans [57] propose un formalisme pour décrire le concept d'asynchronisme dans le modèle RS Signal, ce qui permet de considérer le système comme étant globalement asynchrone.

Dans [58], les auteurs utilisent un modèle basé sur les machines d'états finis synchrones pour modéliser l'architecture de communication d'un SoC. Ce modèle se base sur le formalisme dit d'automates à protocoles synchrones.

Dans [59], le modèle considéré est basé sur les machines à états finis hiérarchiques (HFSM) couplées à un modèle flot de données synchrone (SDF) pour modéliser un SoC reconfigurable et gérer l'ordonnancement de ses reconfigurations dans le temps.

4. Les environnements de modélisation multi-paradigmes

1. Ptolemy

Une des initiatives les plus remarquables sur la définition d'un environnement d'accueil de différents paradigmes est le système Ptolemy [60] qui définit des sémantiques précises pour interfacer les différents modèles (domaines). Ptolemy est avant tout un environnement de modélisation et de simulation. Dans [61] est présentée une méthode de partitionnement basée sur deux domaines : FSM (finite state machine) et SDF. Une approche simplifiée est considérée dans [62]. Dans [63] un modèle mixte FSM/SDF est également considéré.

Parmi les améliorations apportées dans PtolemyII [64], on cite la notion de polymorphisme entre domaines (où les composants peuvent être conçus de manière à opérer dans des domaines différents), la notion de modèles modaux (où les FSM sont combinés hiérarchiquement avec d'autres modèles de calcul) et la notion de domaine à temps continu (qui, une fois combinée avec la modélisation modale, permet la modélisation de systèmes hybrides).

2. Metropolis

Cet environnement est construit à partir d'un métamodèle basé sur un modèle de calcul de type CSP. Dans ce métamodèle on peut englober un ensemble de modèles de calcul comme étant des cas particuliers qui en dérivent. *Metropolis* [65] propose un environnement de conception de systèmes hétérogènes qui intègrent des outils de simulation, de synthèse, d'analyse et de vérification.

3. Rosetta

Il s'agit d'un langage développé dans le cadre de l'initiative SLDL[1]. *Rosetta* [66] permet d'intégrer plusieurs théories de domaines en une structure à sémantique commune et d'organiser et de décomposer les contraintes au niveau système sur les différents domaines de sémantique différente que peut comporter le système. Notons qu'une initiative de standardisation du langage *Rosetta* supportée par Accellera [67] est en cours.

Conclusion

Tout au long de ce chapitre, nous avons vu qu'il n'existe pas de modèle idéal pour toutes les classes de systèmes. Par exemple, le modèle flot de données est plus naturellement utilisé pour les systèmes qui répètent les mêmes traitements sur des données périodiques, comme les systèmes de traitement numérique du signal. Les machines à états finis sont plus appropriées pour modéliser des systèmes qui doivent répondre à des séquences complexes d'événements externes, comme les systèmes dominés par le contrôle. Cependant, le meilleur modèle pour une application donnée est celui qui coïncide étroitement avec les caractéristiques du système qu'il modélise.

Dans la deuxième partie de ce manuscrit, nous présentons le modèle d'application que nous avons choisi en essayant de motiver ce choix. Dans la suite, nous explorons l'espace des architectures reconfigurables, prises comme architectures cibles de notre méthodologie. Le dernier chapitre de cet état de l'art détaille quelques méthodes de partitionnement logiciel/matériel qui nous semblent les plus pertinentes vis à vis de nos objectifs.

1. SLDL : System Level Description Langage

CHAPITRE 2

Les plateformes reconfigurables

Introduction

Nous ciblons dans notre étude des plateformes reconfigurables formées d'une unité reconfigurable connectée à un processeur hôte que ce soit à travers le bus système ou les canaux d'entrées/sorties. Avec l'introduction de ces nouvelles plateformes, il est maintenant possible de combiner sur la même puce deux gestions de calcul différentes : temporelle, sur le CPU et spatiale sur le composant reconfigurable. Ces architectures se révèlent donc être des plateformes particulièrement adaptées pour les applications embarquées qui combinent le contrôle et le calcul intensif. Les performances de ces architectures restent souvent limitées par les délais de communication et de reconfiguration. Les systèmes actuels essayent de résoudre ce problème en plaçant la logique reconfigurable à l'intérieur du cœur de processeur ou l'inverse, ce qui donne naissance à des nouveaux systèmes plus connus sous les noms de *CSoC*[1], *RSoC*[2], *SoPC*[3], *SoRC*[4] par exemple.

1. CSoC : Configurable System on Chip
2. RSoC : Reconfigurable System on Chip
3. SoPC : System on Programmable Chip
4. SoRC : System on Reconfigurable Chip

Les développements les plus récents de ces plateformes sont par exemple les Triscend[1] E5 et A7, le Xilinx Virtex II Pro [68] et l'Altera Excalibur [70]. Les capacités de ces plateformes sont assez variées, allant d'un processeur ARM à 60 MHz et 20.000 portes programmables avec le Triscend A7, à 4 processeurs PowerPC405 à 400 MHz, 10 millions de portes, 10 Mbits de RAM et 556 18*18bit multiplieurs pour le Xilinx Virtex II Pro 2VP125.

Un très grand nombre d'architectures reconfigurables, ou à base de reconfigurable, ont vu le jour pendant les dernières années, que ce soit dans l'industrie ou dans le monde de la recherche. Plusieurs études rétrospectives intéressantes sur ces architectures existent et parmi lesquelles on peut citer [71, 72, 73].

Dans ce chapitre, nous donnons un bref aperçu des différentes classes de plateformes reconfigurables, selon leurs types d'architecture, la granularité de leurs éléments de calcul, leurs types de reconfiguration (statique/dynamique, en distinguant entre différents modes de reconfiguration dynamique et leurs origines (projet industriel ou académique). Ensuite, nous présentons trois grandes familles d'architectures reconfigurables en se basant sur la granularité de leur reconfiguration c'est à dire les architectures reconfigurables au niveau porte, fonction et système.

1. Classification

Les architectures reconfigurables peuvent être classifiées selon différents paramètres. Dans ce paragraphe, nous distinguons les paramètres architecturaux qui, à notre avis, permettent le mieux de les différencier.

1. 1. La Granularité :

La granularité d'un composant reconfigurable représente la taille du plus petit élément fonctionnel. Les FPGAs ont typiquement une faible granularité avec des unités fonctionnelles à deux ou à quatre entrées (bien qu'il existe des architectures de FPGA intégrant des multiplieurs sur des mots de 18 bits). D'autres architectures reconfigurables implémentent des unités arithmétiques à plus gros grain (32 bits pour le Chameleon par exemple).

Typiquement, un composant reconfigurable est configuré en chargeant dans celui-ci une séquence de bits (ou *bitstream*). Le temps de chargement est directement proportionnel à la taille de la séquence. Les architectures à gros

1. http://www.triscend.com/

grain nécessitent des séquences plus courtes que les architectures à grain fin donc un temps de configuration plus faible.

Une faible granularité procure une plus grande flexibilité d'adaptation du matériel à la structure de calcul souhaitée mais elle entraîne un temps de configuration important. De plus, le grand nombre d'interconnexions entre les unités rajoute un délai conséquent à l'exécution du module implanté (malgré des structures permettant de réduire ce délai, comme les chaînes de propagation de retenue que l'on peut trouver maintenant sur quelques FPGA).

1. 2. Le couplage :

Une très grande partie des architectures reconfigurables est construite autour d'un processeur hôte. Celui-ci a généralement la charge des structures de contrôle qui configurent la logique, lancent les traitements, gèrent les transferts de données ainsi que les interfaces. Le degré de couplage entre le processeur et le reconfigurable a un impact direct sur les coûts d'accès aux données partagées et de reconfiguration.

- Couplage en mode périphérique : L'unité reconfigurable communique avec le processeur hôte à travers l'interface d'entrée/sortie, ce qui implique des temps de latence importants. Les calculs peuvent s'effectuer en recouvrement avec ceux du processeur et la configuration reste généralement stable pendant toute la durée d'un traitement.

- Couplage en mode co-processeur : L'unité reconfigurable est connectée au bus local du processeur hôte et partage le même espace mémoire. Les calculs sur le reconfigurable peuvent là aussi s'effectuer en recouvrement avec ceux du processeur dans la mesure ou il n'y pas de conflit au niveau des accès mémoire. La configuration, par contre, peut changer après la réception d'une interruption ou l'exécution d'une instruction particulière par le processeur.

- Couplage direct : L'unité reconfigurable est située sur le chemin de données interne du processeur hôte ce qui minimise au maximum les temps de latence. Les reconfigurations peuvent être très fréquentes ce qui nécessite un temps de reconfiguration de l'unité reconfigurable doit être très faible pour que le système présente un réel intérêt en terme de performances.

1. 3. La reconfigurabilité :

La reconfiguration est le processus qui consiste à charger une nouvelle configuration (ou contexte) dans l'unité reconfigurable pour en changer la fonctionnalité.

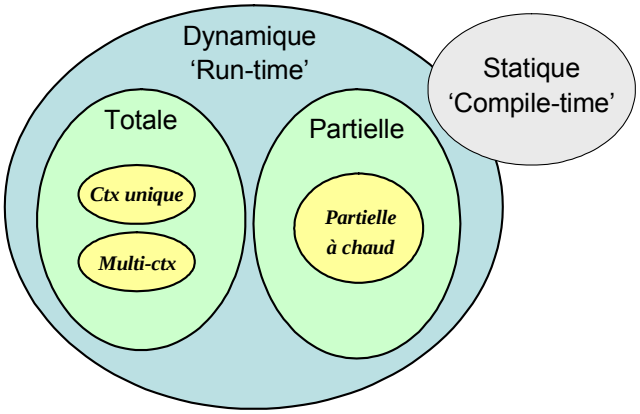


FIGURE 1.2. Les différents types de reconfigurabilité

Il existe deux types de reconfigurabilité :

- la reconfigurabilité statique (ou *compile-time reconfiguration : CTR*) : selon ce mode de reconfiguration, chaque application consiste en une configuration (ou un contexte) unique. Pour reconfigurer un tel système, on doit l'interrompre pour procéder à la reconfiguration et le redémarrer avec la nouvelle configuration.

- la reconfigurabilité dynamique (ou *run-time reconfiguration : RTR*) : Plusieures définitions peuvent être trouvées dans la littérature pour caractériser ce mode de reconfiguration. Nous allons plutôt le considérer comme un ensemble de modes regroupés au sein de la figure 1.2. Nous distinguons deux modes de reconfiguration dynamique selon la manière de construire des contextes :

  * la reconfiguration dynamique totale (ou *global RTR*) : chaque application est divisée en un ensemble de configurations temporelles distinctes dont chacune occupe la totalité des ressources matérielles du composant. On distingue deux types de composants reconfigurables dynamiquement et totalement selon la structure de leurs mémoires de configuration :

    > Les unités reconfigurables à contexte unique (schématisées par la ligne : Reconfiguration totale de la figure 1.3) : elles possèdent un plan de configuration unique (plan de mémoire SRAM contenant les données de configuration). Tout changement de contexte implique la reconfiguration totale du composant qui nécessite un temps de l'ordre de quelques millisencondes dans de nombreuses architectures de ce type. Ceci limite l'utilisation de ce genre de composants à des applications où la fréquence de changements au niveau matériel est relativement faible : de l'ordre de quelques heures, jours ou semaines.

> Les unités reconfigurables multi-contextes : elles possèdent plusieurs plans de configuration. Un plan de configuration unique peut être actif à un certain moment sur le reconfigurable mais le passage à un autre plan peut être très rapide (de l'ordre de quelques nanosecondes). Des projets de recherche industriels, comme ceux de Xilinx [76] ou de NEC [77], basés sur le principe de multiplexage temporel affichent des temps de l'ordre de 5 à 30 nanosecondes. Dans le même sens, des architectures comme *Garp* [78], *MorphoSys* [79] ou *Chameleon* [80] ont opté pour l'utilisation de la notion de cache de reconfiguration. Elle consiste à stocker un nombre de configurations dans des *buffers* (généralement en mémoire locale du composant) et de changer de contexte pendant l'exécution en un temps comparativement très faible par rapport à un chargement de contexte à partir d'une mémoire externe. L'architecture *Chameleon* par exemple, possède un cache permettant de stocker une configuration et de reconfigurer le composant à partir de ce cache en un seul cycle.

* la reconfiguration dynamique partielle (ou *local RTR*) : dans ce mode reconfiguration, on reconfigure sélectivement une *zone* du composant, ce qui réduit considérablement le temps de reconfiguration. On distingue deux principaux types de reconfiguration dynamique partielle :

> Les unités reconfigurables partiellement (schématisées par la ligne : Reconfiguration partielle de la figure 1.3) : Ces composants permettent de reconfigurer sélectivement une *zone* du composant alors que le reste du système reste inactif tout en conservant ses informations de configuration. Le temps de reconfiguration dans ce cas dépend de la surface en terme d'éléments logiques du contexte à reconfigurer

> Les unités à reconfiguration partielle *à chaud* ou *active (Active Partial Reconfiguration, Xilinx*) (schématisées par la ligne : Reconfiguration partielle à chaud de la figure 1.3) : permet de reconfigurer sélectivement une *zone* de l'architecture pendant que le reste du composant poursuit son fonctionnement. Elle est basée sur le concept de 'matériel virtuel' qui est similaire à l'idée de mémoire virtuelle. Plusieurs contextes peuvent être actifs à la fois sur le composant selon le nombre de *zones* actives. Le temps de reconfiguration d'un contexte sur une *zone* de ce composant est masqué par l'exécution d'un autre contexte sur une autre *zone*.

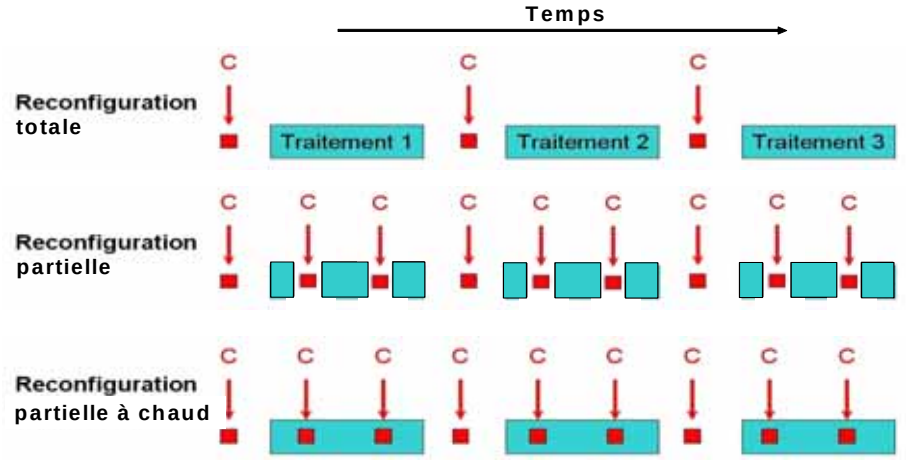


FIGURE 1.3. Les différents types de reconfiguration listés

La reconfiguration partielle à chaud pose de nouveaux défits intéressants liés par exemple à :

- l'allocation spatiale des ressources.

- au partage des ports d'entrée/sortie entre différentes *zones*.

- l'interfaçage des *zones* fixes avec les *zones* reconfigurables et entre deux *zones* reconfigurables voisines (problèmes de communication).

La reconfiguration en général offre d'énormes opportunités, en terme de performance, du fait qu'elle permet d'adapter plus étroitement le composant matériel à la nature du traitement, et en terme d'adaptabilité (flexibilité) du fait qu'elle permet de réagir à des stimuli venant :

- de l'environnement (une QoS[1] donnée par l'utilisateur comme dans les travaux faits à l'*IMEC* [74] utilisant la reconfiguration partielle)

- du système lui même (des résultats intermédiaires de calcul, des valeurs de capteurs...)

en changeant de mode (mode basse consommation, mode dégradé, mode haute résolution...).

Le problème de la reconfiguration basée sur des résultats de calcul intermédiaires (ou des estimées de ces résultats) est abordé dans [75] avec le cas des applications à temps d'exécution variable.

1. QoS : Quality of Service

Pour essayer de réduire (ou de masquer) le temps de reconfiguration de certains composants reconfigurables des architectures, comme celles d'Atmel, ont par exemple opté pour des techniques de compression de *bitstream* pour diminuer le temps de chargement des contextes sur le composant. La décompression matérielle permet un gain assez significatif. D'autres approches [81], plutôt logicielles, étudient les possibilités de masquer ces temps de reconfiguration en pré-chargeant (*Pre-fetching*) le prochain contexte avant d'en avoir le besoin. Cette technique nécessite une connaissance assez globale de l'application et une étude statique hors ligne préalable.

1. 4. L'organisation mémoire :

Les calculs exécutés sur l'unité reconfigurable s'effectuent sur des données rangées en mémoire dès que le volume de données traitées est important. Des résultats intermédiaires de calcul doivent aussi être sauvegardés avant que la logique ne se reconfigure pour exécuter le prochain traitement. L'organisation de la mémoire a donc un grand impact sur le coût d'accès aux données. Cette mémoire peut être centralisée dans de larges blocs comme les *BlockRAM* de la famille *Virtex* de Xilinx, ou distribuée comme dans le cas du *Chameleon* ou du *Stratix* d'Altera.

2. Les architectures reconfigurables au niveau porte

Les architectures reconfigurables au niveau porte les plus répandues sont des architectures à base de FPGA. Il existe deux types de technologies pour ces composants reconfigurables selon que la configuration est effectuée en utilisant des éléments *Anti-Fuse* ou des mémoires *SRAM*. La technologie *Anti-fuse* utilise un fort courant électrique pour créer la connexion entre deux fils et est généralement non-reprogrammable. La technologie basée sur les *SRAM*

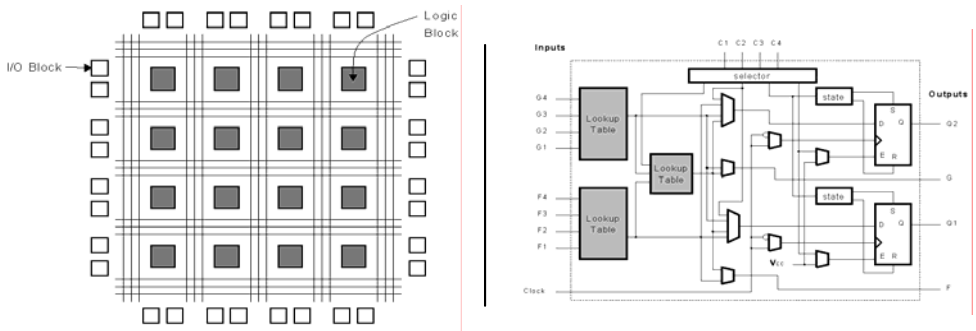


FIGURE 1.4. Architecture générique d'un FPGA et de son bloc logique

permet de reconfigurer le composant en chargeant différents bits de configuration dans les cellules mémoire *SRAM*. Une étude détaillée sur les architectures des FPGAs et des CPLDs[1] est disponible dans [82].

Dans la suite de ce manuscrit, nous désignerons par le terme FPGA, ceux à base de SRAM. Ils sont constitués d'une matrice de blocs de logique combinatoire traversés par un réseau d'interconnexion et qui communiquent avec l'extérieur à travers des blocs d'entrée/sortie. Les fonctions exécutées par les blocs logiques, les blocs d'entrée/sortie ainsi que le réseau d'interconnexion peuvent être indéfiniment reconfigurés en chargeant à chaque fois la séquence de bit (ou *bitstream*) correspondante (voir figure 1.4). Ces unités reconfigurables ont été utilisées pour construire des processeurs et co-processeurs dont l'architecture interne et les interconnexions peuvent être configurées pour s'adapter aux besoins de l'application visée.

Un bloc logique contient typiquement une (ou plusieurs) LUT[2], une (ou plusieurs) bascule(s), de la logique combinatoire supplémentaire (par exemple pour la propagation de la retenue, ou la mise en cascade) et des cellules mémoire SRAM pour contrôler la configuration du bloc. Le bloc logique de base pour Xilinx est le bloc logique configurable (*CLB*[3]) (détaillé dans la figure 1.4) et celui d'Altera est l'élément logique (*LE*[4]). Ces deux fabricants ont doté leurs dernières générations de composants d'une capacité d'intégration telle qu'on retrouve par exemple jusqu'à 225 multiplieurs 18*18, 114K *LEs*, 28 blocs DSP et 10 Mo de RAM disponibles sur le *Stratix* [83] EP1S120 d'Altera et une capacité presque équivalente pour le *Virtex II* [68] de Xilinx (figure 1.5). Ces nouvelles architectures rajoutent un niveau hiérarchique en regroupant les cellules logiques au moyen d'une matrice d'interconnexions locales et en intercalant des structures de mémorisation qui accélèrent les communications entre proches voisins.

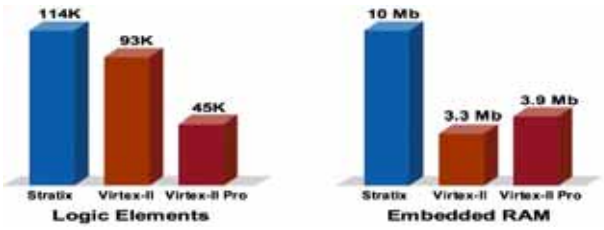


FIGURE 1.5. Comparatif en terme de ressources

1. CPLD: Complex Programmable Logic Device
2. LUT : Look-Up Table
3. CLB : Configurable Logic Block
4. LE : Logic Element

L'architecture du *Stratix* est par exemple largement optimisée pour les calculs intensifs en intégrant deux colonnes de blocs RAM de part et d'autre des blocs DSP pour alimenter ces blocs en données et stocker les résultats des traitements et les données intermédiaires. Pour accélérer encore plus ces calculs, le concepteur peut tirer profit de la diversité des blocs mémoire, plusieurs tailles de blocs sont disponibles (512, 4K octets et 512K octets) ainsi que des bandes passantes différentes en considérant le nombre de ports de ces mémoires (respectivement de 4, 2 et 1).

L'une des architectures FPGA les plus intéressantes est le Xilinx *6200* [84] qui offrait des propriétés de reconfiguration dynamique partielle à chaud rapide (de l'ordre de 100µs pour reconfigurer totalement l'unité) et une interface pour une connexion directe au bus système du processeur hôte. Mais Xilinx a cessé de produire cette architecture parce que les outils sous-jacents n'arrivaient pas à prendre en compte les capacités de cette architecture.

Atmel a proposé deux familles de composants qui offrent aussi la propriété de reconfiguration dynamique partielle à chaud : l'*AT40K* [85] et l'*AT6000* [86]. De plus, Atmel a doté ces architectures d'un cache logique [87] qui consiste à stocker un certain nombre de plans de configurations dans le cache qui, associé à un algorithme de compression du *bitstream*, réduit considérablement le temps de reconfiguration. L'architecture *Ardoise* [88] est un exemple de plateforme multi-FPGA *AT40K* reconfigurable essentiellement en mode ping-pong (un FPGA est en train de traiter pendant que l'autre est reconfiguré pour le traitement suivant) mais elle ne permet pas de tirer profit de toutes les capacités offertes par l'*AT40K*.

Une architecture devenue désormais classique est celle de la famille *Virtex* de Xilinx (qui englobe les composants Virtex, Virtex-E, Virtex-II, Virtex-II Pro, Virtex-4) [89]. L'organisation générale consiste en une matrice de blocs logiques (CLBs) structurée en colonnes intercalées avec des colonnes de blocs dédiés (mémoire *BlockRAM* de 4096 bits chacun, des blocs DSP, des cœurs de processeurs) et entourée par des blocs d'entrée/sortie programmables. C'est une architecture qui supporte la reconfiguration dynamique partielle à chaud (appelée reconfiguration partielle *active* [90]), basée soit sur la notion de *modules*, soit sur la notion de différence. La première permet de reconfigurer dynamiquement un *module* formé d'un minimum de 4 colonnes de *slices* (sur toute la hauteur du composant) et un maximum allant jusqu'à la largeur totale du composant avec chaque fois des incréments de 4 colonnes. Toutes les ressources logiques (IO-Blocks, ressources de routage, *BlockRAM*, blocs DSP...) contenues dans la largeur du *module* considéré font partie de son *bitstream*. Chaque *module* communique avec le *module* voisin (qu'il soit fixe ou lui aussi reconfigurable) a travers une macro de bus spéciale détaillée dans [90]. La reconfiguration partielle *active* peut aussi être basée sur la notion de différence lorsque les changements à faire entre l'ancienne et la

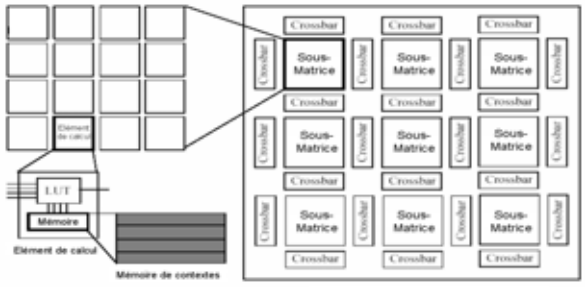


FIGURE 1.6. L'architecture bloc du DPGA

nouvelle configuration sont minimes. Dans ce cas, on charge dans le composant un *bitstream* correspondant à la différence entre les deux versions, ce qui réduit le temps de reconfiguration.

L'étude détaillée dans [91] présente une méthodologie et les outils associés pour gérer la reconfiguration dynamique partielle à chaud de la famille Virtex-E basée sur l'outil PARBIT.

Une architecture qui serait intermédiaire entre les architectures reconfigurables au niveau porte et celles reconfigurables au niveau fonctionnel est l'architecture du DPGA[1] [92]. Sa structure consiste en une matrice de cellules (sous-matrices) de calcul hiérarchisée comme illustré dans la figure 1.6. Chaque sous-matrice est elle-même composée d'une matrice d'éléments de calcul de base, chacun formé par une LUT et une DRAM locale 4*32 bits permettant de stocker 4 contextes de configuration de la LUT ainsi que du réseau d'interconnexion. A chaque cycle, un identifiant de contexte à 2 bits est simultanément envoyé à tous les éléments ce qui permet, en un cycle (similairement à une instruction), de basculer vers le contexte correspondant et l'exécuter. Puisque la reconfiguration est au niveau des LUT et de leur interconnexion, cette architecture a été classée comme reconfigurable au niveau porte. Mais on peut considérer que la notion de contrôleur de reconfiguration (très simplifié dans le cas du DPGA) est assez représentative des architectures appartenant à la deuxième classe d'architectures reconfigurables, comme cela est détaillé dans le prochain paragraphe.

3. Les architectures reconfigurables au niveau fonctionnel

Les architectures reconfigurables au niveau fonctionnel comportent des blocs dédiés ou reconfigurables à gros grain reliés par un réseau d'interconnexion reconfigurable. Dans la suite sont présentées quelques architectures reconfi-

1. DPGA : Dynamically Programmable Gate Array

gurables au niveau fonctionnel, en montrant leurs spécificités et leur domaines d'utilisation.

MorphoSys : L'architecture MorphoSys [93] a été développée par les Universités de Californie (Irvine) et de Rio de Janeiro. Elle est constituée d'une matrice 8*8 de cellules reconfigurables (RC), une mémoire de contextes, un cœur de processeur RISC (Tiny_RISC), un buffer de données et un contrôleur DMA comme illustré par la figure 1.7. Excepté la mémoire globale, MorphoSys est un SoC complet.

Le processeur contrôle les opérations de base de la matrice reconfigurable à travers des instructions spéciales ajoutées à son jeu d'instructions. Il lance aussi les transferts de données depuis et vers le buffer de données ainsi que les chargements des configurations dans la mémoire de contexte et à partir de celle-ci.

Cette architecture permet d'atteindre des performances intéressantes du fait qu'elle est optimisée pour réduire (ou au moins masquer) les délais dus par exemple aux reconfigurations et aux transferts de données. En effet, MorphoSys permet la reconfiguration de la matrice en un cycle à partir de la mémoire de contextes, et permet aussi d'avoir un recouvrement entre traitement et transferts de données grâce à l'utilisation du *frame buffer.*

Cette architecture est considérée à gros grain puisqu'elle opère sur des données de largeur 8 à 16 bits bien que chaque RC intègre une unité à grain fin (*FGB*[1] qui est une sorte de petit FPGA) opérant au niveau bit. La version M2 de l'architecture a été optimisée pour les applications de télécommunication sans fil.

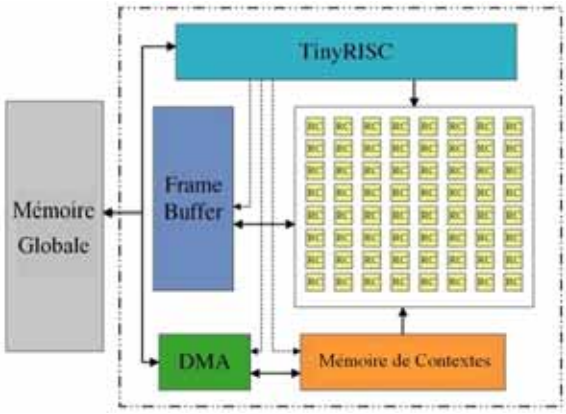


FIGURE 1.7. L'architecture bloc de MorphoSys (M2)

1. FGB : Fine Grain Block

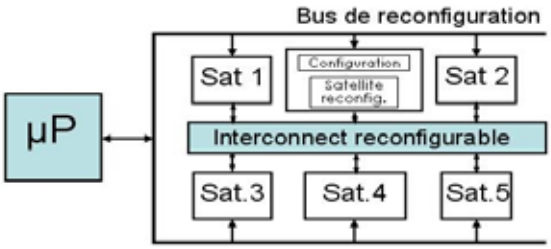


FIGURE 1.8. L'architecture bloc de Pleiades

L'étude détaillée dans [94] propose un compilateur ciblant l'architecture MorphoSys. Il exploite le parallélisme aux niveaux grain fin et gros grain pour construire un unique contexte qui s'exécute sur la matrice reconfigurable. Cette méthode utilise un algorithme de partitionnement de type *Clique-Partitionning*.

Pleiades : Cette architecture a été développée à Berkeley. Pleiades [95] a été conçue dans un souci de faible consommation (différentes tensions d'alimentation sont supportées, un mode basse consommation...). Elle est basée sur un réseau d'interconnexion de type *Crossbar* qui relie un processeur hôte et des unités d'exécution de granularités différentes (satellites) ainsi que de la mémoire (figure 1.8). Pour chaque domaine d'application, une nouvelle instance d'architecture peut être créée en changeant le type et le nombre d'unités satellites. L'architecture *MAIA* [96] par exemple a été conçue pour le codage de parole et intègre un FPGA, un ARM8 et de la mémoire.

La méthode de partitionnement proposée dans [97] ciblant cette architecture part d'une implémentation entièrement logicielle sur le processeur hôte. Si les contraintes de temps et de consommation ne sont pas respectées, les tâches sont migrées manuellement une par une, vers les unités satellites, ou en utilisant la formulation ILP proposée par Kalavade dans [98].

Dans le monde industriel, la jeune entreprise californienne *MorphICs* a repris le concept de *Pleiades* pour proposer une solution architecturale pour les terminaux 2.5G et 3G [99]. *Infineon* a fait l'acquisition de cette société en 2003.

Garp (Berkeley) : L'architecture Garp [78] combine une unité de calcul reconfigurable à un processeur MIPS. L'unité reconfigurable est sous la forme d'une matrice de blocs logiques organisée en 32 colonnes de 24 blocs chacune. Un des 24 blocs est un bloc de contrôle et les blocs restants sont des blocs logiques de granularité fine opérant sur des données de largeur 2 bits.

Le chargement et l'exécution des contextes de configuration sur la partie reconfigurable sont contrôlés par le processeur par un jeu d'instruction étendu. Le couplage entre le processeur et l'unité reconfigurable est un cou-

plage en mode coprocesseur. La reconfiguration peut être partielle dans le sens ou on peut configurer une colonne de la matrice pendant que les autres colonnes exécutent leurs traitements. Le composant est aussi doté d'un cache de configuration pouvant supporter 4 configurations pour chaque colonne. La reconfiguration à partir de ce cache ne prend que 4 cycles.

Dans [100], les auteurs proposent un compilateur pour cette architecture, mettant en œuvre des techniques utilisées pour les compilateurs d'architectures VLIW afin d'identifier le parallélisme au niveau instruction dans le programme source. Il construit également un ordonnancement de l'exécution des parties de l'application sur l'unité reconfigurable.

CS2000 (Chameleon Systems [80]*)* : Cette plateforme reconfigurable multi-applications, multi-protocoles (RCP[1]) de télécommunication intègre un processeur ARC 32 bits et une unité de calcul reconfigurable (RPF[2]) qui opère sur des données de largeur 32 bits (figure 1.9). L'unité RPF est organisée en 4 *tranches* pouvant être reconfigurées indépendamment les unes des autres. Chaque *tranche* contient trois *tuiles* qui intègrent chacune sept unités de calcul sur 32 bits, deux multiplieurs 16*24 bits, quatre mémoires locales et une unité de contrôle logique.

Deux chaînes d'outils de programmation distinctes sont utilisées pour programmer ce composant. La première chaîne cible le RPF et prend comme langage de spécification une description en *Verilog* (des travaux sont en cours pour intégrer cette chaîne dans un outil de plus haut niveau comme *Matlab*).

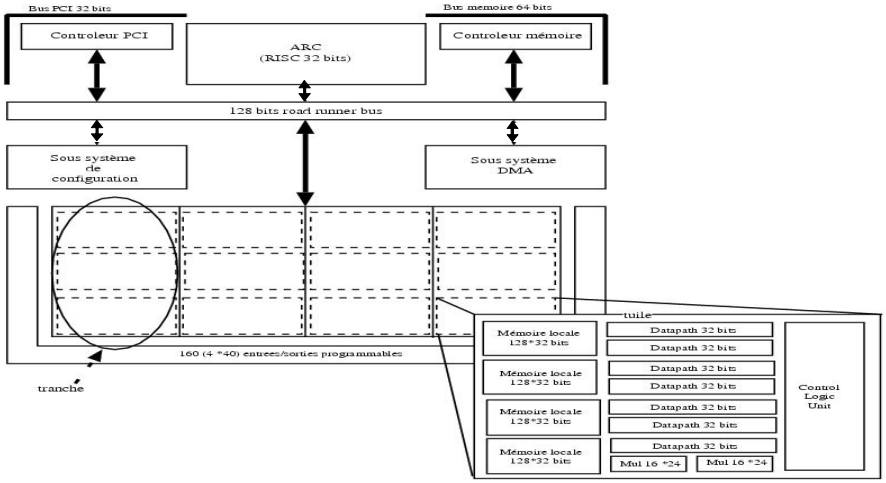


FIGURE 1.9. L'architecture du SC2000

1. RCP : Reconfigurable Communication Processor
2. RPF : Reconfigurable Processing Fabric

La deuxième chaîne cible le processeur ARC, elle est basée sur les outils GNU en ajoutant des librairies, appelée eBIOS, qui tournent sur le processeur ARC pour contrôler le RPF. Ces librairies permettent au concepteur de charger les contextes de configuration (*bitstream*), initier le calcul sur le RPF, contrôler son état, etc.

MRC6011 (Motorola): Motorola[1] a proposé en 2002 de combiner son DSP MSC8126 avec un nouveau composant reconfigurable (RCF[2]) : le *MRC6011*, comme solution pour les stations de base 3G. Le *MRC6011* a été conçu pour les calculs intensifs des applications en bande de base, les antennes adaptatives, la détection multi-utilisateur, etc.

Le *MRC6011* est constitué de 6 cœurs reconfigurables identiques structurés en deux modules. L'unité globale est capable d'atteindre des fréquences de fonctionnement de 250 MHz et des performances de l'ordre de 24 GMACS[3]. Chaque cœur reconfigurable est constitué d'une matrice de 16 éléments de calcul contrôlés par un processeur RISC optimisé, une mémoire de contextes, comme indiqué sur la figure 1.10.

Chaque élément de calcul contient une unité MAC 16*16 bits, une unité de corrélation complexe et opère sur des données de largeur 16 bits, donc de granularité assez élevée.

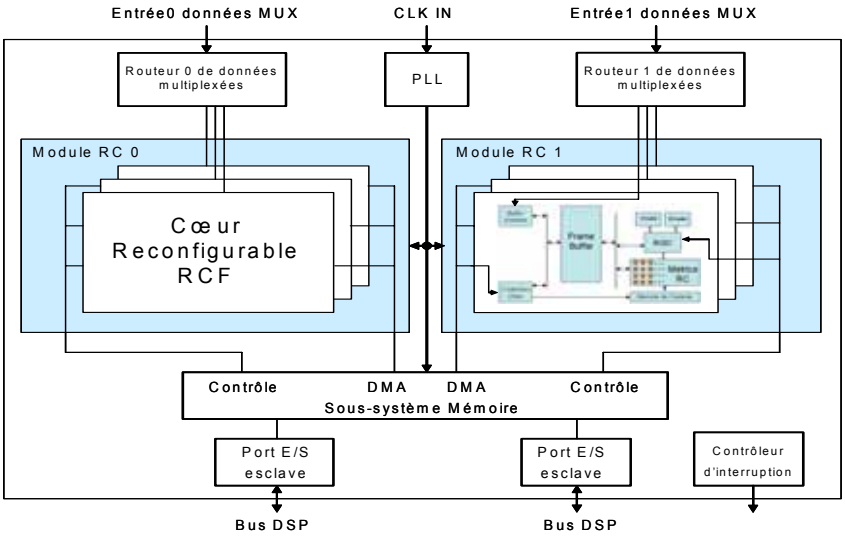


FIGURE 1.10. L'architecture bloc du MRC6011

1. http://motorola.com/semiconductors/
2. RCF : Reconfigurable Compute Fabric
3. GMACS : Giga Multiplication/ACcumulation par Seconde

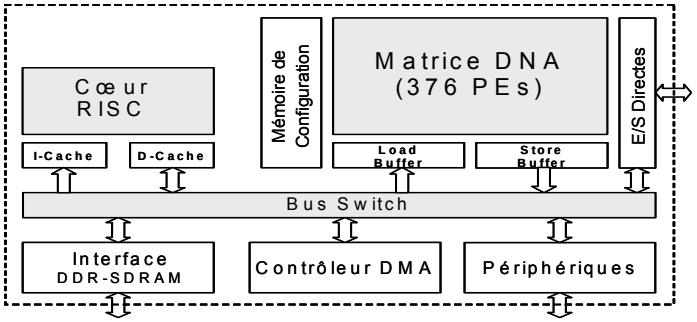


FIGURE 1.11. L'architecture bloc du DAP/DNA_2

DAP/DNA (IPFlex): *IPFlex*[1], une filiale de *Fujitsu*, a proposé dernièrement sa nouvelle version de plateforme dynamiquement reconfigurable : la DAP/DNA2. Cette plateforme est constituée d'un processeur RISC connecté à une matrice d'éléments de calcul (PE[2]).

Le processeur traite des parties de l'application et contrôle la reconfiguration dynamique de la matrice de calcul DNA. Il lui délègue le traitement d'autres parties nécessitant une grande capacité de calcul, ce qui permet d'avoir du parallélisme et du pipeline dans les traitements.

La matrice DNA permet, grâce à une mémoire de contextes, une reconfiguration en un cycle des 376 éléments de calcul ainsi que de leur interconnexion (figure 1.11). Chacun de ces PE opère sur des données de taille 32 bits donc de granularité assez importante. L'environnement de développement *DAP/DNA-FW II* permet, en partant d'une spécification de haut niveau (de type MATLAB Simulink de *MathWorks*), d'assister le concepteur dans les différentes étapes du processus de développement jusqu'à l'implantation sur le composant.

Adapt2400 (QuickSilver Technology) : QuickSilver[3] a présenté au début de l'année 2004 un composant de calcul adaptatif (ACM[4]) innovant avec un ensemble d'outils permettant au concepteur de développer rapidement des plateformes hétérogènes complexes. Cette architecture cible essentiellement le domaine du traitement du signal et des images avec un souci de basse consommation.

1. http://www.ipflex.com/
2. PE: Processing Element
3. http://www.quicksilvertech.com/
4. ACM: Adaptive Computing Machine

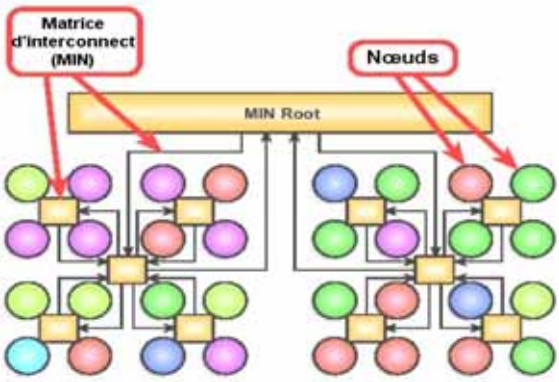


FIGURE 1.12. Structure hiérarchique de l'Adapt2400

L'architecture schématisée par la figure 1.12 est constituée d'une matrice hiérarchique d'éléments de calcul algorithmique (appelés nœuds) connectés ensemble via un réseau d'interconnexion (MIN[1]).

Il existe quatre types de nœuds prédéfinis et un type qui peut être défini par le concepteur en insérant son propre IP dans la partie algorithmique du nœud. Ces nœuds sont de granularités diverses:

- Le type DBN est optimisé pour les manipulations au niveau bit et mot de faible taille. Il atteint des vitesses de traitement de l'ordre de 45 GOPS[2].

- Le type AXN s'adapte à un ensemble de traitements systématiques sur des mots de largeur 8 à 32 bits.

- Le type PSN est un nœud de contrôle complexe dédié à la gestion de protocoles, du système d'exploitation, etc. Typiquement un RISC évolué.

- Le type XMC est un contrôleur mémoire intelligent.

Un réseau MIN d'un ACM peut supporter 32 nœuds et il est possible de relier en cascade jusqu'à quatre ACM pour atteindre un nombre maximum de 128 nœuds pouvant travailler à une fréquence de 250 MHz dans une technologie 0.13 µm.

1. MIN: Matrix Interconnect Network
2. GOPS : Giga Operations Par Seconde

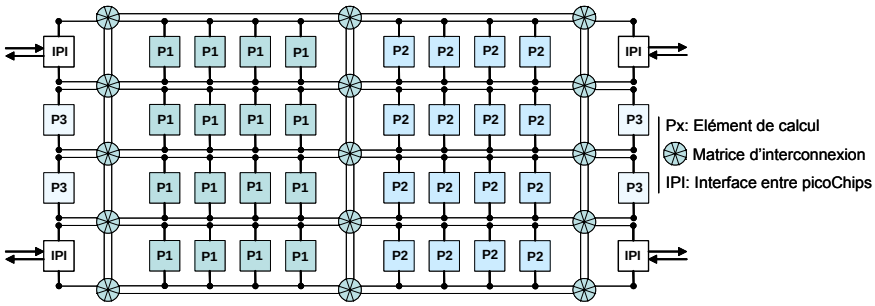


FIGURE 1.13. Structure du PC102

PC102 (picoChip) : Le PC102[1] est une matrice de 322 éléments de calcul (AE[2]) interconnectés par un bus (picoBus) de largeur 32 bits et des *switch* programmables comme illustré par la figure 1.13. Ces AE sont de granularité diverse et peuvent être classés selon quatre types différents :

- Standard : ce sont des processeurs RISC 16 bits programmables. Ils sont au nombre de 240.

- Mémoire : ce sont des éléments qui possèdent une unité de multiplication et de la mémoire supplémentaire. Ils sont au nombre de 64.

- Accélérateur : ce sont 14 coprocesseurs optimisés pour accélérer certains types de traitements.

- Contrôle : ce sont 4 éléments optimisés pour l'implémentation des fonctions de contrôle dans les stations de base.

Le composant peut atteindre une fréquence de fonctionnement de 160 MHz et une capacité de calcul de 197 GIPS[3].

LineDancer (Aspex => Philips): Philips[4] à dévoilé en 2003 sa solution architecturale pour la SDR[5]et les applications sans fil. Elle est sous la forme d'un processeur SIMD[6] massivement parallèle et entièrement reconfigurable de façon logicielle.

1. http://www.picochip.com/
2. AE : Array processing Element
3. GIPS : Giga Instructions Par Seconde
4. http://www.semiconductors.philips.com/
5. SDR : Software Defined Radio
6. SIMD : Simple Instruction Multiple Data

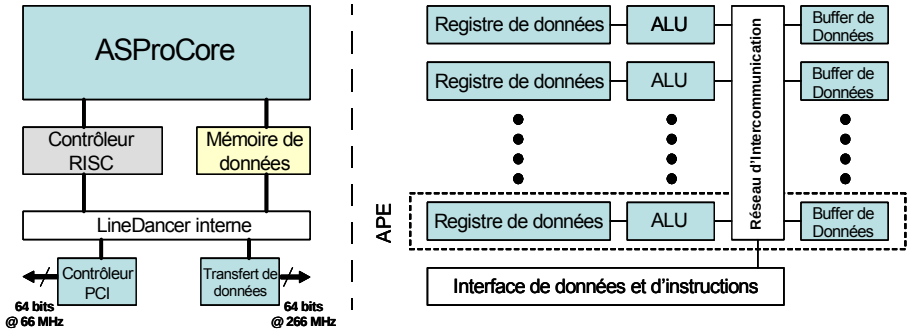


FIGURE 1.14. Diagramme Bloc du LineDancer et de l'ASProCore

Ce processeur est constitué principalement d'une unité reconfigurable : le ASProCore, d'un contrôleur RISC (une implémentation de l'architecture SPARC), 128 Kbytes de mémoire et d'interfaces E/S.

Le ASProCore est couplé au contrôleur RISC en mode co-processeur. Il exécute un ensemble d'instructions qui correspondent à des extensions du jeu d'instructions du processeur. L'ASProCore est implémenté comme un processeur sériel au niveau bit, et parallèle associatif au niveau mot c'est à dire que tous les éléments de calcul (APE[1]) exécutent la même opération logique ou arithmétique de manière sérielle sur un bit. Chaque APE représente un canal de traitement avec un registre de données et une ALU sérielle au niveau bit.

La reconfiguration est dite dynamique et ne concerne que le réseau d'interconnexion et l'alimentation des registres de données des APEs (de taille variable). Si on se fixe l'APE comme brique de base du composant, la granularité est dans ce cas très fine et si on considère un nombre de canaux (d'APEs) parallèles correspondants à la largeur du mot à traiter on peut considérer la granularité comme adaptative. La figure 1.14 présente l'architecture générale du *LineDancer* et d'un APE. On peut avoir jusqu'à 4096 APEs et l'ensemble peut atteindre des fréquences de fonctionnement de 300 MHz.

A7S (Triscend) : La famille A7 de Triscend[2] intègre un processeur ARM7TDMI de fréquence 60MHz, 16 Ko de mémoire SRAM *scratchpad*, un contrôleur DMA et 448 à 2048 cellules logiques (CSL[3]). Le couplage est de type coprocesseur. Les CSL opèrent sur des données de largeur 32 bits ce

1. APE: Associative Processing Element
2. http://www.triscend.com/
3. CSL: Configurable System Logic

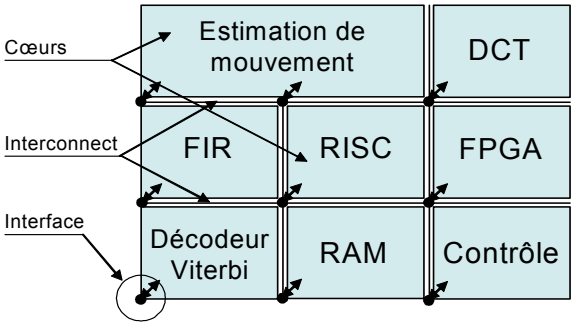


FIGURE 1.15. L'architecture aSoC

qui représente une granularité assez élevée. En mars 2003, *Xilinx* a fait l'acquisition de la société *Triscend*.

L'aSoC (Université du Massachusetts) : C'est une architecture en tuiles basée sur un réseau maillé d'interconnexions point à point pipelinées [101, 102]. Comme illustré par la figure 1.15, les tuiles sont des cœurs de traitement hétérogènes. L'un des avantages qu'offre cette architecture est qu'elle supporte des transferts de données ordonnancés que ce soit de manière statique hors ligne aussi bien que dynamique en ligne.

Dans [103], les auteurs partent d'une spécification de haut niveau de l'application exprimée en langage C qu'ils traduisent en une représentation intermédiaire sous forme de graphe de flots de données et de contrôle hiérarchique (HCDFG). Une étape d'exploration architecturale estime ensuite les performances de chaque cœur de calcul. Partant de SUIF de l'université de Stanford pour générer un graphe de flot de données et de contrôle (CDFG), l'environnement *AppMapper* permet d'annoter des parties du code (blocs de base) avec le cœur cible désiré (les choix son fait à la main sur la base d'estimations en temps réel) et d'allouer les parties du code assignées à un même type de cœurs en utilisant un algorithme glouton.

DART (Université de Rennes 1) : DART [104] est une architecture à reconfiguration dynamique partielle à chaud conçue pour traiter les applications de télécommunication mobiles de prochaine génération. L'architecture est composée d'un contrôleur de tâches, de ressources de mémorisation et de ressources de calcul (*Clusters*). Le contrôleur de tâches est chargé d'assigner aux *clusters* les différents traitements à exécuter. Une fois configuré, chaque *cluster* travaille de manière autonome et gère ses propres accès à la mémoire de données. La granularité de chaque *cluster* est multiple puisqu'il intègre un cœur de FPGA à grain fin et 6 DPRs[1] opérant sur des données de largeur 8 à 16 bits. L'application est spécifiée en utilisant le langage C. Le flot global est

1. DPR: DataPath Reconfigurable

basé sur le compilateur SUIF, et sur le compilateur reciblable CALIFE [105] développé à l'IRISA.

Rampass (CEA-List) : Rampass [106] est une architecture reconfigurable originale développée par le laboratoire LCEI du CEA Saclay. Les principales innovations de cette architecture reposent sur une séparation des ressources de contrôle (RAC: Reconfigurable adapté au contrôle) et des ressources de calcul (RAO: Reconfigurable adapté aux calculs). La partie RAC permet d'implanter des graphes de Petri. Le contrôle est alors géré par la propagation des jetons dans ces graphes.

Dans le cadre de l'EPML POMARD[1], un travail intéressant a été initié sur l'intégration d'un ou de plusieurs clusters de DART au sein de l'architecture Rampass sous forme d'une ressource RAO. DART se chargerait de la partie traitement et Rampass du contrôle des reconfigurations et des chemins de données.

Systolic Ring (Université de Montpellier) : L'architecture Systolic Ring [107] est une architecture à reconfiguration dynamique partielle à chaud, à gros grain. L'unité de calcul de base de cette architecture est le D-Node qui consiste en un Multiplieur et une UAL câblés travaillant sur des mots de 16 bits stockés dans une file de registres 4*16 bits. Chaque D-Node a un contrôleur local qui possède une mémoire de configuration pouvant stocker jusqu'à huit instructions.

XPP 64-A (PACT[2]) : Le XPP 64-A est un composant reconfigurable composé d'une matrice 8*8 d'éléments de calcul de type ALU (ALU-PAEs[3]) et de deux colonnes d'éléments mémoire (RAM-PAEs), de quatre interfaces d'E/S et d'un gestionnaire de configurations comprenant un cache de configurations.

4. Les architectures reconfigurables au niveau système

Les architectures reconfigurables au niveau système peuvent être schématisées par l'exemple de la figure 1.16 relative au système S5000. Ce sont plutôt des processeurs programmables (ou configurables) auxquels sont associés un ensemble de ressources de calcul spécifiques au domaine d'applications visé, un contrôleur unique et un banc de registres central. L'une des architectures

1. EPML POMARD : http://www.pomard.enssat.fr/
2. http://www.pactxpp.com/
3. PAE : Processing Array Element

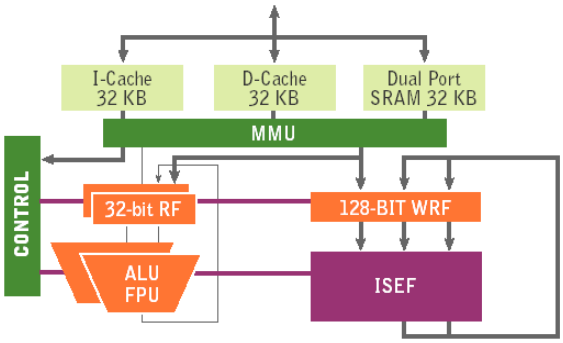


FIGURE 1.16. L'architecture bloc du S5000

reconfigurables au niveau système les plus prometteuses est cette architecture S5000 de *Stretch Inc.*

S5000 (Stretch Inc.) : *Stretch Inc.*[1] a proposé au début de l'année 2004 son processeur reconfigurable de façon entièrement logicielle basé sur le processeur RISC Xtensa de *Tensilica*[2] et d'un module reconfigurable (ISEF[3]) représentant un ensemble d'extensions du jeu d'instructions du processeur RISC (voir figure 1.16). L'unité ISEF est une sorte de FPGA conçue spécialement pour implémenter des ALUs de tailles variables, des multiplieurs, des unités de décalage, etc. L'architecture du S5000 permet au concepteur de définir de nouvelles instructions optimisées pour le type d'applications visées (généralement des applications de calcul intensif) en les codant en C/C++.

Le couplage dans ce cas est direct entre le processeur hôte et l'unité reconfigurable, la granularité est variable puisque l'ISEF peut opérer sur des mots de largeur allant de 8 à 384 bits.

Conclusion

Dans ce chapitre nous avons analysé différentes architectures reconfigurables en passant par celles désormais classiques (MorphoSys, Garp, Pleiades, Chameleon, XPP...) et par quelques nouvelles architectures, parfois inspirées de ces dernières, parfois assez innovantes. Nous avons choisi de les classer selon la granularité de leur reconfiguration en essayant de faire le point sur les outils dédiés disponibles.

1. http://www.stretchinc.com/
2. http://www.tensilica.com/
3. ISEF : Instruction Set Extension Fabric

Nous avons pu voir dans ce chapitre que les architectures reconfigurables au niveau porte ont un sérieux désavantage relatif au temps de reconfiguration assez conséquent (malgré les techniques de compression de *bistream*, les capacités de reconfiguration partielle...) par rapport à celles reconfigurables au niveau fonction ou système. D'un autre côté, les architectures reconfigurables au niveau fonction ont tendance à perdre de leur flexibilité en intégrant des types de fonctions de base dédiés à un domaine d'application. Pour permettre de garder cette flexibilité, ces architectures sont pénalisées par un surcoût en surface lié à l'intégration d'un contrôleur local de configuration (qui peut se limiter à une mémoire de contextes) pour chaque fonction. Quant aux processeurs reconfigurables au niveau système, ils restent aujourd'hui incapables de constituer à eux seuls une solution architecturale embarquée complète que ce soit du point de vue des performances de calcul que de la consommation d'énergie (lecture, décodage des instructions et accès aux données dans les larges banc mémoire).

Dans le dernier chapitre de cette partie, nous nous intéressons à quelques approches automatiques de partitionnement logiciel/matériel que nous avons rencontrées dans la littérature et ce en les analysant suivant les techniques utilisées pour l'allocation, l'affectation et l'ordonnancement qui sont les étapes principales du partitionnement.

CHAPITRE 3

Partitionnement logiciel / matériel

Introduction

Le partitionnement automatique d'une spécification est un problème complexe (il s'agit d'un problème NP-difficile) du fait du grand nombre de paramètres à considérer. Dans la suite nous passons en revue quelques approches de partitionnement en les distinguant en fonction de leurs objectifs d'optimisation et des applications et architectures ciblées.

La conception de systèmes enfouis spécialisés basés sur la technologie reconfigurable souffre de l'absence de méthodes et d'outils capables d'exploiter efficacement le parallélisme et les possibilités de reconfiguration dynamique qu'offrent ou vont offrir ce type d'architectures. L'une des principales étapes de conception est celle de partitionnement. Dans le cas général, cette étape se subdivise en trois parties principales :

- Une partie qui effectue l'*allocation* en choisissant le type et le nombre de ressources matérielles et logicielles nécessaires.
- Une partie qui effectue le *partitionnement spatial* en prenant des décisions sur l'affectation (assignation) des tâches sur le matériel et le logiciel.
- Une partie qui effectue l'*ordonnancement* des exécutions des tâches et des communications.

Cependant, dans le cas des architectures reconfigurables, une nouvelle partie s'intercale entre partitionnement et ordonnancement :

- Une partie qui effectue le *partitionnement temporel* en agençant les tâches matérielles dans des segments temporels (ou contextes) différents.

Du fait de la complexité du processus de partitionnement, plusieurs approches ont préféré laisser au concepteur le soin de déterminer une architecture (étape d'allocation) et de choisir une répartition des fonctionnalités de l'application sur celle-ci (étape de partitionnement spatial). Dans ce cas, les travaux visent à apporter une aide à la conception par l'intermédiaire d'un environnement de co-simulation qui permet d'animer le modèle de l'application en fonction des paramètres de l'architecture choisie. Ce type d'approche est utilisé dans le projet *POLIS* [44] qui a servi de base au développement de l'outil *VCC* de Cadence, et dans l'environnement CoWare [111].

Dans la suite, nous considérons les approches de partitionnement automatique en détaillant les différentes étapes listées plus haut.

1. Etape d'allocation

Contrairement à quelques définitions de la littérature, nous entendons par allocation l'étape qui consiste à trouver le meilleur ensemble de composants pour implémenter les fonctionnalités d'un système. Cependant, il faudrait choisir ces composants parmi des centaines de composants. A une extrême, on trouve des composants matériels très rapides, spécialisés, mais très coûteux comme les ASICs. A l'autre extrême, on a des composants logiciels flexibles, moins chers mais moins rapides comme les processeurs à usage général. Entre ces deux extrêmes, existent d'innombrables composants (dont les composants reconfigurables) qui offrent différents compromis en terme de coût, performance, flexibilité, consommation, taille et fiabilité par exemple.

L'étape d'allocation est l'une des étapes les plus déterminantes dans le processus de partitionnement. En effet, elle fixe et dimensionne très tôt dans le processus de conception l'architecture du système et en particulier ses performances maximum alors que des dysfonctionnements (non-respect des contraintes ou des débits) ne sont souvent repérés que très tard et après réalisation du système complet. C'est pour cela que dans l'industrie, ou le mot d'ordre est '*First Silicon Right*' (juste et du premier coup), les concepteurs, même chevronnés, tendent à sur-dimentionner l'architecture dès l'étape d'allocation en choisissant des processeurs plus rapides, des bus avec des débits plus importants ce qui fait qu'en général, on remet en cause très rarement les décisions prises à ce niveau de la conception.

Parmi les approches qui ont traité le problème d'allocation de façon automatique, on peut citer l'étude faite dans [108] qui, étant donné un ensemble de fonctions, détermine de façon automatique une allocation d'une architecture multiprocesseurs, qui satisfait des contraintes de temps et de coût. L'outil de partitionnment *CODEF* [109], élaboré au sein du laboratoire *I3S* en collaboration avec *Philips Semiconductors*, est un environnement qui permet d'explorer automatiquement et simultanément différentes allocations de processeurs, de DSPs et d'accélérateurs matériels, et surtout d'avoir des retours rapides de performances en terme de temps d'exécution et de taille de silicium.

2. Etape de partitionnement Spatial

Le problème du partitionnement se limitait souvent au partitionnement spatial ou à l'affectation des différents composants fonctionnels du modèle de l'application sur les composants de l'architecture allouée. Une formulation intéressante de ce problème est développée dans [2] et peut être résumée comme étant un problème de partitionnement de trois types d'objets de spécification sur les composants de l'architecture. Ces objets sont :

- les variables : Ces variables contiennent les valeurs des données et doivent être assignées à des composants mémoire.

- les «comportements» : Ces objets transforment les valeurs des données et doivent être assignées à des unités de traitement.

- les canaux : Ces objets transfèrent les données d'un comportement à un autre et doivent être assignées à des bus ou à des réseaux d'interconnexion plus évolués.

La plupart des études concentrent leurs efforts sur le partitionnement des comportements sur les unités de traitement de l'architecture en sous-entendant que le partitionnement des autres objets en découle naturellement.

Ce partitionnement nécessite d'abord de définir la granularité de ces comportements. Elle dépend de la granularité utilisée lors de la spécification et définit, comme on l'a vu dans le premier chapitre, le plus petit objet fonctionnel indivisible utilisé durant le partitionnement, comme les tâches, les processus, les boucles, les blocs de traitement, les opérations arithmétiques, les expressions booléennes. Les approches détaillées dans [113, 114, 115] considèrent une granularité assez fine au niveau opérations. D'autres comme [9, 116] considèrent une granularité au niveau boucles. Cependant, vu la complexité croissante des applications traitées, on tend naturellement à considérer des

granularités assez importantes afin de limiter le nombre d'objets ainsi que leurs interactions. La plupart des approches actuelles partent de spécifications sous forme de graphe de tâches de granularité importante comme dans [117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133]. Une étude interessante détaillée dans [134] permet de sélectionner la meilleure granularité en partant d'un code séquentiel en C ou VHDL et en utilisant une étape de *détermination de procédures* suivie par une étape de *pre-clustering.*

Ensuite, le partitionnement impose de déterminer la fonction de coût qui permet de mesurer la qualité d'une solution donnée et guider l'algorithme de partitionnement vers une meilleure solution. Cette fonction est souvent une combinaison pondérée de plusieurs métriques (souvent corrélées, parfois même antagonistes) comme le temps d'exécution [120, 9, 116, 123, 125, 126, 127, 128, 131, 132, 133], la consommation d'energie [119, 120], le coût de revient [119, 120, 137, 117, 118], la taille [120, 9, 123, 125, 126, 127, 128, 131, 132]. Cette fonction de coût doit aussi tenir compte des contraintes imposées au système ce qui rajoute souvent des termes correcteurs issus de différentes techniques comme :

- La minimisation de l'erreur quadratique moyenne, qui aide l'algorithme à converger vers des solutions satisfaisants les contraintes.

- La fonction Barrière, qui interdit l'exploration de solutions à l'extérieur de l'espace de conception autorisé.

- La fonction Pénalité, qui pénalise fortement l'exploration de solutions qui pourraient entrainer de forts dépassements des contraintes mais permet l'exploration des régions voisines des limites définies par ces contraintes.

Ces différentes formulations de la fonction de coût sont plus largement détaillées dans [135, 136].

Et enfin, il faut choisir un algorithme de partitionnement efficace pour explorer l'immense espace de solutions et en retenir une (ou plusieurs) qui vérifie(nt) les contraintes imposées au départ et optimise la fonction de coût. Ce problème d'affectation (comme le problème d'allocation et d'ordonnancement dans le cas multi-processeurs hétérogènes) est NP-difficile. Aussi, les approches optimales basées sur la programmation linéaire en nombres entiers [124, 128] ou l'exploration exhaustive [9, 138] ne peuvent être appliquées qu'à des problèmes assez réduits. Cependant, pour garantir des temps de calcul raisonnables, il est nécessaire de sacrifier la garantie d'optimalité et dans ce cas, des algorithmes d'améliorations itératives comme les algorithmes génétiques [125, 126, 119, 120, 121, 133] ou le recuit simulé [122, 123, 137] ont été utilisés avec succès pour résoudre le problème d'affectation. Une

méthode de programmation dynamique est utilisée dans [8]. Un algorithme d'allocation/ordonnancement (en général par liste) est considéré dans des travaux comme par exemple [113, 132]. Plus récemment, des approches par *clustering* sont utilisées, par exemple [40, 117, 118, 123] dans le but de privilégier le regroupement sur une même unité des fonctions ou tâches qui communiquent.

Dans la suite nous citons quelques approches plus spécifiques à notre problème qui ont traité le problème de partitionnement spatial en rappelant leur spécification de départ, la granularité de leurs objets, la fonction de coût considérée et l'algorithme de partitionnement utilisé.

L'approche détaillée dans [127] propose une méthode de partitionnement par migration de tâches entre le matériel et le logiciel suivie d'un algorithme d'ordonnancement par *list scheduling* modifié. Mais cette approche ne considère pas la faculté de reconfiguration partielle du FPGA ni le fait que, lors de la reconfiguration, le logiciel peut traiter d'autres tâches en parallèle.

L'étude faite dans [123] représente un bon état de l'art sur les approches de partitionnement spatial et détaille quatre types d'algorithmes dont trois classiques : le recuit simulé, Kernighan et Lin et un algorithme de *clustering* hiérarchique. Le quatrième algorithme est basé sur la notion de systèmes experts. Ces algortithmes ont été comparés par les auteurs sur la base des mêmes contraintes de temps, de surface et de mémoire et ceci en terme de qualité de solution et de temps de calcul.

3. Etape de partitionnement temporel

C'est une étape nouvelle dans le processus de partitionnement afin de tenir compte de la capacité de reconfiguration dynamique de certaines architectures. Comme la taille du composant matériel est souvent beaucoup plus petite que la somme des ressources nécessaires pour toutes les configurations, au lieu d'essayer de redimensionner le matériel ou de diminuer les tailles des configurations, on charge (et décharge) les configurations dans le (et du) composant lorsqu'on en a besoin. Cette technique permet le partage en temps de la même ressource materielle.

Cette étape consiste donc à grouper les tâches (ou fonctions de forte granularité), déja assignées à une implémentation matérielle sur le composant reconfigurable, au sein de contextes de configuration. Ces contextes vont être chargés puis déchargés de façon séquentielle sur le reconfigurable selon un ordre bien défini qui sera déterminé lors de la prochaine étape d'ordonnancement. Généralement, dans le cas d'architectures à reconfiguration partielle

(Virtex, AT40K, AT6000...), le temps de reconfiguration de ces contextes dépend linéairement de leur taille.

L'une des universités les plus actives dans le domaine du partitionnement et de l'exploration architecturale dans le cadre des architectures reconfigurables dynamiquement est l'Université de *Cincinnati* dans l'Ohio aux Etats Unis. L'une des premières études sur le partitionnement temporel a été réalisée à cette Université par Kaul et al. en 1998 [124] où ils ont proposé une linéarisation de ce problème pour le rendre sous forme ILP et ensuite le résoudre par une méthode de *branch and bound*. Dans [126], I. Ouaiss et al. ont proposé un flot de partitionnement (appelé *SPARCS*) avec une étape de partitionnement temporel résolue par programmation linéaire en nombres entiers suivie d'une étape de partitionnement spatial résolue par un algorithme génétique. Ces deux approches ciblent des architectures multi-FPGA. L'étude détaillée dans [127] cible une architecture formée d'un processeur connecté à une unité reconfigurable dynamiquement et traite le problème de partitionnement temporel conjointement avec celui de l'ordonnancement à l'aide d'un algorithme de *list-scheduling* modifié, mais cette approche ne considère pas la faculté de reconfiguration partielle de l'unité reconfigurable. En 2000, Ganesan et al. ont présenté dans [130] une méthode de partitionnement temporel qui gère l'exécution et le pipeline des reconfigurations sur une architecture reconfigurable partiellement (XC-6200) en partant d'une spécification en C/VHDL séquentielle traduite ensuite sous forme de CDFG.

Dans [113], les auteurs présentent une approche qui combine le partitionnement temporel et la réutilisation d'unités fonctionnelles pour minimiser le temps d'exécution total sous contrainte de surface maximale. Le problème est résolu par un algorithme de *list-scheduling* modifié. Cette approche part d'une spécification sous forme d'un graphe de flot de données et cible une architecture formée d'un processeur et d'une unité reconfigurable multiplexée en temps[1] (où le temps de reconfiguration est de l'ordre de quelques nanosecondes).

4. Etape d'ordonnancement

Cette étape a pour but d'ordonnancer les exécutions des tâches, les communications et les reconfigurations du FPGA et de vérifier que les choix de partitionnement et d'allocation respectent les contraintes imposées.

1. Time-multiplexed Reconfigurable Unit

Le problème d'ordonnancement multi-processeurs hétérogènes, même statique et hors ligne, est un problème NP-complet dont la complexité augmente encore en considérant des composants reconfigurables dynamiquement et extrêmement parallèles.

Dans [138], les auteurs présentent une approche d'ordonnancement des contextes de configuration sur l'architecture reconfigurable *MorphoSys* ainsi que des tâches au sein de ces contextes afin de minimiser le temps d'exécution global.

Le problème d'ordonnancement est souvent couplé au problème de partitionnement temporel puisqu'ils ont de nombreuses similitudes si on considère les dépendances temporelles inter-contextes similaires à celles inter-tâches. Un contexte B, qui dépend du contexte A, ne peut être chargé sur le reconfigurable avant que A ne soit chargé et exécuté [110].

Les approches détaillées dans [113, 127, 131, 133] considèrent une résolution par *list scheduling* modifié des problèmes de partitionnement temporel et d'ordonnancement.

Notons que l'une des premières approches qui a étudié le problème de partitionnement dans le cadre d'architectures reconfigurables dynamiquement et a proposé une méthodologie complète ciblant l'architecture XC-6200 est celle faite dans [141]. Elle part d'une spécification sous forme d'un CDFG et d'une solution candidate et par raffinements successifs arrive à générer les *bitstream* des différents contextes nécessaires ainsi que le contrôleur et l'ordonnanceur de configuration.

Par ailleurs, quand la séquence de tâches à traiter est imprévisible (et dépend de l'environnement ou de résultats de calculs intermédiaires), les décisions d'ordonnancement doivent être faites dynamiquement et en-ligne. Des approches comme [132, 141] traitent le problème d'ordonnancement dynamique multi-contextes en intégrant au sein de l'unité reconfigurable un ordonnanceur qui sélectionne le contexte à charger selon un mécanisme de priorité.

Des études récentes [75, 142, 143] ont abordé le cas où les décisions d'affectations des fonctions sur les unités, de même que l'ordonnancement, se font dynamiquement et en-ligne. Ceci pose les problèmes de migration de tâches, de sauvegarde de contexte, d'utilisation optimale des ports d'entrées/sorties, de communication, etc.

Ces études permettent d'adapter les traitements aux besoins de l'application en temps réel et d'optimiser ainsi l'utilisation des ressources mais elles rajoutent une complexité de calcul supplémentaire ainsi que de la surface silicium.

Conclusion

Il apparaît donc que le problème du partitionnement logiciel/matériel est approché de nombreuses façons suivant les modèles d'application et d'architecture considérés. Il ne semble pas envisageable actuellement de proposer une méthode de partitionnement générale, en particulier du fait de l'absence d'un modèle ou d'un environnement de modélisation de sémantique bien définie pour décrire les différents types de comportement d'une application (contrôle, traitements numériques et en temps continu par exemple) [22]. Aussi, pour obtenir une approche efficace, il est nécessaire d'identifier une classe d'applications et un modèle d'architecture adapté qu'il s'agit de paramétrer de manière efficace en fonction des caractéristiques de l'application cible, pour obtenir une solution optimisée par rapport aux critères choisis.

PARTIE 2

Méthodologie de partitionnement logiciel / matériel

Sommaire

Objet

La plupart des architectures reconfigurables présentées dans le chapitre 2 de la partie 1 sont des architectures innovantes et possèdent chacune des qualités intéressantes au niveau performances, reconfigurabilité et interconnexion entre les blocs de calcul. Ces architectures intègrent généralement un (ou plusieurs) processeur(s) et de la logique reprogrammable avec des granularités diverses. Ces architectures parallèles exploitent généralement un parallélisme à grain fin (de type ILP) et sont principalement dédiées à accélérer des fonctions de traitement. Nous pouvons citer l'exemple de l'architecture XPP de PACT. Les outils proposés pour 'programmer' ce composant sont principalement un compilateur/vectoriseur qui est capable d'obtenir à partir d'un code C, ou proche de C, les contrôles à destination des opérateurs et de l'interconnexion ainsi que la machine d'états qui séquence l'ensemble des contrôles.

Outre la flexibilité apportée par ces coprocesseurs reconfigurables, il apparaît que les outils associés facilitent le passage entre une description en C et leur programmation évitant ainsi une ré-écriture de la fonctionnalité dans un HDL en vue de synthèse. Cependant le choix des parties du code C à (dé)placer sur le coprocesseur reconfigurable est réalisée sur la base d'un *profiling* du code. Ce type d'outils vise à accélérer des traitements au sein d'une fonction ou d'une procédure suivant un mode co-processeur. Cette approche limite de fait l'exploitation du parallélisme potentiel entre fonctions pouvant être allouées sur le processeur et dans un même contexte de l'unité reconfigurable. Notre objectif est de considérer une approche de partitionnement qui opère au niveau fonction, niveau pour lequel peu de méthodes et d'outils sont disponibles.

Le problème du partitionnement peut se décrire comme la recherche d'une répartition des fonctionnalités d'une spécification sur une architecture cible, cette répartition étant déterminée de manière à garantir certaines performances et/ou optimiser un ou plusieurs critères.

Dans l'approche de partitionnement développée dans le projet *EPICURE* et détaillée dans ce manuscrit, l'objectif est de minimiser le temps global d'exécution de l'application en profitant au mieux des ressources disponibles, en particulier de la reconfiguration partielle de la partie matérielle et en exploitant la possibilité d'exécution concurrente des unités logicielle et matérielle. La granularité choisie est importante pour pouvoir profiter au mieux de l'accélération apportée par le coprocesseur reconfigurable. Cibler un temps

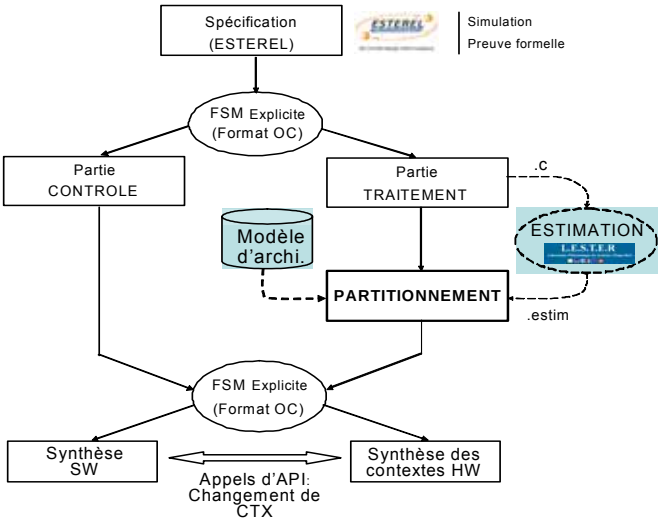


FIGURE 2.1. Le flot de partitionnement global

d'exécution minimum, éventuellement plus faible qu'une contrainte de temps imposée par l'application cible, permet par exemple de traiter des images de taille plus importante dans le cas d'une caméra intelligente ou de baisser la tension d'alimentation et la fréquence d'horloge pour réduire la consommation.

La méthode de partitionnement réalisée opère sur une représentation mixte flots de contrôle/flots de données. La partie contrôle est décrite à l'aide du formalisme *SyncCharts* (le formalisme graphique du langage *Esterel*) mis en œuvre dans l'outil Esterel Studio, et les traitements sont représentés par l'intermédiaire d'appels de procédures C.

L'ensemble de ce flot est repris plus en détail dans la suite de cette partie. Le premier chapitre détaille l'étape de modélisation sous Esterel Studio, le choix du langage, du format Automate (**oc**) ainsi que la construction des graphes de flots de données (GFD). Le second chapitre est consacré à l'étape de modélisation de l'architecture reconfigurable ciblée, en mettant l'accent sur les paramètres architecturaux considérés, les estimations en terme de temps d'exécution et de ressources des procédures écrites en C de l'application et le schéma d'exécution de l'application sur l'architecture.

Le dernier chapitre est consacré à l'étape clé de notre méthodologie : l'étape de partitionnement réalisée sur la base d'un algorithme génétique. L'exploration des solutions est effectuée par le processus de renouvellement des populations de l'algorithme génétique et l'évaluation de chaque individu est réalisée par un algorithme de *Clustering*. Cet algorithme effectue conjointement l'ordonnancement des fonctions ou tâches sur le processeur et sur le reconfigurable et construit les contextes de configuration.

Aborder le problème du partitionnement impose, comme on l'a vu dans la première partie de ce manuscrit, de modéliser l'application cible, l'architecture considérée et également de préciser le schéma d'exécution retenu de l'application sur l'architecture. Définir des modèles précis permet d'obtenir des comportements réalistes mais un excès de précision peut accroître fortement les temps de partitionnement. Des compromis sont à déterminer. Les chapitres suivants présentent les modèles retenus dans notre approche et détaillent notre méthode de partitionnement.

CHAPITRE 1

Choix du modèle d'application

Introduction

Le modèle d'application choisi est le modèle réactif synchrone à base du langage *Esterel*. Dans la suite nous allons argumenter ce choix puis effectuer une présentation générale de ce langage pour ensuite détailler notre modélisation à base d'*Esterel*.

1. Pourquoi Esterel ?

Les systèmes actuels contiennent de plus en plus de traitements de données mais surtout, la part de contrôle croît encore plus vite comme le montre la figure 2.2. Les *bugs* relatifs aux algorithmes de traitement sont généralement des bugs locaux donc relativement aisés à détecter et à corriger, alors que ceux relatifs à l'organisation système (systèmes dominés par le contrôle qui interagissent avec leurs environnement) sont par exemple des *bugs* introduits pendant la phase d'intégration avec une portée globale. Ils impliquent généralement le système complet et sont souvent découverts tardivement, ces *bugs* sont difficiles à détecter, difficiles à mettre en évidence par simulation et longs à corriger.

Depuis quelques années déjà, les industriels dans le domaine de la microélectronique ont identifié ce danger qui se traduit par des côuts de validation

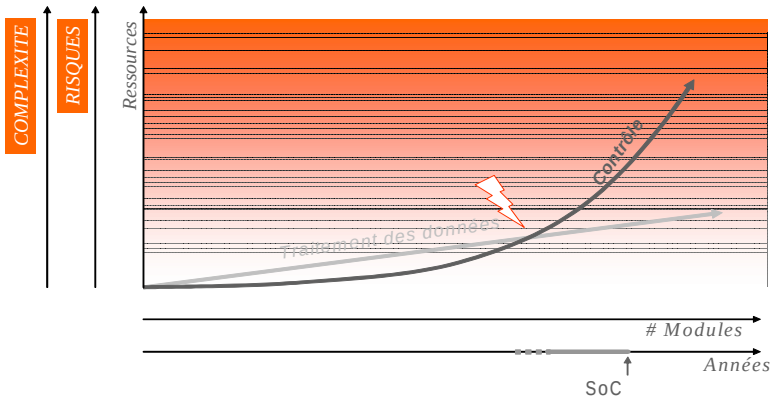


FIGURE 2.2. Les SoC de plus en plus dominés par le contrôle [149]

de plus en plus importants. Ils se sont orientés vers des langages de description de haut niveau qui permettraient de valider le système et de trouver et corriger plus tôt et plus facilement ces erreurs. Ils ont émis des besoins clairs en terme de nouveaux flots de développement pour des *SoC* de plus en plus complexes [149]. Ces besoins se traduisent principalement par :

- Un besoin en terme de spécification exécutable.

- Un besoin en terme de plate-forme de vérification formelle.

Cette analyse illustre l'importance de décrire précisemment la partie contrôle de l'application mais aussi les traitements induits par les changements d'états. En effet, les ressources qui contribuent principalement à la surface et à la consommation d'un *SoC* sont dues pour l'essentiel aux traitements, aux communications et aux données de l'application. Cependant, l'optimisation de ces ressources impose d'appréhender globalement les comportements de l'application, comportements qui sont la conséquence des évolutions des valeurs des contrôles. Par conséquent, il apparaît important de modéliser à la fois le contrôle et les traitements en mettant plus particulièrement l'accent sur le contrôle du fait des effets globaux indiqués précédemment.

Dans ce contexte, *Esterel* se distingue des autres puisqu'il permet de modéliser, à des niveaux d'abstractions variés, des sytèmes complets, complexes, dominés par le contrôle et d'en générer un modèle exécutable pouvant être simulé et formellement prouvé. *Esterel* possède aussi de fortes bases sémantiques lui permettant de synthétiser du matériel ainsi que du logiciel avec exactement le même comportement que celui décrit dans la spécification de départ[1].

1. Specification to implementation : correct by construction

Dans l'industrie, *Esterel* est utilisé dans plusieurs sociétés avec déjà quelques *success stories* [150]. Dans la recherche, *Esterel* est utilisé dans de nombreux travaux y compris dans des outils de co-conception logicielle/matérielle, par exemple l'outil *POLIS*, repris ensuite par Cadence dans le cadre de l'outil *VCC*.

Le choix d'*Esterel* comme langage de spécification en entrée de notre flot de co-conception n'est donc pas fortuit. Il a également été motivé par notre collaboration avec *Esterel Technologies* dans le cadre du projet RNTL *EPICURE*.

2. Le langage Esterel : Quelques notions de base

Esterel [35] est un langage réactif synchrone de haut niveau créé il y a plus d'une quinzaine d'années pour programmer des systèmes réactifs au niveau comportemental et au cycle près. Il intègre les notions de séquencement, d'émission/réception de signaux, de concurrence et de préemption. *Esterel* permet, comme on l'a vu dans le paragraphe précédent, de valider le système modélisé en donnant accès à la vérification formelle ainsi qu'à des techniques de génération de tests.

Le langage textuel original a été plus tard complété par un formalisme graphique sous forme d'une hiérarchie de machines à états finis concurrentes et communicantes appelé *SyncCharts* [145] (maintenant appelé *SSMs*[1]). Les deux formalismes, textuel et graphique, peuvent être librement mixés.

Esterel est généralement dédié à la programmation de systèmes matériels ou logiciels dominés par le contrôle. Ces systèmes sont réactifs, c'est à dire qu'ils réagissent continuellement à des stimuli issus de leur environnement en renvoyant d'autres stimuli. Ils sont aussi commandés par les entrées : ils réagissent à une cadence imposée par leur environnement. Une application réactive généralement manipule des données et du contrôle.

Esterel produit des signaux de sortie discrets en réaction à des signaux d'entrée. A un instant donné, un signal est caractérisé par son '*status*' de présence (une variable booléenne) et par une valeur typée. Un tel signal est appelé *signal valué*. Un signal ne portant pas d'autres informations à part son *status* de présence est appelé *signal pur*. La valeur d'un *signal valué* ne peut changer que lorsque le signal est présent. Cette valeur est persistante, c'est à

1. SSM : Safe State Machines

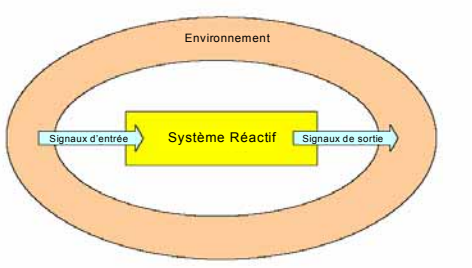


FIGURE 2.3. Interaction par signaux d'entrée/sortie [146]

dire que le signal garde la valeur qu'il avait à l'instant de sa dernière présence.

Dans une approche synchrone, les signaux sont les uniques acteurs permettant de modéliser les échanges d'information entre le système réactif et son environnement. Les signaux émis par l'environnement au système réactif sont appelés signaux d'entrée et les signaux engendrés par le système réactif sont appelé signaux de sortie. Les signaux d'entrée et de sortie définissent l'interface du système réactif (voir figure 2.3). Une *SSM*[1] décrit le comportement d'un système réactif, c'est à dire, la relation entre les signaux de sortie et les signaux d'entrée.

Les systèmes réactifs évoluent par des réactions successives se produisant à des instants dicrets, ce qui se traduit par un modèle d'évolution cyclique présenté par la figure 2.4. Une réaction se compose donc de trois phases :

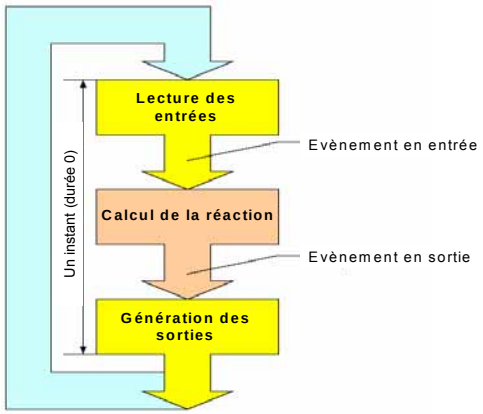


FIGURE 2.4. L'évolution cyclique [146]

1. Dans la suite, on appellera SSM le formalisme initialement appelé SyncCharts, conformément à la dénomination d'Esterel Technologies.

- Lecture des signaux d'entrée : Cette phase collecte les *status* de présence et les valeurs de chacun des signaux d'entrée.

- Calcul de la réaction : Cette phase calcule le prochain état interne de la *SSM* ainsi que les *status* de présence et les valeurs de chacun des signaux de sortie.

- Génération des sorties : Cette phase émet les signaux de sortie vers l'environnement. Lors de cette phase, on peut faire appel à des procédures C externes qui peuvent changer les valeurs de ces signaux de sortie.

Il est necessaire de vérifier à postériori que les hypothèses synchrones sont valides pour le système commandé :

durée de calcul < temps de réaction du système.

2. 1. Le formalisme SSM :

Dans la suite, nous ne considérons que le formalisme graphique d'*Esterel* détaillé dans [146]. Ce formalisme est assez proche de celui d'une machine d'états hiérarchique. Les notions d'*état* et de *transition* dans les *SSM* sont déduites de celles des machines d'états, mais au lieu de réagir à des évènements, le système réagit à des signaux.

Les *états* d'une SSM s'apparentent le plus aux états des FSM en rajoutant la notion de hiérarchie. Ces *états* peuvent encapsuler d'autres *états* et on parle alors de *macro-états*.

Les SSMs sont généralement formées de plusieurs machines d'états hiérarchiques. Ces machines ont des évolutions concurrentes, communiquent entre elles et se synchronisent à l'aide de signaux. Pour les distinguer des FSM classiques, nous allons utiliser, comme dans [146], le nom de graphes d'états et de transitions (STG[1]) pour désigner des graphes d'*états* connectés par des *transitions* étiquetées et possédant un état initial. Le *macro-état* ABRO de la figure 2.6 par exemple est un STG formé d'un état initial et du *macro-état* ABO.

A un instant donné de l'exécution d'une SSM, un STG donné possède un et un seul *état* actif. La notion d'*état interne* (ou de *configuration* au sens des Statecharts) d'une SSM peut donc être défini par l'ensemble de ces *états* actifs concurrents. Un des *état internes* possibles de l'exemple de la figure 2.6 est {ABO, WaitAandB, wA, dB}.

1. STG : State Transition Graph

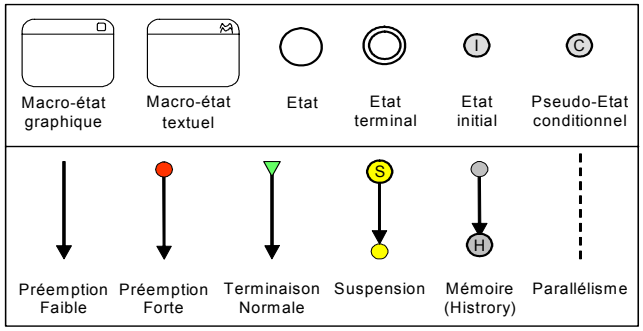


FIGURE 2.5. Eléments de syntaxe des SSM

Un *état interne* caractérise une condition qui peut persister pour une période de temps significative. Quand il est dans un *état interne,* le système est réactif à un ensemble de signaux et selon les valeurs et les *status* de présence de ces signaux, il peut atteindre (par franchissement de transitions) un autre *état interne*. Les *SSM* ont à la fois les possibilités des machines de Mealy (un signal peut être émis lors d'une transition) et des machines de Moore (un signal peut être émis en permanence tant qu'un état est actif).

Les *transitions* entre un *état* source et un *état* destination (que ce soit des *états* ou des *macro-états*) sont de trois types différents : les *transitions* à préemption forte, celles à préemption faible et les terminaisons normales (voir figure 2.5). La préemption (ou aussi avortement) forte interdit toute exécution de l'*état* préempté, dans l'instant de préemption. Il est représenté par un arc ayant comme origine un rond rouge. La préemption faible laisse au contraire terminer l'instant avant la destruction effective, l'*état* préempté peut, en fait, exprimer ses "dernières volontés". La terminaison normale permet de quitter instantanément un *macro-état* une fois qu'il ait atteint son état terminal.

Une *transition* peut être étiquetée par deux champs optionnels : un déclencheur et un effet (*trigger/effect*). Le déclencheur peut être un test (sur le *status* de présence, la valeur) d'un signal ou d'une combinaison de signaux utilisant les opérateurs *and, or, not* ou des fonctions plus complexes. Un effet peut être l'émission d'un ou de plusieurs signaux. Dans le cas d'une situation indéterministe où il existe plusieurs *transitions* sortantes d'un *état* donné, on affecte à chacune une priorité distincte correspondant à l'ordre de considération de leurs déclencheurs.

Une terminaison normale se distingue graphiquement par le triangle à son origine. Alors que les préemptions sont provoquées explicitement par des *triggers*, une terminaison normale est spontanée. Elle résulte du fait que chacune des composantes du *macro-état* correspondant est dans un *état* terminal (voir figure 2.6).

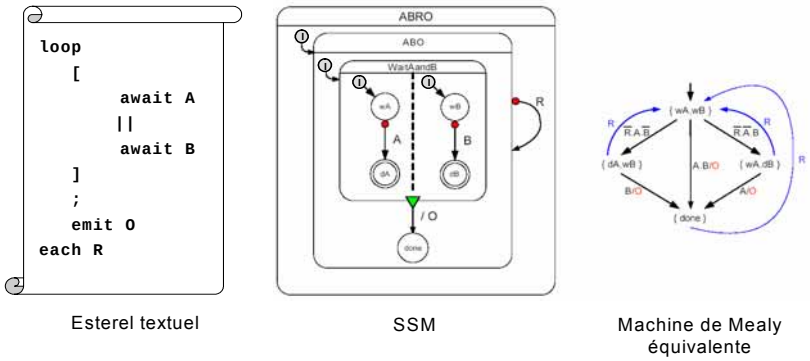


FIGURE 2.6. L'exemple ABRO

G. Berry propose dans son article d'introdution au langage Esterel [148] un exemple très simple, mais caractéristique des systémes réactifs, connu sous le nom *ABRO*.

Ce système a trois entrées A, B, R et une sortie O. Le signal O doit être émis dès qu'il y a eu occurrences de A et B, depuis l'instant initial ou depuis l'occurrence précédente de R. Le signal R est prioritaire, il remet le système dans son *état* initial (*Reset*). On trouve dans cet exemple un mélange de séquentialité, de parallélisme et de préemption :

- Contrôle séquentiel : O doit être émis après la réception de A et B.

- Contrôle parallèle : on attend les occurrences de A et B séparément.

- Préemption : l'occurrence de R fait avorter (immédiatement) les attentes de A et B.

La figure 2.6 donne la représentation de l'exemple *ABRO* en *Esterel* textuel, en *SSM* ainsi qu'en une machine de Mealy. Pour un exemple aussi simple que celui d'*ABRO*, la spécification en terme d'*Esterel* textuel semble la plus simple mais pour des systèmes plus complexes, la hiérarchie et la structure en modules des *SSM* simplifent fortement leur description.

De plus amples informations sur les *SSM* peuvent être trouvées dans [146, 148]. Dans la suite nous nous concentrons sur la compilation du code *Esterel* et la génération de code C et HDL.

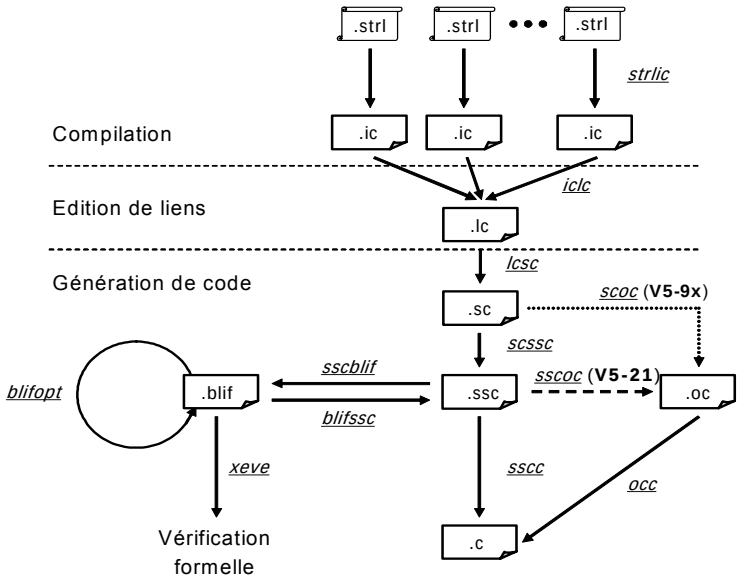


FIGURE 2.7. Une partie de la chaîne de compilation d'*Esterel* V5

2. 2. Compilation du code Esterel :

L'outil *Esterel Studio* intègre un compilateur qui traduit les programmes *Esterel* en code C pour la simulation ou pour l'implémentation logicielle ainsi qu'en Verilog ou VHDL pour la synthèse matérielle.

Le compilateur *Esterel* se base en général sur une représentation sous forme d'une machine de *Mealy.* Cette représentation peut être sous forme d'automate explicite (représentation explicite des états) ou sous forme de circuit (représentation implicite des états).

1. Compilation sous forme de circuit

Elle correspond à la mise en équation (*netList*) de la partie contrôle du programme. La partie données étant externalisée, l'activation des actions s'effectue au moyen d'*output* spécifiques. L'intérêt majeur de cette représentation réside dans le fait que sa taille est linéaire par rapport à la taille du programme source. Le format circuit de base est le format **sc** qui, une fois 'décyclisé' et trié, donne naissance au format **ssc** permettant la génération de circuits (voir figure 2.7).

2. Compilation sous forme d'automate

Le programme Esterel est dans ce cas compilé sous forme d'automate explicite. En partant du format **sc**, on explore l'espace des états atteignables. L'explicitation peut rendre exponentielle la taille de la représentation du pro-

gramme ce qui limite l'utilisation de ce format à des programmes de taille 'raisonnable'.

Par ailleurs, les avantages qu'offre ce schéma de compilation sont assez intéressants comme par exemple :

- Le fait qu'il permet de faire une vérification de propriétés plus aisée grâce à l'explicitation de l'espace des états atteignables.

- Le fait qu'il offre aussi une exécution efficace du code (le code généré est plus rapide) parce qu'on n'exécute qu'une partie du programme qui contient, explicitement, l'ensemble des actions à exécuter avant de se mettre dans l'état de destination (contrairement à un code généré après compilation circuit qui nécessite de passer en revue toutes les équations pour en extraire les informations sur ces actions et l'état de destination).

Pour ces raisons, nous avons opté pour l'utilisation du format intermédiaire **oc** dans notre flot avec la richesse d'informations qu'il offre car il permet de greffer des informations supplémentaires issues de la phase de partitionnement et de générer ensuite du code C efficace à l'aide de occ, comme illustré par la figure 2.7. Le format **oc** est décrit plus en détails dans le prochain paragraphe.

2. 3. Le format oc

C'est un format intermédiaire commun à *Lustre* et *Esterel*. Le format **oc**[1] est portable et permet la génération efficace de code C, ADA, etc. La structure du code **oc** est sous forme de modules.

Un module comporte typiquement :

- Un entête qui indique la version du compilateur utilisé (dans la figure 2.8, c'est la version V5 du compilateur).

- Une série de tables décrivants les objets auxquels l'automate peut faire référence.

- Un automate explicite

Dans la figure 2.8, nous illustrons une partie du code **oc** de l'exemple *ABRO*. Nous nous sommes intéréssés en particulier aux tables suivantes:

1. oc : object code

2. 3. 1. La table des types utilisateur

Elle contient les noms des types définis par l'utilisateur lors de la saisie de la spécification (en *SSM* ou *Esterel*). Ils sont énumérés et se distinguent des types *Esterel* prédéfinis (_boolean, _integer, _string, _float, _double) qui eux sont précédés du caractère $.

2. 3. 2. La table des procédures externes

Elle contient les noms des procédures C externes, les nombres et types de variables prises par référence et par valeur. Chaque procédure est déclarée de la manière suivante :

Nom (liste des types des arguments par référence) (liste des types des arguments par valeur).

Par exemple la procédure prédéfinie (précédée par un $) qui copie la valeur de la deuxième variable booléenne dans la première est donnée par :

$0: _assign__boolean ($0) ($0)

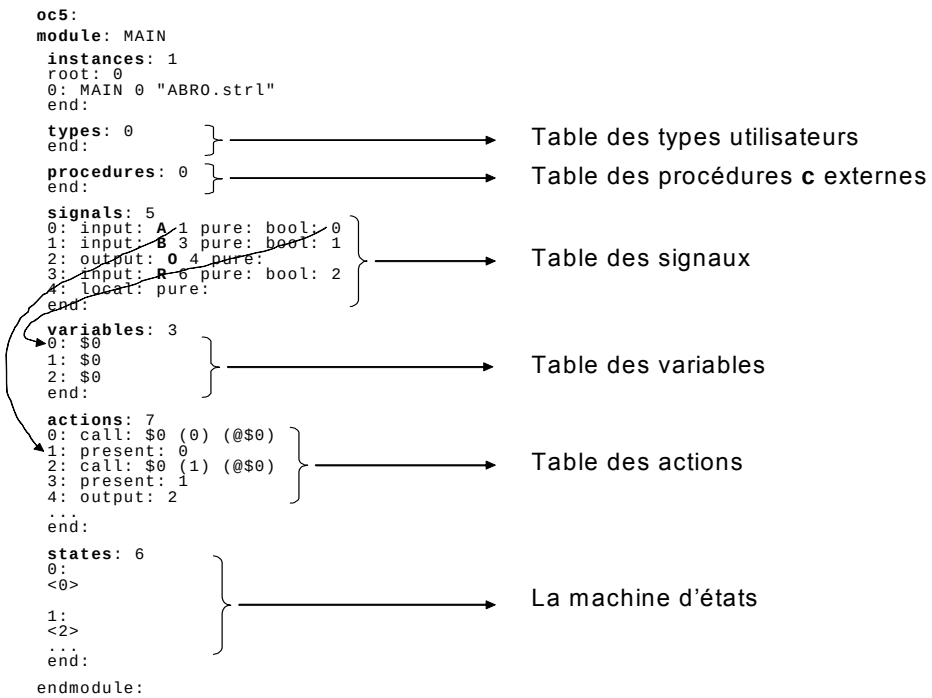


FIGURE 2.8. Extrait du fichier ABRO.oc

2. 3. 3. La table des signaux

Les signaux déclarés dans cette table ont la syntaxe suivante :

nature canal bool

- *nature* : un signal peut être soit visible, c'est à dire utilisé pour la transmission de valeurs, la synchronisation, l'accés à des paramètres externes de façon asynchrone, soit invisible, c'est à dire utilisé pour la communication interne et reste dans ce cas local. On distingue des signaux comme :

input : *nom present-action*

où *nom* est le nom du signal d'entrée, *present-action* est le numéro de l'action qui teste la présence de ce signal. Le signal A de la figure 2.8 est un signal **input**. Son action *present-action* est celle référencée avec le numéro 1 de la table des actions.

output : *name out-action*

où *nom* est le nom du signal de sortie, *out-action* est le numéro de l'action qui émet ce signal. Le signal O de la figure 2.8 est un signal **output**. Son *out-action* est l'action n° 4 de la table des actions.

sensor : *name*

où *nom* est le nom du capteur.

local :

pour les signaux locaux.

Cette liste n'est pas exhaustive et nous laissons le lecteur se reporter au document de référence [151].

- *canal* : Ce paramètre détermine la nature de la valeur véhiculée par le signal et peut donc être :

pure :

pour un signal pur ne véhiculant aucune valeur.

single : *var-index*

pour un signal valué qui ne peut être émis qu'une seule fois lors d'une réaction. L'index *var-index* pointe sur la variable contenant la valeur du signal.

multiple : *var-index comb-fun-index*

pour un signal valué qui peut être émis plusieurs fois lors d'une réaction. De même, *var-index* contient l'index de la variable contenant la valeur du signal et *comb-fun-index* est l'index de la fonction qui combine les valeurs des différentes émissions.

- *bool* : Ce paramètre renvoie l'index de la variable qui contient l'information sur le *status* de présence du signal considéré. Le paramètre *bool* du signal A

de la figure 2.8 est l'index 0, ce qui signifie que la variable n° 0 de la table des variables contient l'information sur le *status* de présence de ce signal.

2. 3. 4. La table des variables

Les variables déclarées dans cette table ont la syntaxe suivante :

type-index [value: *initial-value*]

Où *type-index* désigne le type de la variable. Le champs 'value:' n'est présent que lorsque la variable a été initialisée avec *initial-value*. La variable n° 0 de la table des variables de la figure 2.8 est par exemple de type $0, qui est le type booléen prédéfini d'*Esterel*.

2. 3. 5. La table des actions

Cette table spécifie toutes les actions élémentaires que l'automate peut exécuter durant une transition. Ces actions peuvent être de deux sortes :

- Des tests : Ce sont des actions suivies par deux chemins d'exécution possibles. Il existe trois types d'actions test :

present : *signal-index*

Cette action teste la présence d'un signal d'entrée et *signal-index* désigne l'index de ce signal.

if : *expression*

Cette action évalue l'expression booléenne *expression.*

dsz : *variable-index*

L'action 'decrement and skip on zero' décrémente la variable *variable-index* puis teste si elle est inférieure ou égale à zéro.

- Des *linear actions* : Ce sont des actions 'linéaires' qui n'affectent pas le flot de contrôle d'une transition. Les plus significatives (pour notre cas) sont les actions suivantes :

call : *procedure-index (variable-index-list) (expression-list)*

Cette action réalise les appels de procédures où *procedure-index* est l'index de la procédure dans le tableau des procedures, *variable-index-list* est la liste des arguments passés par référence et *expression-list* la liste des arguments passés par valeur. L'action n° 0 de la table des actions de la figure 2.8 est la suivante :

call: $0 (0) (@$0)

Cette procédure prédéfinie (déjà détaillée dans le paragraphe : table des procédures) copie la valeur @$0 dans la variable n°0 de la table des variables. La variable @$0 correspond à *false* car @ dénote une constante, $ une valeur prédéfinie et l'index 0 correspond à *false*. Ceci revient donc à mettre le *status* de présence du signal d'entrée A à *false*.

output : *signal-index*

Cette action diffuse un signal (ainsi que sa valeur) de nature **output**, ou **inputoutput** dans l'environnement.

2. 3. 6. L'automate

Un automate est décrit par les transitions entre ses différents états. Ces transitions consistent en une séquence d'actions. Toutes les actions qui se produisent pendant un *tick Esterel*, qui peuvent être parallèles, sont mises à plat dans cette représentation. Les actions test provoquent des branches binaires correspondants aux branches *then* et *else* du test. Ces branches peuvent se rejoindre plus tard ou rester indépendantes. A la fin de chaque série d'actions se trouve un état destination.

Chaque transition dans le format **oc** peut être représentée sous la forme d'un DAG[1] qui lui-même peut faire appel à d'autres DAG. Un DAG appelé sur plusieurs transitions ou plusieurs fois sur la même transition, permet d'avoir une représentation plus compacte de l'automate. Les DAG partagés sont groupés dans une table. Ceci n'est utilisé que dans la version optimisée du format.

Voici un exemple d'une table des DAG :

```
dags: 8
...
2: 24  <5>
3: 44 67 68 69 70 <4>
4: 9 5 3
...
end:
```

Le DAG 4 par exemple est un *open-dag* puisqu'il ne comporte pas d'état de destination (<state-index>) au niveau des feuilles, contrairement aux DAG 2 et 3 qui sont des *closed-dag*.

1. DAG: Directed Acyclic Graph

Prenons l'exemple de cet automate:

```
states: 42
startpoint: 1
sink: 0
calls: 234
0: <0>
1: 33  56  4  67  7  8  9  22  <2>
2: 0 (60  <3>)  ( )
   2 (33  34  35  36  <4>)  ( )
   6 (1  4  5 [2]) ( )
    <6>
...
end:
```

Où 42 est le nombre d'états de l'automate. L'état 0 est un état 'puits' (*sink state*) et l'automate démarre par l'état 1 (*startpoint*). *Calls* est le nombre de micro-étapes sommées sur toutes les transitions.

L'état n° 2 effectue d'abord le test d'indice 0. Si le test est vrai, il effectue la *linear action* n° 60 et l'état de destination est dans ce cas l'état <3>. Si le test est faux, la branche 'else' est exécutée (ici rien) et l'exécution se poursuit avec le test d'indice 2. Le test d'indice 6 contient une référence de type *closed dag* qui correspond au DAG n° 2 de la table des DAG de l'exemple précédent. On effectue dans ce cas, en plus des *linear actions* 1, 4 et 5, l'action 24 avant de se mettre à l'état <5>.

Pour plus de détails sur le format **oc**, le lecteur peut se reporter à [151]. Dans la suite, nous allons présenter l'utilisation faite de ce fomat intermédiaire au sein de notre flot de partitionnement.

*3. Analyse du fichier **oc** et génération des GFD*

Comme détaillé dans le paragraphe précédent, le format intermédiaire **oc** décrit la machine d'états explicite sous la forme d'une machine de *Mealy* qui, à partir de l'état courant, analyse les entrées pour calculer les actions (sorties) et déterminer l'état suivant. Ces actions opèrent sur des signaux d'*Esterel* et représentent des structures conditionnelles ou des appels de fonctions ou de procédures externes écrites en langage C.

Il est important de noter que dans l'approche développée, seules les procédures sont considérées car à l'inverse des fonctions, des effets de bord masqués ne peuvent pas être réalisés avec les procédures. Cette restriction simplifie l'analyse de dépendances nécessaire pour calculer pendant le partitionnement un ordonnancement des exécutions des traitements et des communications

FIGURE 2.9. Exemple de communication entre deux STG

associées. On pourrait aisément étendre la méthode de partitionnement au cas des fonctions en imposant des restrictions sur les fonctions elle-mêmes.

3. 1. Analyse des dépendances entre les procédures

Dans le formalisme **oc** les actions sur chaque transition sont fournies sous la forme d'une liste. Par exemple sur la figure 2.10, la liste d'actions 0 1 2 3 spécifiée sur la transition entre l'état courant <1> et l'état destination <2> correspond à quatre actions de type *call* (appels de procédures externes). A partir de cette liste d'appels et de la connaissance des paramètres d'entrée et de sortie de chacune de ces procédures, une analyse de dépendances est réalisée afin de construire un GFD qui représente le parallélisme de traitement exploitable.

Les procédures opèrent sur des variables *Esterel* locales au STG auquel elles appartiennent. Ces variables peuvent être communiquées entre procédures d'un même STG. Elles peuvent aussi être communiquées à d'autres STG et pour cela il faut impérativement passer par des signaux globaux. Dans ce cas, la valeur de la variable est véhiculée par un *signal valué* du même type qui sera émis et vu au même instant par tous les STG de la SSM.

Prenons l'exemple de la figure 2.9, les STG-1 et STG-2 formés respectivement des *états* (1, State1, exit1) et (2, State2, exit2) sont deux STG concurrents. L'occurrence du signal A déclenche la transition de l'*état* 1 vers l'*état* State 1. La variable entière x de STG-1 prend d'abord la valeur du signal valué A (x := ?A), ensuite cette variable est modifiée par référence par la procédure f (call f(x)()). Par contre, cette valeur n'est pas modifiée par la procédure g qui utilise x par valeur (call g()(x)).

Pour communiquer la dernière valeur de cette variable x à STG-2, on émet un signal B, de type entier, avec cette valeur (emit B(x)). Cette émission déclenche la transition de l'*état* 2 vers l'*état* State 2 du STG-2. La variable entière y récupère cette valeur (y := ?B) permettant à la procédure h de l'utiliser (call h(y)()). Toutes ces actions sont exécutées dans le même *tick Esterel* pour atteindre l'*état interne* {State 1, State 2}.

Dans une réaction synchrone, les signaux émis peuvent participer à la réaction en provoquant l'émission de nouveaux signaux (l'émission de A à provoqué l'émission de B dans l'exemple de la figure 2.9). Ces actions instantanées peuvent provoquer des cycles de dépendances qui amènent à des réactions incorrectes. Il est important de noter qu'Esterel Studio vérifie qu'aucun cycle de causalité n'existe dans la SSM, c'est à dire éviter que la conséquence ait une influence directe sur la cause. De ce fait, les GFD qu'on peut extraire sont de nature acyclique.

Dans l'exemple de la figure 2.10, d'après la table des actions du code **oc**, la procédure P_3 opère sur les variables 1 et 2 qui ont été préalablement utilisées par référence respectivement par P_1 et P_2. Il existe donc une dépendance de données entre P_1 et P_3 d'une part et entre P_2 et P_3 d'autre part. La procédure P_4 opère uniquement sur la variable 3 non utilisée par les autres procédures, ce qui implique que cette procédure peut s'exécuter en parallèle à toutes les autres.

Dans le cas des DAG, ces derniers sont dépliés récursivement afin d'obtenir la liste d'actions la plus longue possible. Le traitement des deux types de DAG (*open-dags*, et *closed-dags*) est strictement identique.

Chaque nœud d'un GFD représente une procédure (notée tâche dans la suite) de l'application qui peut être exécutée sur le processeur (*CPU*) ou l'unité reconfigurable (*UCR*). A chaque arc entre deux tâches on associe un entier représentant la taille des données (en octets) à échanger entre les deux tâches. Ce nombre d'octets est déterminé à partir de la connaissance de la taille des types des variables communiquées et définies dans le fichier **oc**.

En effet, lorsque nous effectuons le calcul de la taille des données à transférer entre deux tâches dépendantes, nous avons besoin de connaître la taille des types élémentaires (prédéfinis d'*Esterel*) ainsi que celle des types utilisateur. Un fichier standard contient la taille des cinq types prédéfinis d'*Esterel* (_boolean, _integer, _string, _float, _double), et si l'utilisateur dans sa spécification d'origine utilise des types autres que ceux là (ces types se retrouvent dans la table des types du fichier **oc**, comme les types **u_int** et **user** de la figure 2.10), il doit inclure leurs tailles dans le fichier spécifié (comme les deux dernières lignes du fichier 'taille des types' de la figure 2.10).

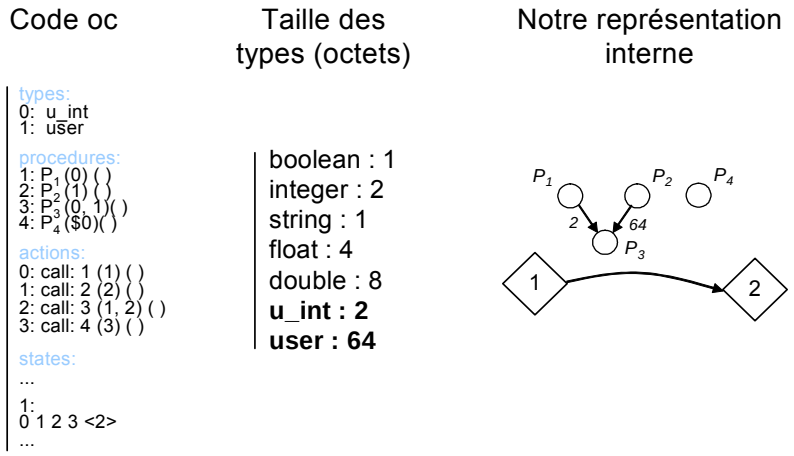


FIGURE 2.10. Traduction en une représentation interne

De manière classique, une tâche ne peut commencer son exécution que si toutes ses tâches prédécesseurs ont terminé leurs exécutions et si toutes les dépendances associées sont résolues. Les dépendances d'une tâche sont résolues si toutes les données nécessaires à son exécution sont disponibles vis à vis de l'unité qui exécute la tâche. Ces dépendances peuvent entraîner des communications de données entre les unités de l'architecture.

3. 2. Notre représentation interne de l'automate

On réfère cette représentation interne par le terme ***graphe de contrôle***. L'application est décrite ainsi par une structure mixte composée par la machine d'états explicite et un ensemble de graphes de flots de données, chacun associé à une transition de la machine d'états, représentant les traitements à effectuer.

Dans la suite de ce manuscrit, nous appellerons *transition* la relation reliant deux états du *graphe de contrôle* entre eux et *arc* la relation entre deux nœuds d'un GFD.

La figure 2.11 représente le cas d'une description **oc** avec une action de type test. L'action 3 est la conditionnelle qui, suivant sa valeur, conduit à l'état <2> ou à l'état <3>. Suivant la transition considérée, les appels de procédures à exécuter sont différents. Pour des raisons de commodité, un *nœud-test* (ici le *nœud-test* $<T_3>$) est inséré dans la représentation de notre *graphe de contrôle*[1].

1. Dans la suite, on appellera *transition-test* la transition qui part ou arrive à un *nœud-test*.

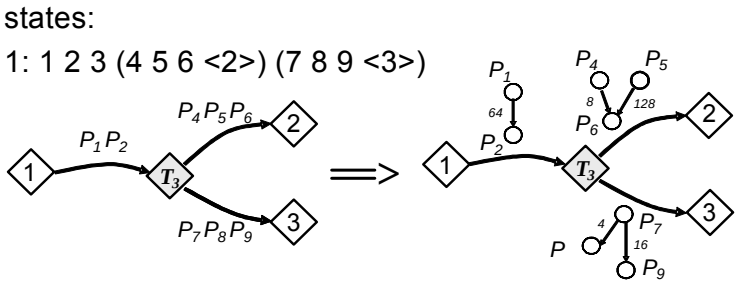


FIGURE 2.11. Cas d'un test lors d'une transition

Par ailleurs, les procédures activées sur une *transition-test* entrante d'un *nœud-test* (par exemple les procédures P_1 et P_2 de la transition entrante de $<T_3>$ sur la figure 2.12) peuvent poursuivre leurs exécutions au delà de la prise de décision relative au test, à condition qu'il n'y ait pas de dépendances entre les valeurs produites par les procédures et le test.

Afin d'éviter que ces tests n'imposent une fausse dépendance sur les exécutions des procédures, une autre étude de dépendance est réalisée entre les variables par référence des procédures de la *transition-test* entrante et celles testées par le *nœud-test*. Dans le cas où il n'existe pas de dépendance, ces procédures se retrouvent déplacées au niveau des deux *transition-test* sortantes comme illustré par la figure 2.12.

Ce déplacement d'appels de procédures sur les transition-test sortantes a pour conséquence de dupliquer des nœuds dans les GFD. Ce qui, dans la phase de partitionnement, nécessiterait plus de ressources matérielles en parallèle dans un même contexte par rapport à deux contextes séquentiels avant déplacement des procédures. Il présente cependant l'avantage de diminuer considérablement la latence d'état à état du fait de la suppression de la fausse dépendance de contrôle.

Cette approche de construction de GFD par transition peut néanmoins conduire après partitionnement à un faible taux d'utilisation des ressources de calcul (reconfigurable et processeur) si la description en SSM de l'application fait apparaître en moyenne peu de traitements entre deux *ticks* d'horloge.

Pour tenter de palier cette difficulté, il est possible d'augmenter la quantité de traitements pris en compte à chaque partitionnement en considérant une approche par chemins dans le *graphe de contrôle* et de procéder ainsi à une optimisation du temps d'exécution (au niveau partitionnement) plus globale que les optimisations locales faites sur les GFD des différentes transitions.

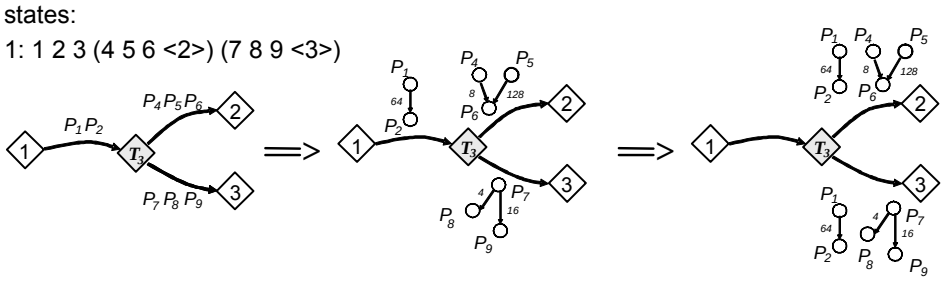


FIGURE 2.12. Déplacement d'appels de procédures sur des *transitions-test*

3. 3. Construction des chemins dans le graphe de contrôle

Un chemin dans le *graphe de contrôle* est constitué par un ensemble connexe d'états et de transitions tel que chaque transition ou état n'apparaît qu'une seule fois dans le chemin. L'origine du chemin est un état ou un *nœud-test*. La terminaison du chemin est également un état ou un *nœud-test*.

Une étape de *Profiling* de l'application est d'abord effectuée pour estimer les probabilités (ou fréquences) de passage par les différentes branches conditionnelles. Il est bien sûr nécessaire de considérer des séquences de test significatives pour effectuer ce *profiling*.

Dans notre approche nous considérons, qu'outre les états extrémités, il y a au plus un *nœud-test* dans un chemin. L'explication de cette limitation est donnée ci-dessous.

Un exemple de chemin est illustré dans la figure 2.13. Le choix du chemin C_1 = [<1> - <T_3> - <2> - <T_{13}>] a été guidé par l'ensemble des probabilités de passage par les branches calculées à la suite de la phase de *profiling*. Sur chaque chemin ainsi construit, les GFD relatifs aux différentes transitions qui forment ce chemin sont concaténés afin de construire un GFD global utilisé ensuite par le partitionnement. Cette concaténation doit tenir compte des points de synchronisation induits par les changements d'états et les tests. Pour cela nous avons introduit des arcs de dépendance 'fictifs' (sans communication de données), utilisés pour l'ordonnancement. Le GFD résultant du chemin C_1 de la figure 2.13 est une concaténation des trois GFD relatifs aux trois transitions qui constituent C_1.

La concaténation des GFD s'effectue de la manière suivante. Soient e_i, e_j deux transitions du *graphe de contrôle* telles que e_i est une transition entrante à l'état <E_k> et e_j est une transition sortante de cet état. On rappelle que les GFD sont par nature acycliques. Entre les nœuds terminaux du GFD associé à l'arc e_i et les nœuds initiaux du GFD associé à l'arc e_j on introduit des arcs

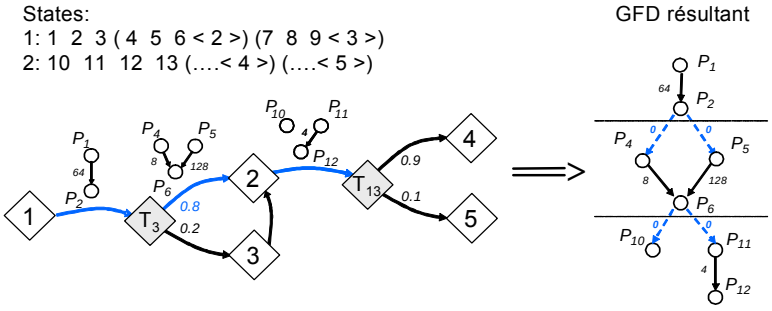


FIGURE 2.13. Exemple d'un chemin dans le graphe de contrôle et du GFD résultant

de dépendances afin de garantir que toutes les fins d'exécution des procédures sur la transition e_i aient lieu avant les débuts d'exécution des procédures sur la transition e_j. Il en est de même pour les transitions entrantes et sortantes des états-test du *graphe de contrôle*.

Cependant, le fait de considérer des chemins pendant le partitionnement impose qu'à l'exécution il soit nécessaire de s'assurer que le contexte correct est effectivement actif sur le reconfigurable lorsqu'un état du *graphe de contrôle* possède plusieurs transitions entrantes. Par exemple, lors de la transition de l'état <2> vers le *nœud-test* <T_{13}> de la figure 2.13, il faut s'assurer que la transition précédente était relative au chemin C_1 avant de transiter vers le *nœud-test* <T_{13}> du fait que les procédures sur cet arc ont été partitionnées suivant ce chemin. Si tel n'est pas le cas (c'était le chemin [<T_3> - <3> - <2>] qui a été utilisé pour transiter vers l'état <2>) il est nécessaire d'effectuer une reconfiguration pour mettre en place les procédures de la transition <2> -> <T_{13}> partitionnées suivant le chemin C_1. Cette vérification de chemin est facilitée par le fait que l'interface *ICURE* renvoie sur demande du processeur le numéro du contexte en cours sur le reconfigurable (voir le chapitre 2 de cette partie pour plus de détails). Par conséquent, au début de la transition <2> -> <T_{13}>, si le numéro du dernier contexte en cours sur le reconfigurable ne correspond pas à celui de l'arc <T_3> -> <2> alors une reconfiguration est nécessaire.

L'approche par chemin conduit à effectuer des spéculations sur les exécutions des procédures associées aux arcs qui composent le chemin. Par exemple, dans le cas du chemin C_1, si le *nœud-test* <T_3> aiguille le contrôle vers l'état <3>, la reconfiguration spéculée suivant le chemin C_1 devient inutilisée. Du fait que les temps de reconfiguration peuvent être coûteux, nous avons limité la longueur des chemins comme indiqué ci-dessus. Cependant l'approche décrite ici peut s'appliquer à des chemins de longueur quelconque.

Des expérimentations seraient nécessaires pour évaluer l'efficacité obtenue en fonction de valeurs de probabilité de transition affectées aux arcs du *graphe de contrôle*.

3. 4. Extraction des transitions (chemins) rédondants

Du fait de la nature rédondante d'un DAG, il serait par exemple intéressant, selon la granularité des procédures qui y sont appelées, de partitionner les DAG très récurrents et de les maintenir dans la mémoire de contexte (ou même dans le cache interne de contexte si l'UCR en dispose). Puisque nous n'avions pas eu accès à une version optimisée (générant les DAG) du compilateur *Esterel*, nous avons cherché à extraire les chemins (ou transitions) rédondant(e)s dans la machine d'état dans le but de réduire le nombre de GFD construits.

Par transition rédondante nous voulons dire toute transition, construite selon la procédure décrite dans le paragraphe 3.2, qui est entièrement identique à une transition déjà construite. La comparaison porte sur la suite d'actions de type :

- appels de procédure (**call**)
- génération de signaux de sortie (**output**)

Nous entendons par chemin rédondant, tout chemin construit selon la procédure détaillée dans le paragraphe 3.3 qui est entièrement identique à un chemin déjà construit. La comparaison porte sur l'ensemble des actions déjà testées pour comparer les transitions en tenant compte en plus des actions de type :

- tests sur des variables (**test, dsz**)
- tests sur des signaux (**present**)

Comparer les numéros d'actions nous permet de nous assurer qu'on compare les mêmes actions opérant sur les mêmes signaux et variables.

Les GFD rédondants sont ensuite éliminés après cette optimisation, ce qui permet de réduire significativement le nombre de GFD à partitionner.

Conclusion

Dans ce chapitre, nous avons présenté notre modèle de spécification basé sur le formalisme graphique SSM d'*Esterel*. Ce formalisme est ensuite traduit

par le compilateur *Esterel* sous forme d'un automate de Mealy explicite (format **oc**) que nous avons traité pour en extraire un *graphe de contrôle* comprenant sur chaque transition un graphe de flot de données.

Nous avons exposé une méthode pour augmenter le nombre de procédures pris en compte au niveau partitionnement en proposant une construction de GFD par chemins dans le *graphe de contrôle*. Nous avons aussi présenté une méthode de réduction du nombre de GFD générés pour le partitionnement en éliminant les GFD redondants.

Le prochain chapitre détaille le modèle d'architecture choisi pour ensuite présenter le schéma d'exécution retenu de l'application sur l'architecture.

CHAPITRE 2

Choix du modèle d'architecture reconfigurable et de son schéma d'exécution

Introduction

Le modèle d'architecture choisi est un modèle formé d'un processeur et d'une unité reconfigurable. Il est assez générique dans le sens où il prend en compte des unités reconfigurables avec différents niveaux de granularité (au niveau logique, au niveau fonction...), différents modes de couplage (on verra plus loin que le mode co-processeur est le plus ciblé par cette approche) et différents modes de reconfiguration (totale, partielle).

L'architecture étant définie, il s'agit de déterminer des modèles d'estimations des temps d'exécution, de reconfiguration et de communication.

Ces estimations sont nécessaires au partitionnement. En particulier, des estimations précises permettent d'obtenir des solutions optimisées après partitionnement, mais une grande précision peut accroître fortement le processus d'estimation. Des compromis sont en général considérés.

Une fois les modèles d'application et d'architecture fixés, il reste à définir le schéma d'exécution de l'application sur l'architecture afin d'obtenir un comportement à l'exécution correspondant à la spécification SSM traduite en machine d'états de type Mealy.

1. Le modèle d'architecture retenu

L'architecture ciblée dans notre approche est composée d'un processeur connecté à une unité de calcul reconfigurable (UCR) à travers une interface intelligente conçue par le CEA et appelée *ICURE*[1] [112]. Cette architecture est schématisée par la figure 2.14.

Le processeur embarqué considéré peut être d'un type quelconque à condition de permettre un couplage efficace avec l'UCR par l'intermédiaire d'*ICURE*. De plus, un processeur à comportement relativement déterministe sur lequel des temps d'exécution pire cas (WCET[2]) peuvent être déterminés avec précision permet d'optimiser l'ensemble, d'un point de vue ressources et consommation d'énérgie.

L'unité reconfigurable est une unité de granularité multiple possèdant des cellules logiques (*CL*) et des cellules dédiées (*CD*). Ce type de circuit s'est quelque peu généralisé ces dernières années et nous avons adapté notre modèle d'architecture pour tenir compte de ces évolutions. Les cellules logiques sont utilisées pour synthétiser tous les opérateurs de la description de la tâche. Les cellules logiques utilisées pour réaliser une tâche peuvent être reconfigurées pour réaliser une nouvelle tâche. Contrairement aux cellules logiques, les cellules dédiées réalisent une opération prédéfinie et ne sont pas reconfigurables. Les cellules dédiées permettent de réaliser efficacement certaines opérations, alors que leur réalisation par synthèse de cellules logiques est coûteuse en surface, en consommation d'énergie et en temps d'exécution. Comme exemples de cellules dédiées on peut citer les mémoires, les multiplieurs, des multiplieurs/accumulateurs.

1. Intelligent Control Unit for Reconfigurable Element
2. Worst Case Execution Time

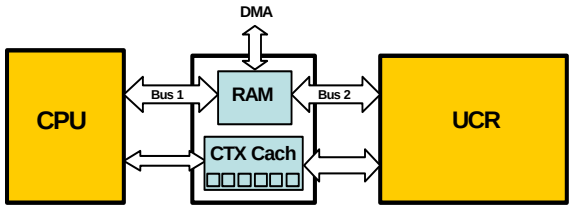


FIGURE 2.14. L'architecture cible

L'UCR peut être reconfiguré totalement ou partiellement à partir de l'interface *ICURE*. Cependant, nous avons considéré l'hypothèse que la reconfiguration partielle ne peut se faire en recouvrement avec des traitements exécutés sur l'UCR (pas de reconfiguration dynamique partielle au sens de la définition donnée au chapitre 2 de la partie 1). Traitements et reconfiguration sont donc exclusifs sur l'UCR. Les tâches sont groupées au sein de contextes pour diminuer leurs temps de communication. L'interface *ICURE* active la reconfiguration de ces contextes de manière séquentielle. Le schéma d'exécution est plus amplement détaillé à la fin de ce chapitre.

L'interface *ICURE* illustrée par la figure 2.15 comprend principalement une mémoire de données, un cache de contextes, une matrice de routage reconfigurable et un contrôleur de reconfiguration. Cette interface a pour rôle de faire communiquer de façon transparente le CPU et l'UCR et de gérer intelligemment les processus de reconfiguration. Le CPU est connecté à *ICURE* par un bus d'adresses/données standard. L'interface *ICURE* est connectée à l'UCR au moyen de ses ports d'E/S bidirectionnels et d'un lien de reconfiguration.

La reconfiguration de l'UCR est effectuée par une unité de gestion dédiée située à l'intérieur de l'interface comme illustré par la figure 2.15. Les données de reconfiguration sont stockées dans une structure appelée *contexte* qui encapsule le *bitsream*, les informations de routage et le code objet nécessaire pour la configuration et l'exécution des tâches affectées à l'UCR. Le CPU a accès aux fonctionnalités qu'offre *ICURE* par l'intermédiaire de fonctions API qui permettent une utilisation transparente des ressources du reconfigurable. L'appel à une fonction matérielle s'opére en deux étapes :

- Le CPU commence d'abord par demander le chargement du contexte correspondant à la fonction en appelant la fonction *load_context()*. Cette phase de chargement est entièrement et automatiquement effectuée par l'interface *ICURE*.

- Une fois la phase de chargement effectuée, le CPU peut entrer dans la phase d'exécution et utiliser la fonction matérielle désirée par un simple méchanisme d'appel logiciel.

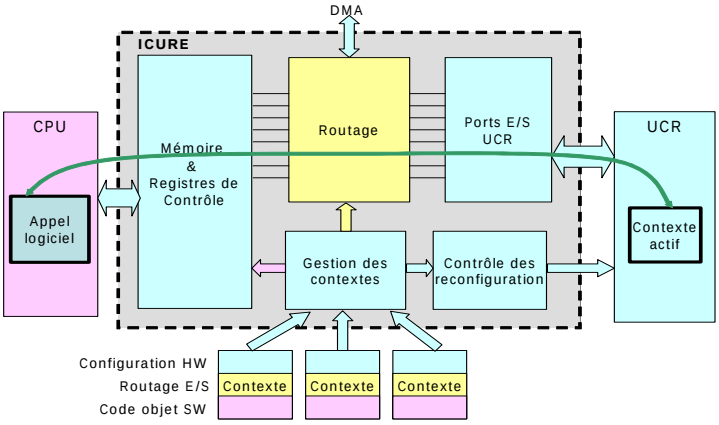


FIGURE 2.15. Détails de l'interface *ICURE*

La matrice de routage (*Xmap*) est un *crossbar* reconfigurable qui a pour rôle de faire correspondre les bancs mémoire aux ports d'E/S de l'UCR. La complexité de réalisation ainsi que le volume de ce crossbar augmentent avec le nombre de ports à connecter, ce qui pousse le CEA à considérer par exemple des structures à base de nano-tubes de carbone pour palier ces difficultés.

Le cache de contexte est une mémoire qui contient l'ensemble des contextes issus de la phase de partitionnement. Un contexte (figure 2.16) est une structure originale, conçue par le CEA, formée de :

- d'une partie binaire (*bitstream*) responsable de la configuration de l'UCR.

- d'une partie binaire pour configurer le *crossbar* afin d'assurer un bon routage des données à partir de (respectivement vers) la mémoire *ICURE* vers les (respectivement à partir des) ports d'E/S de l'UCR.

- d'un code objet SW pouvant effectuer la même fonctionnalité que le *bitstream* correspondant, dans le cas d'un mauvais fonctionnement de l'UCR.

2. Estimation des temps et des ressources à destination du partitionnement

Le modèle général d'architecture étant établi, il est nécessaire de déterminer les modèles d'estimation des temps de fonctionnement de l'architecture et des ressources matérielles requises de l'UCR pour exécuter les tâches de l'application. Ces valeurs sont utilisées par le partitionnement.

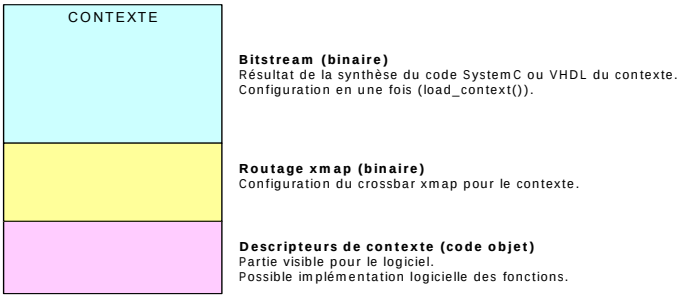


FIGURE 2.16. Anatomie d'un contexte

On distingue les temps de reconfiguration, de communication, d'exécution et les nombres de CL et de CD de l'UCR suivant la performance visée.

2. 1. Estimation des temps de reconfiguration de l'UCR

Il est important de tenir compte des temps de reconfiguration de l'UCR dans la phase de partitionnement puisque ces temps peuvent être très importants selon les composants ciblés.

Les constructeurs donnent généralement le temps de reconfiguration total du composant reconfigurable et parfois un temps de reconfiguration par cellule logique dans le cas d'une architecture reconfigurable partiellement. Si ce temps de reconfiguration par *CL* n'est pas disponible, il peut être estimé de plusieures manières dont celle utilisée dans notre approche. Nous partons de l'hypothèse que le temps de reconfiguration de l'UCR dépend linéairement du nombre N_k de cellules logiques *CL* utilisées pour réaliser le contexte *k* sur l'UCR. Soit T_F le temps que prend une reconfiguration totale, et $S_{max(CL)}$ la taille maximale de l'UCR en terme de Cellules Logiques, alors le temps de reconfiguration par *CL* peut être estimé à :

$$T_{reconf/CL} = \frac{T_F}{S_{max(CL)}} \qquad \textbf{(EQ 2.1)}$$

On évalue le temps de reconfiguration du contexte *k* par :

$$T_{reconf}(k) = N_k \cdot T_{reconf/CL} \qquad \textbf{(EQ 2.2)}$$

2. 2. Estimation des temps de communication

Le modèle de communication que nous avons établi essaye de tenir compte de la spécificité de l'architecture ciblée sans que sa prise en compte n'entraîne un accroissement trop important de la complexité de la méthode de partitionnement.

Dans notre modèle de communication, les transferts de données entre les deux unités s'effectuent au travers de la mémoire double-ports située dans l'interface *ICURE* connectée au processeur par son bus de données ($bus_{CPU\text{-}ICURE}$) et au reconfigurable par un bus spécifique ($bus_{ICURE\text{-}UCR}$). Ces transferts sont de type asynchrone, c'est à dire que les données sont d'abord écrites dans la mémoire d'*ICURE* puis lues depuis cette mémoire (ce qui nécessite une ressource mémoire assez importante).

On fait aussi l'hypothèse que les communications sur le $bus_{CPU\text{-}ICURE}$ sont bloquantes pour les traitements sur le CPU, c'est à dire qu'il n'y a pas de recouvrement entre traitement et communication sur le CPU. Par contre, celles sur le $bus_{ICURE\text{-}UCR}$ ne le sont pas pour les traitements sur l'UCR, c'est à dire que les traitements effectués sur une partie d'un contexte peuvent se faire en parallèle avec la communication effectuée par une autre partie du contexte. Détaillons maintenant le modèle de communication considéré.

Les caractéristiques prises en compte dans notre modèle sont les suivantes :

- débit et largeur des bus de données.
- temps d'accès à la mémoire de l'interface *ICURE*.

Le modèle de communication choisi permet d'avoir un temps de transfert dépendant linéairement du nombre de données à échanger [40]. Des modèles de communication plus riches peuvent être trouvé dans [152].

Soit ρ_i le nombre d'octets communiqués sur l'arc e_i d'un GFD et λ_l le nombre d'octets par paquet supporté par le bus *l*. Soient α_l le temps de communication d'un paquet sur le bus *l* et Ω_l le temps d'accès par paquet sur ce bus. Le temps d'accès mémoire (en lecture ou écriture) est représenté par Ω_{Mem} et la largeur du banc mémoire par λ_{Mem}. Le temps global pour communiquer les données représentées par l'arc e_i d'une unité vers une mémoire (ou vis versa) est donné par :

$$T_{com}(e_i) = \left\lceil \frac{\rho_i}{\lambda_l} \right\rceil \cdot (\alpha_l + \Omega_l) + \left\lceil \frac{\rho_i}{\lambda_{Mem}} \right\rceil \cdot \Omega_{Mem} \qquad \textbf{(EQ 2.3)}$$

Dans le cas d'une tâche τ_j exécutée par le processeur (respectivement l'UCR) dont au moins un de ses prédécesseurs τ_i est exécuté par l'UCR (resp. le processeur), et puisque notre choix a été porté sur un type de communication asynchrone, les données produites par τ_i à destination de τ_j doivent être copiées de la mémoire d'*ICURE* (resp. du processeur) dans la mémoire du processeur (resp. d'*ICURE*).

Les temps de transfert associés à cette communication sont évalués à l'aide de l'équation (2.4) et l'arc e_i entre ces deux tâches est libellé par :

$$T_{com}(e_i) = \left\lceil \frac{\rho_i}{\lambda_{CPU-ICURE}} \right\rceil \cdot (\alpha_{CPU-ICURE} + \Omega_{CPU-ICURE}) + \left\lceil \frac{\rho_i}{\lambda_{ICURE-UCR}} \right\rceil \cdot (\alpha_{ICURE-UCR} + \Omega_{ICURE-UCR}) + 2 \cdot \left\lceil \frac{\rho_i}{\lambda_{Mem}} \right\rceil \cdot \Omega_{Mem} \quad \textbf{(EQ 2.4)}$$

On convient que le temps de communication entre deux tâches affectées au CPU est nul. Dans le cas où deux tâches dépendantes sont exécutées par l'UCR et appartiennent à un même contexte (i.e. il n'y a pas de reconfiguration) aucun temps de communication n'est comptabilisé. En effet, ces temps sont du même ordre de grandeur que ceux relatifs aux transferts de données entre les entités logiques du circuit, synthétisées à partir de la description comportementale de la procédure.

Si les deux tâches τ_i et τ_j sont exécutées par le reconfigurable mais situées dans deux contextes différents, des temps de communication entre le reconfigurable et la mémoire d'*ICURE* sont considérés. En effet, les données calculées par la tâche productrice τ_i sur le reconfigurable doivent être sauvegardées dans la mémoire d'*ICURE* avant que la reconfiguration du contexte contenant la tâche consommatrice τ_j ne soit activée (ici, on considère le cas pire parce qu'il existe des architectures, notamment la famille *Virtex* de Xilinx, qui permettent de stocker les données intermédiaires dans des cellules dédiées de type block RAM tout en reconfigurant une autre partie du FPGA). Ensuite, les données sont transférées de la mémoire d'*ICURE* vers la tâche τ_j lorsque la reconfiguration est terminée.

Dans un premier temps, et étant donné que les contextes ne sont pas encore construits, nous libellons l'arc e'_i entre les deux tâches affectées à l'UCR τ_i et τ_j avec un temps de communication relatif au cas pire, c'est à dire égal à la somme d'un temps de transit sur le $bus_{ICURE\text{-}UCR}$, d'un temps d'accès en écri-

ture à la mémoire d'*ICURE*, d'un temps d'accès en lecture et d'un temps de transit sur le $bus_{ICURE\text{-}UCR}$ que l'on simplifie par :

$$T_{com}(e'_i) = 2 \cdot \left\lceil \frac{\rho_i}{\lambda_{ICURE-UCR}} \right\rceil \cdot (\alpha_{ICURE-UCR} + \Omega_{ICURE-UCR}) + 2 \cdot \left\lceil \frac{\rho_i}{\lambda_{Mem}} \right\rceil \cdot \Omega_{Mem} \quad \textbf{(EQ 2.5)}$$

3. Estimation des coûts en temps et en ressources de l'UCR

L'estimation des ressources matérielles (ou l'estimation architecturale) et du nombre de cycles associés a été réalisée, dans le cadre du projet *EPICURE*, au laboratoire *LESTER* de l'Université de Bretagne Sud en utilisant l'outil *Design Trotter* [20].

Il faut noter que cette estimation ne vise pas à déterminer les WCET des fonctions considérées. Le calcul du WCET (problème NP-complet) consiste à déterminer un temps d'exécution pire cas connaissant une architecture. Comme l'architecture n'est pas figée, on souhaite connaitre l'évolution du temps d'exécution en fonction du nombre de ressources allouées à la tâche. Par ailleurs, les architectures 'matérielles' sont plus déterministes que celle des processeurs car elles sont dédiées à l'exécution d'une fonction. Par conséquent, une fois qu'un nombre de ressources a été déterminé, il serait possible de calculer un WCET par des techniques classiques. Dans notre cas, nous avons axé notre travail sur le partitionnement et nous utilisons directement les résultats d'estimations, sans calculer les WCET des tâches. Cette approximation ne remet pas en cause les principes de partitionnement mis en œuvre.

Chaque procédure C est d'abord transformée en un graphe de flot de données et de contrôle hiérarchique (GFDCH) pour extraire le maximum de parallélisme entre traitements et transferts de données contenu dans le code C. Ces opérations élémentaires de traitement et de communication sont stockées dans les structures de niveau hiérarchique le plus bas qui sont les GFD. A un niveau plus haut on retrouve les GFDC qui sont construits lorsqu'une structure conditionnelle est rencontrée. Le GFDCH global peut contenir d'autres GFDCH ainsi que des GFDC comme illustré par la figure 2.17.

Sur la base de ce GFDCH et d'un modèle architectural abstrait de l'unité reconfigurable, des estimations sont calculées selon une approche ascendante (*bottom-up)*. Les GFD sont d'abord caractérisés et ordonnancés selon des

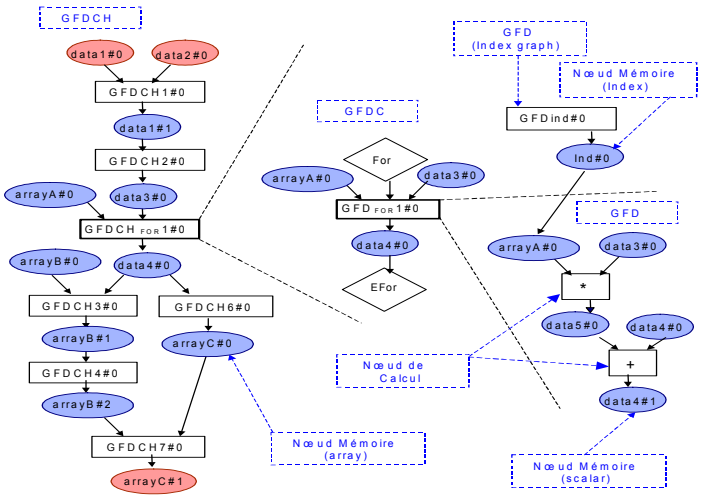


FIGURE 2.17. Exemple d'un GFDCH, CFDC et GFD selon la représentation interne de l'outil *DesignTrotter*

cycles abstraits. Ils sont ensuite combinés pour construire des GFDC et des GFDCH.

Un point important est que cette manière ascendante de combiner les GFD permet d'avoir, pour la même procédure considérée au départ, plusieurs estimations obtenues à des niveaux de granularité différents. Le concepteur possède alors des retours d'estimation rapides qui peuvent le guider dans le choix de la granularité au niveau de la spécification *Esterel* de départ.

Une courbe de points discrets correspondants à des compromis temps/ressources est calculée pour chaque GFD. Des techniques d'exploration architecturale qui intègrent par exemple des procédés de déroulage de boucles sont utilisées pour avoir plus de points d'estimation. Ensuite, selon que ces GFD et GFDC soient en parallèle, séquentiels, en exclusion mutuelle ou sous forme de boucle, leurs différentes courbes de compromis sont combinées hiérarchiquement pour donner à la fin la courbe du GFDCH global correspondant à la procédure C à estimer. Un exemple de courbe de compromis temps/ressources est donné par la figure 2.18. Les ressources matérielles peuvent être de plusieurs types différents (mémoire, blocs DSP, multiplieurs, UARTs, cellules logiques, etc).

Les estimations sont basées sur des composants synthétisés sur FPGA. Les délais sont obtenus en convertissant les cycles abstraits en cycles réels. Cette approche d'estimation peut être utilisée pour déduire des temps d'exécution sur cible logicielle (travail en cours au *LESTER*) en utilisant un modèle de l'architecture du processeur cible et en contraignant les ressources du FPGA afin de se limiter à ce modèle.

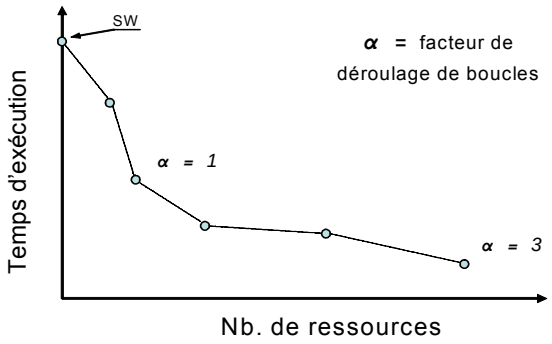


FIGURE 2.18. Exemple de courbe de compromis temps/ressources

Pour la partie logicielle, nous utilisons donc ces estimations de temps après modélisation du processeur cible ou (si nous avons accès à ce processeur) en exécutant la tâche par le processeur sur des séquences de test significatives pour essayer d'en déduire un temps d'exécution représentatif. Nous n'avons pas considéré ici de temps d'exécution pire cas sur le processeur du fait que sur le reconfigurable nous ne disposons que d'estimations des temps d'exécution. Les principes de partitionnement exposés dans la suite resteraient inchangés si des WCET des tâches étaient considérés en lieu et place de ces estimations matérielles et logicielles des temps d'exécution.

Du fait que le processeur fait partie de l'architecture, l'utiliser pour exécuter une tâche ne modifie pas le nombre de ressources disponibles dans l'UCR. Ainsi, les points relatifs à une implantation logicielle correspondent à une surface nulle et se trouvent donc sur l'axe des temps comme illustré par la figure 2.18.

Pour plus de détails sur cette étape d'estimation architecturale, nous invitons le lecteur à se reporter aux articles [20, 112].

4. Schéma d'exécution de l'architecture

L'objectif est d'effectuer le partitionnement de l'application décrite par la machine d'états et l'ensemble des graphes de flots de données associés, de telle sorte que le temps d'exécution global de l'application soit minimisé.

Pour assurer une cohérence au niveau de la localité des données entre GFD, nous avons fait le choix que chaque GFD récupère toutes ses données en entrée à partir de la mémoire d'*ICURE* et retourne en fin d'exécution les données qu'il a modifié à cette mémoire (figure 2.19).

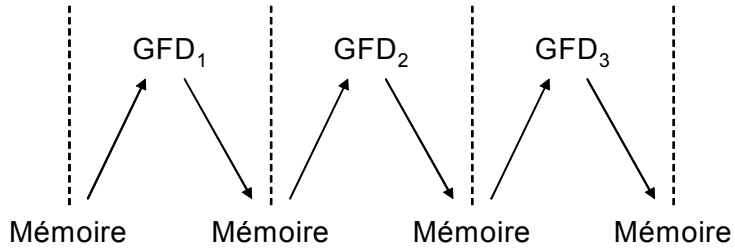


FIGURE 2.19. Choix sur la localité des données entre l'exécution de deux GFD successifs

De même, nous avons choisi d'allouer l'exécution de la machine d'états au processeur. Ceci résulte de la volonté de respecter la sémantique d'*Esterel* qui rend difficile le problème de la répartition de l'automate sur plusieurs unités asynchrones. Une autre possibilité eut été d'implémenter la machine d'états sur le reconfigurable. Cependant une fois le partitionnement effectué, la machine d'états modifiée aura pour charge le lancement des procédures affectées sur le processeur et la commande de l'interface *ICURE*, y compris les requêtes de reconfiguration. Sur ce dernier point il apparaît difficile de confier au reconfigurable la gestion de sa propre reconfiguration. Par conséquent, l'allocation de l'exécution de la machine d'états au processeur est l'approche la plus judicieuse.

Le schéma d'exécution qui en résulte consiste à associer au processeur le calcul de la fonction de transition d'états ce qui implique que le processeur :

- exécute les actions conditionnelles (l'action correspondante au *nœud-test* $<T_3>$ de l'exemple de la figure 2.13 par exemple).

- exécute les opérations sur les signaux et variables propres à *Esterel.*

- exécute ou gère les actions de type appels de procédures externes.

L'exécution de la machine d'états conduit implicitement à réaliser des points de synchronisation à chaque changement d'état du fait que les exécutions de toutes les actions listées sur une transition doivent être terminées avant le changement d'état effectif. Par conséquent, on peut considérer que le partitionnement peut s'effectuer de manière indépendante sur chaque GFD associé à un arc de la machine d'états, l'objectif étant alors de minimiser le temps d'exécution de ces GFD afin de réduire le temps d'exécution global de l'application. Dans ce cas, pour chaque GFD, le partitionnement construit un ensemble de contextes matériels (ensemble éventuellement vide si aucune procédure n'est affectée au reconfigurable) qui sont exécutés séquentiellement. C'est l'interface *ICURE* qui est chargée d'effectuer les reconfigurations et de lancer les exécutions des procédures suivant le séquencement

défini par le partitionnement et géré par le processeur sur la base de la spécification *Esterel*.

Une fois le partitionnement effectué, l'objectif est de préciser le schéma d'exécution sur l'architecture afin de mettre en œuvre effectivement le résultat d'ordonnancement de telle sorte que le comportement obtenu pour chaque GFD sur l'architecture après partitionnement respecte la spécification *Esterel* de départ.

L'outil ***OCRebuilder***, détaillé dans la partie 3, commence par changer les tables d'actions en regroupant les appels de procédures sur une transition par un seul appel à une procédure appelée *Schedule()* comme illustré par la figure 2.20. La table d'état se trouve elle aussi changée avec une seule action (de type *call*) par transition. Ensuite l'outil génère une table d'ordonnancement pour chaque transition à partir des informations fournies par le partitionnement qui contient la liste des procédures, leur affectation, leurs dépendances ainsi que leur état :

- procédure non encore exécutée
- exécution en cours
- exécution de la procédure terminée (état '*fin d'exécution*')

La table associée à la transition à effectuer dans le *graphe de contrôle* est passée en argument à la procédure *Schedule()* comme illustré par la figure 2.20

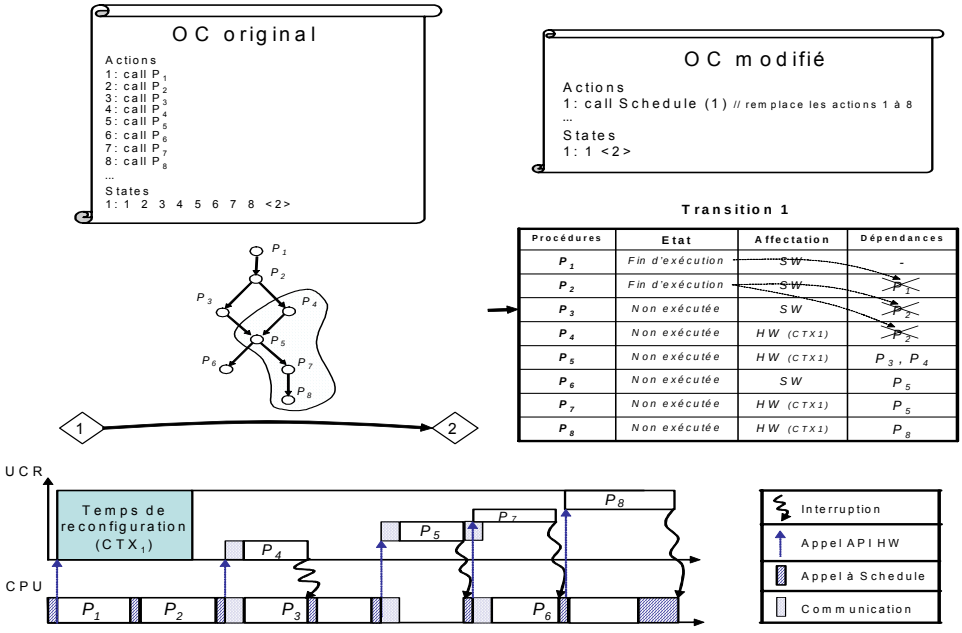


FIGURE 2.20. Schéma d'exécution considéré

pour la table de la transition n° 1 de l'état <1> à l'état <2>. Les appels des procédures P_1 à P_8 sont donc remplacés par l'appel de la procédure *Schedule(1).*

Une fois le fichier **oc** reconstruit et les tables d'ordonnancement générées par l'outil ***OCRebuilder***, on utilise l'outil ***occ*** de la chaîne de compilation qu'offre *Esterel* pour générer le code C qui sera exécuté par le processeur.

Le schéma d'exécution retenu est basé sur :

* Des interruptions émises par l'interface *ICURE* sur détection d'une fin d'exécution d'une procédure affectée au reconfigurable ou d'une fin de fonction *load_context()*.

* Des appels à *Schedule()* sur le processeur. Cette procédure est chargée du lancement des exécutions des procédures de l'application. Elle est activée à chaque interruption issue d'*ICURE* ou d'une fin d'exécution d'une procédure logicielle.

Il faut noter ici que l'algorithme d'ordonnancement utilisé dans notre approche et détaillé dans le chapitre suivant est basé sur l'hypothèse que les tâches ne sont pas préemptives, et que la notion de préemption (ou d'interruption) n'est introduite que pour s'assurer que le comportement à l'exécution respecte la spécification *Esterel* de départ.

Au début d'exécution d'une table d'ordonnancement, la procédure *Schedule()* est appelée sur le CPU et elle commence par rechercher le premier contexte à charger sur l'UCR et procède à un appel à l'API d'*ICURE load_context()* avec le numéro de contexte définis par le partitionnement pour que l'interface active la reconfiguration (c'est le cas du contexte CTX_1 de la figure 2.20). Cette procédure lance ensuite l'exécution des procédures dont les dépendances sont résolues (ou sans dépendances) et affectées à ce contexte. Une fois toutes les procédures matérielles ordonnançables lancées, la procédure *Schedule()* exécute la procédure affectée au processeur (P_1 est d'abord exécutée sur le CPU).

A la fin d'une procédure matérielle, l'interface envoie une interruption au CPU. Sur réception de cette interruption, ou après la fin de la procédure SW courante, la procédure *Schedule()* est appelée. Elle fait passer dans l'état '*fin d'exécution*' la procédure à l'origine de cet appel et l'efface du tableau des dépendances des autres procédures. Ensuite, elle recherche dans la table d'ordonnancement la ou les prochaines procédures, matérielles d'abord, puis logicielle à activer en fonction des dépendances (une fois la procédure P_2 exécutée, on a passé son état à '*fin d'exécution*' et on l'efface du tableau de

dépendances des procédures P_3 et P_4). Ce principe est répété tant que toutes les procédures listées dans la table d'ordonnancement ne sont pas dans l'état *'fin d'exécution*'.

Une fois le GFD exécuté, le processeur termine l'exécution de la transition vers l'état destination de l'automate *Esterel*. Ces principes sont illustrés sur la figure 2.20.

Conclusion

Dans ce chapitre nous avons présenté le modèle d'architecture que nous allons cibler tout au long de ce travail.

L'approche de partitionnement proposée prend en compte les reconfigurations partielle et totale de l'UCR. Cependant, nous avons considéré l'hypothèse que la reconfiguration partielle ne peut se faire en recouvrement avec des traitements exécutés sur l'UCR (pas de reconfiguration dynamique partielle à chaud). Traitements et reconfiguration sont donc exclusifs sur l'UCR.

Pour pouvoir tenir compte de la reconfiguration dynamique partielle à chaud (au sens de la définition donnée au chapitre 2 de la partie 1) dans un outil d'aide à la décision au niveau système, il faut intégrer plus de paramètres spécifiques à ce type d'architectures et notamment :

- l'utilisation spatiale des ressources à chaque instant.

- la localité des ports d'E/S libres.

- des estimations de coût en terme de temps et de ressources pour chaque macro de communication (cas de la famille *Virtex* par exemple [90]) ajoutée pour faire dialoguer deux contextes voisins.

On remarque que tous ces paramètres sont très dépendants d'outils travaillant à des niveaux d'abstraction assez bas (comme les outils de placement/routage ou des estimateurs plus précis). Ce n'était pas notre choix de départ, donc nous avons préféré cibler des composants reconfigurables partiellement ou totalement sans pour autant mettre de côté la possibilié de considérer cette reconfiguration dynamique à chaud.

Nous avons aussi exposé le schéma d'exécution de l'application sur l'architecture une fois le partitionnement réalisé. Ce schéma est simple et prend en compte les particularités de l'architecture reconfigurable. Il essaye avant tout de faire que le comportement obtenu pour chaque GFD sur l'architecture après partitionnement respecte la spécification *Esterel* de départ.

Dans le chapitre suivant, nous présentons plus en détails l'approche de partitionnement qui représente le cœur de ce travail.

CHAPITRE 3

Partitionnement logiciel/matériel basé sur un Algorithme Génétique

Introduction

Comme nous l'avons déjà précisé dans le chapitre 3 de la Partie 1, le problème du partitionnement logiciel/matériel dans le cadre des architectures reconfigurables (sans le problème d'allocation dans le sens choix des composants de l'architecture) comprend trois principales étapes : le partitionnement spatial (ou affectation), le partitionnement temporel et l'ordonnancement.

Dans ce chapitre nous présentons notre méthode pour résoudre les différentes étapes de ce problème. Cette méthode est basée sur un algorithme génétique (AG) couplé à une heuristique de Clustering. Dans notre cas, le critère retenu est le temps d'exécution global de l'application. D'autres critères peuvent être envisagés, par exemple la consommation d'énergie. Dans un premier temps, nous allons essayer de motiver notre choix d'utilisation d'un AG. Ensuite nous allons présenter quelques notions de base sur ces algorithmes pour exposer enfin notre méthode de partitionnement.

1. Pourquoi un AG ?

Nous avons modélisé notre problème de partitionnement en utilisant un AG. Les algorithmes génétiques, qui rentrent dans le cadre des Algorithmes Evolutifs (AE), ont été développés au départ par *John Holland* en 1975 [153], et ont été depuis utilisés avec succès pour résoudre plusieurs problèmes dans le domaine des systèmes VLSI comme le placement/routage, l'optimisation de code pour les DSP ou la génération de tests fonctionnels.

Ces algorithmes ont montré de très bonnes performances dans la résolution de problèmes sur lesquels peu d'informations sont disponibles ou pour lesquels on doit considérer de multiples critères d'optimisation (souvent antagonistes). C'est exactement le problème rencontré dans le cas de la conception des système embarqués mixtes flots de données, flot de contrôle où la complexité de ces systèmes fait qu'elle ne peut plus être gérée qu'à des niveaux d'abstractions plus élevés, d'où la nécessité de manipuler des modèles moins précis (donc moins d'informations). Les concepteurs de ce type de systèmes doivent souvent optimiser des critères différents comme le temps d'exécution, la surface, le débit des données, le coût, la consommation, etc. Les Algorithmes Evolutifs en général sont une alternative interessante par rapport aux autres approches d'optimisation puisqu'ils peuvent être adaptés rapidement à la taille du problème et intégrer de nouveaux critères en changeant seulement leurs fonctions de coût.

2. Notions de base sur les AGs

Certains principes de l'évolution ont été soulignés par *Darwin* sous la maxime «les plus adaptés survivent». Le terme d'adaptation correspond à l'aptitude d'un individu à vivre (et se reproduire) dans le milieu où il se trouve. C'est le fondement du calcul évolutif.

Mais au delà de la survie d'un individu se pose le problème de la survie de son espèce. Là encore les progrès de la biologie nous ont permis de comprendre comment une espèce évoluait au fil des générations : à la sélection Darwinienne s'ajoute un autre phénomène : l'évolution génétique. La génétique biologique est une science trop vaste pour que nous entrions dans ses détails, mais elle indique que le patrimoine génétique (i.e l'architecture matérielle d'un individu) évolue au cours du temps, menant à des représentants de l'espèce toujours plus adaptés à leur milieu.

S'inspirant de ce phénomène, *J. Holland* a imaginé au début des années 70 le principe des algorithmes génétiques. Tout comme l'ADN représente un

codage de la construction d'un individu chez les êtres vivants, dans les algorithmes génétiques les solutions possibles sont codées dans l'espace de recherche, et l'évolution des individus (uniquement représentés par leur génotype) dans un milieu artificiel mène à l'amélioration des performances de ces individus.

Ainsi, il devient possible d'optimiser tout problème pour lequel un codage des solutions et une fonction d'évaluation sont constructibles.

Un algorithme génétique se base sur une procédure itérative durant laquelle un ensemble de générations sont créées, une par itération. L'entière population évolue simultanément de façon à ce que la probabilité de convergence vers un minimum local soit réduite. La procédure d'évolution peut être représentée par le graphe ou par l'algorithme générique de la figure 2.21. Ainsi, les différentes étapes d'un AG sont :

2. 1. Etape de Codage

C'est un problème crucial, parce que l'algorithme n'opère pas directement sur les solutions elles mêmes mais sur des codages de ces solutions sous forme de chromosome. Chaque solution est codée par un génotype (séquence de gènes) approprié dans l'espace de recherche, généralement représenté par un vecteur valué binaire ou réel de taille fixe ou variable.

Généralement, l'algorithme part d'une population initiale constituée d'un ensemble de solutions générées aléatoirement. Certains algorithmes combinent des heuristiques déterministes spécifiques au problème considéré pour

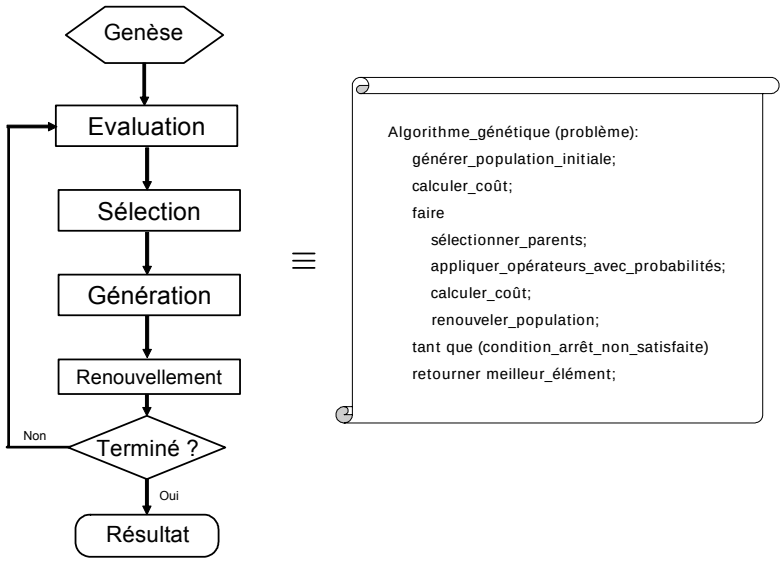


FIGURE 2.21. La procédure d'évolution classique d'un AG

générer une ’meilleure’ population initiale. Ces individus doivent ensuite être triés en fonction de leur coût issu de l’étape évaluation.

2. 2. Etape d'évaluation

Un autre problème crucial dans un AG est l’évaluation des solutions. L'algorithme évalue la performance de chaque individu (ou chromosome) à l'aide d'une fonction de coût. Le choix de cette fonction a une influence importante sur la performance générale de l’algorithme génétique.

2. 3. Etape de sélection

L'AG sélectionne les membres de la population courante (parents) qui vont être autorisés à se reproduire et créer ainsi les nouveaux membres de la prochaine génération.

Plusieurs stratégies de sélection ont été proposées comme :

- Sélection aléatoire : les parents sont sélectionnés aléatoirement sans considérer leur coût.
- Sélection par tournois : un nombre fixe d’éléments est sélectionné aléatoirement et les meilleurs individus (en terme de coût) joueront le rôle de parents dans la prochaine génération.
- Sélection à la roulette : on attribue à chaque individu un secteur de la roulette proportionnel à son coût et on procède ensuite à un tirage aléatoire pour sélectionner les parents.

2. 4. Etape de génération

Les individus sélectionnés (les parents) génèrent de nouveaux individus en utilisant des opérateurs génétiques de mutation et de croisement.

- La *mutation* est une reproduction homozygote pendant laquelle un parent unique crée un individu par une copie de son génotype où certains gènes sont modifiés. Cet opérateur assure essentiellement une tâche d'exploration des nouvelles zones de l'espace de recherche.
- Le *croisement* est une reproduction hétérozygote dans laquelle le génotype de l'individu créé dérive de ceux de ses parents. Le croisement assure une tâche d'exploitation des gènes courants du fait qu'il tend à combiner les caractères intéressants des deux parents.

2. 5. Etape de renouvellement

Plusieurs politiques de renouvellement de la population sont possibles, la politique élitiste est souvent pratiquée. Il s'agit de garder un ensemble formé des meilleurs individus (classés à la suite de l'étape d'évaluation) de la population courante (ensemble dit aussi *élite*) et de remplacer le reste de la population par la nouvelle génération.

Ce processus d'évolution continue jusqu'à ce qu'une condition d'arrêt spécifiée par l'utilisateur soit atteinte (une condition sur une solution est satisfaite, un nombre limite de générations est atteint…).

3. Notre méthode de partitionnement

Comme indiqué ci-dessus, nous avons déjà fait l'hypothèse que l'exécution de la machine d'états est effectuée par le processeur de l'architecture (voir Partie 2, chapitre 2). Le partitionnement porte donc sur l'ensemble des GFD associés aux transitions de la machine d'états, que ces transitions soient prises individuellement ou regroupées dans des chemins.

Les tâches des GFD peuvent être réparties sur les deux unités de l'architecture c'est à dire le processeur et le reconfigurable. Les tâches sur le reconfigurable sont regroupées dans des contextes (notion de partitionnement temporel). Un seul contexte est actif à la fois sur le reconfigurable. Par contre plusieurs tâches peuvent s'exécuter en parallèle au sein d'un contexte. Un objectif est aussi d'exploiter le parallélisme entre le processeur et le reconfigurable.

L'affectation et l'ordonnancement doivent considérer non seulement l'ensemble des tâches des GFD de l'application mais également les opérations rendues nécessaires par la mise en œuvre de ces tâches sur l'architecture. Ces opérations sont principalement les communications et la reconfiguration totale ou partielle.

Nous avons déjà passé en revue (au chapitre 3 de la partie 1) les différents problèmes que comprend le partitionnement logiciel/matériel ciblant une architecture reconfigurable (que sont les problèmes d'affectation, de partitionnement temporel et d'ordonnancement). La résolution de chacun de ces problèmes dans le cadre notre algorithme génétique est détaillée dans les paragraphes suivants.

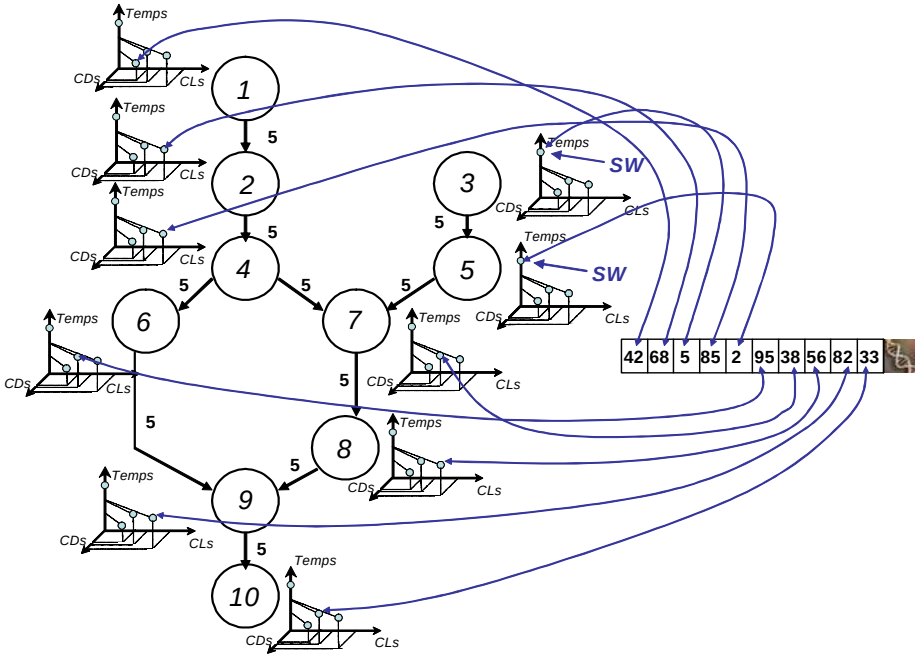


FIGURE 2.22. Codage d'un chromosome solution

3. 1. Problème d'affectation

D'abord, le problème d'affectation est traité dans le codage même des individus.

3. 1. 1. Codage des individus

A l'issue de la phase d'estimation (chapitre 2 de la partie 2), nous disposons pour chaque tâche d'un ensemble de points d'implantations dans le plan temps d'exécution/ressources. Le nombre de ces points est variable en fonction de l'exploitation du parallélisme disponible dans chacune des tâches.

Le codage de toute solution fournit une information sur la correspondance entre chaque tâche et l'un de ces points d'implantation comme illustré par la figure 2.22.

Soit *N* le nombre de tâches. Notre méthode de codage code un chromosome *C* avec un vecteur de gènes de longueur *N*. Chaque gène *C(i)* est un entier représentant un pourcentage. La valeur maximale 100% qu'un gène *C(i)* peut prendre est associée à l'implantation la plus coûteuse en terme de ressources de la tâche *i*.

Un problème se pose ici du fait que chaque implantation de tâche peut utiliser des nombres différents de cellules logiques (CL) et de cellules dédiées (CD).

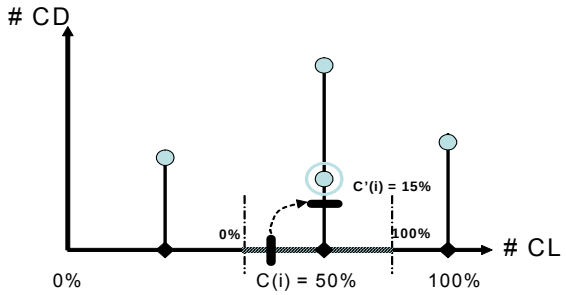


FIGURE 2.23. Sélection du point d'implantation

Avec un seul entier C(i) il s'agit néanmoins de différencier ces implantations. Ce pourcentage est d'abord pris sur l'axe des CL en sélectionnant le (ou les) point(s) le (les) plus proche(s) sur cet axe.

Si plusieurs points partagent le même nombre de CL, on les différencie par rapport à leur nombre de CD comme schématisé par la figure 2.23. Pour cela, on calcule un nouveau pourcentage de la façon suivante : on délimite la zone propre au point le plus proche de *C(i)* sur l'axe des CL (zone hachurée sur la figure 2.23) et on recalcule un *C'(i)* correspondant au nouvel emplacement de *C(i)* sur cette zone. Ensuite on transpose ce pourcentage *C'(i)* sur l'axe des CD et on sélectionne le point d'implantation le plus proche. Si plusieurs points partagent le même nombre de ces CD (cas où on considère plusieurs types de cellules dédiées), on les différencie de la même manière selon la procédure précédente. Donc, avec cette procédure, un gène correspond à un point d'implantation et un seul.

Toutes les solutions délivrées par ce codage sont ainsi viables. L'exemple du chromosome présenté par la figure 2.22 alloue les tâches 3 et 5 à une implantation *SW* et toutes les autres à l'UCR. Un autre avantage de cette méthode de codage est qu'elle rend aisée la mise en œuvre des opérateurs génétiques dans l'étape de génération de la nouvelle population. Les tâches affectées à l'UCR **doivent** être groupées au sein de contextes (ou *Clusters*) pour arriver finalement à évaluer les performances de l'individu.

3. 2. Problèmes de partitionnement temporel et d'ordonnancement

Notre approche effectue un ordonnancement statique, hors ligne, et non préemptif des tâches des GFD. Ce type d'ordonnancement permet de réduire le surcoût lié à la gestion des tâches et de déterminer aisément l'évolution du nombre de ressources nécessaires sur le reconfigurable pour exécuter les tâches qui y sont affectées. On peut ainsi obtenir des solutions qui respectent des contraintes sur les nombres maximum de cellules logiques et/ou dédiées présentes dans le reconfigurable.

Dans les approches classiques d'ordonnancement statique hors ligne de tâches, le début d'exécution d'une tâche est fixé une fois son ordonnancement réalisé. Dans notre cas il faut prendre en compte les temps nécessaires pour la reconfiguration lorsqu'un nouveau contexte est constitué. Le temps de reconfiguration d'un contexte dépend du nombre de cellules logiques obtenues par la synthèse des tâches qui composent ce contexte. Par conséquent, au moment du partitionnement, augmenter la taille d'un contexte suite à l'affectation d'une nouvelle tâche au reconfigurable augmente le temps de reconfiguration ce qui repousse d'autant le début des exécutions des tâches précédemment affectées au contexte. Cependant, il est possible de profiter du parallélisme existant entre la reconfiguration et l'exécution d'une tâche par le processeur. Par conséquent, les problèmes de l'ordonnancement et de la définition des contextes sont interdépendants.

L'objectif pour chaque GFD est de regrouper les tâches affectées à l'UCR au sein de *k* contextes tels que :

- la taille de chaque contexte ne dépasse pas la taille maximale de l'UCR.

- il existe une relation de précédence entre les *k* contextes.

L'exécution du GFD entier est réalisée en gérant l'exécution concurrente des tâches logicielles avec l'exécution séquentielle des *k* contextes matériels.

Les problèmes de partitionnement temporel et d'ordonnancement sont considérés conjointement dans notre approche vue la forte intéraction existante entre ces deux problèmes. Ils sont résolus dans le cadre de l'étape d'évaluation des individus de l'algorithme génétique.

3. 2. 1. Evaluation des individus

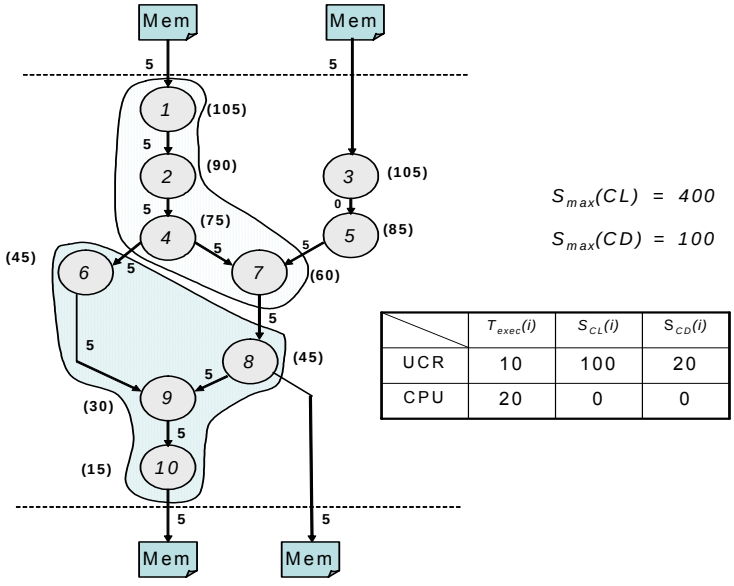
La performance de chaque individu (ou solution) délivré par l'AG est évaluée pour le classer à l'intérieur de la population courante. Notre approche consiste à évaluer une solution en terme de son temps global d'exécution y compris les temps de reconfiguration de l'UCR suite aux changements de contextes, les temps de communication entre les différentes tâches et les temps de communication avec la mémoire d'*ICURE*.

3. 2. 1. 1. Calcul des temps de communication préliminaires

Le codage même du chromosome fournit une allocation des tâches pour l'UCR et le CPU ce qui permet de calculer des temps de communication 'préliminaires' en utilisant les équations (2.3), (2.4) et (2.5) du chapitre 2 de la partie 2. Les arcs des GFD sont maintenant étiquetés par des temps de communication et non plus des tailles de données. Dans la figure 2.22 et pour des raisons de clareté, nous avons choisi de mettre tous les temps de communication à 5 unités de temps.

3. 2. 1. 2. Définition des contextes

L'objectif dans le projet EPICURE est de déterminer des solutions les plus rapides comme indiqué précédemment. Pour ce faire nous utilisons une approche inspirée de celle détaillée dans *COSYN* [40] pour grouper les tâches dans des contextes. Nous commençons par attribuer des priorités aux tâches en partant des feuilles du graphe et en remontant de proche en proche jusqu'aux racines du graphe. La priorité d'une tâche est la longueur du plus long chemin partant de cette tâche et arrivant à une feuille du graphe évaluée en terme de temps d'exécution et de communication (les priorités sont entre parenthèses sur la figure 2.24). Afin de réduire la longueur des chemins pour chaque tâche et de ce fait, celle du chemin critique, nous avons choisi de regrouper au sein d'un même contexte les tâches tout au long de ce chemin ce qui minimise (a priori) les temps de communication. La priorité P_i de la tâche i est calculée en considérant le temps d'exécution $T_{exec}(i)$ de la tâche i, les priorités de ses successeurs j, les temps de communication $T_{com}(e_j)$ sur l'arc



	$T_{exec}(i)$	$S_{CL}(i)$	$S_{CD}(i)$
UCR	10	100	20
CPU	20	0	0

FIGURE 2.24. Construction des *Clusters*

e_j entre *i* et *j* calculés et les temps de communication $T_{com}(i \rightarrow \text{mémoire})$ entre la tâche *i* et la mémoire selon l'équation (2.3) de la manière suivante :

$$P_i = T_{exec}(i) + SUP\,[\,SUP_{j \in successeurs(i)}\,(P_j + T_{com}(e_j))\,,\,T_{com}(i \rightarrow \text{mémoire})\,] \quad \textbf{(EQ 2.6)}$$

Les tailles $S_{CTX\_max(CL)}$ et $S_{CTX\_max(CD)}$ du contexte (ou *Cluster*) sont limitées par les tailles maximales $S_{max(CL)}$ et $S_{max(CD)}$ de l'UCR en terme de CL et de CD. En pratique, on limite la taille du contexte en terme de *CL* $S_{CTX\_max(CL)}$ à 80 ou 85 % de la taille totale $S_{max(CL)}$ par ce qu'au delà de cette limite, on rencontre plusieurs problèmes liés au placement et au routage des blocs formant le contexte.

Initialement, toutes les tâches sont triées par ordre de priorité décroissante. On considère la tâche *i* non 'clusterisée' caractérisée par ($T_{exec}(i)$, $S_{CL}(i)$, $S_{CD}(i)$) possédant la plus haute priorité, où $S_{CL}(i)$ (respectivement $S_{CD}(i)$) le nombre de CL (respectivement CD) correspondant au point d'implantation pointé par *C(i)* dans le chromosome. On marque cette tâche 'clusterisée'.

Les ressources disponibles $S_{CTXcurr}$ dans le contexte courant CTX_{curr} (initialement à S_{CTX_max}) sont décrémentées de S_i. Cette procédure de construction de contextes est itérée avec les tâches $j(T_{exec}(j), S_{CL}(j), S_{CD}(j))$ affectées à l'UCR tant que :

$$S_{CL}(j) \leq S_{CTXcurr(CL)} \quad \&\& \quad S_{CD}(j) \leq S_{CTXcurr(CD)} \quad \textbf{(EQ 2.7)}$$

Sinon, un nouveau contexte est créé et le processus est ainsi répété jusqu'à ce que toutes les tâches affectées à l'UCR soient attribuées à des contextes.

La figure 2.24 illustre la procédure de *clustering* dans le cas simplifié où les temps d'exécutions des tâches, le nombre de CL et de CD sont identiques et ne dépendent que de l'affectation. Les tâches 1 et 3 lisent des données en entrée à partir de la mémoire d'*ICURE* alors que les tâches 8 et 10 y stoquent des résultas en fin d'exécution du GFD.

3. 2. 1. 3. Mise à jour des temps de communication

Une fois les contextes définis, l'algorithme met à jour les temps de communication intra-Contexte (au sein du même contexte) et inter-Contextes (entre différents contextes). Les temps de communication intra-Contexte sont mis à zéro comme indiqué dans le chapitre 2 de cette partie. Soient $E_i(CTX_k)$ et $E_o(CTX_k)$ respectivement les arcs entrants et sortants du contexte CTX_k. Pour

chaque arc $e_j \in E_i(CTX_k) \cap E_o(CTX_l)$ des contextes CTX_k et CTX_l, le temps de communication est mis à jour de la façon suivante :

$$T_{com}(e_j) = T_{com}(e_j) + T_{reconf}(CTX_k) \quad \textbf{(EQ 2.8)}$$

où $T_{com}(e_j)$ est le temps de communication calculé en utilisant l'équation (2.3). Sur l'exemple de la figure 2.25 et après définition des contextes avec les tailles et temps de reconfiguration indiqués sur la figure 2.25, les temps de reconfiguration pour les deux contextes sont :

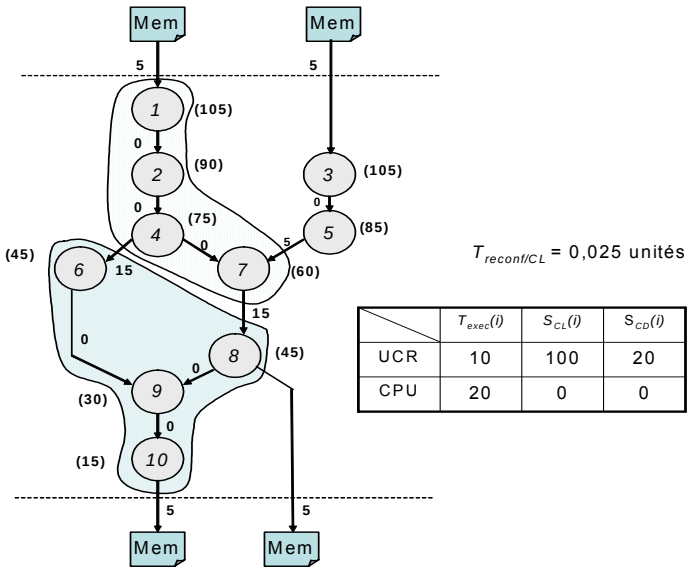
$T_{reconf}(CTX_1) = T_{reconf}(CTX_2) = 0.025 * 400 = 10$ unités.

Pour l'exemple de la figure 2.25, la mise à jour du temps de communication entre les tâches 7 et 8 donne :

$T'com(e_{7\text{-}8}) = 5 + 10 = 15$.

3. 2. 1. 4. Ordonnancement :

Comme indiqué précédemment, notre approche effectue un ordonnancement statique, hors ligne, et non préemptif des tâches des GFD. Cet ordonnancement est basé sur les priorités ASAP[1] calculées lors de la phase de *Clustering*.



	$T_{exec}(i)$	$S_{CL}(i)$	$S_{CD}(i)$
UCR	10	100	20
CPU	20	0	0

FIGURE 2.25. Mise à jour des temps de communication

1. ASAP : As Soon As Possible

Au terme de l'étape de *Clustering*, les contextes sont figés et les temps de reconfiguration deviennent donc fixes. L'algorithme d'ordonnancement commence par analyser les tâches de plus haute priorité en suivant les étapes suivantes :

- Ordonnancer leurs communications entrantes à partir de la mémoire d'*ICURE*.

- Ordonnancer leurs communications entrantes à partir des autres tâches.

- Ordonnancer leurs exécutions sur l'unité cible.

- Ordonnancer leurs communications sortantes vers les autres tâches.

- Ordonnancer leurs communications sortantes vers la mémoire d'*ICURE*.

Si la tâche est matérielle et que son contexte correspondant n'est pas configuré, on commence par ordonnancer la reconfiguration de ce contexte avant de procéder aux cinq étapes ci-dessus.

Les hypothèses qui ont été faites lors de cette étape d'ordonnancement découlent des caractéristiques de l'architecture détaillées dans le chapitre 2 de cette partie. Ces hypothèses sont les suivantes :

- Pas de recouvrement entre traitement et communication pour le processeur (pas de prise en compte de *DMA*[1]).

- Recouvrement possible entre traitement et communication pour l'UCR.

- Pas de recouvrement entre traitement (ou communication) et reconfiguration pour l'UCR c'est à dire que la reconfiguration partielle dynamique n'est pas prise en compte, et les données intermédiaires entre deux contextes successifs transitent par la mémoire d'*ICURE*.

Le temps de début au plus tôt d'une tâche (temps d'exécution ASAP) est calculé de la manière suivante :

- Si la tâche *i* est logicielle :

$$ASAP(i) = ASAP(\text{CPU}) + \sum t_{com}(\text{mémoire} \rightarrow i) \qquad \textbf{(EQ 2.9)}$$

où $ASAP(\text{CPU})$ est la date à laquelle le CPU est libre et $t_{com}(\text{mémoire} \rightarrow i)$ le temps de communication depuis la mémoire d'*ICURE*. Ce dernier temps regroupe les temps de communication des données provenant des prédécesseurs matériels de la tâche *i* ainsi que des données provenant d'autres GFD (cas de la tâche 3 de la figure 2.25).

1. DMA : Dynamic Memory Access

Ensuite on met à jour la valeur de *ASAP*(CPU) :

$$ASAP(\text{CPU}) = ASAP(i) + T_{exec}(i) + \sum t_{com}\,(i \rightarrow \text{mémoire}) \qquad \textbf{(EQ 2.10)}$$

t_{com} ($i \rightarrow$ mémoire) correspond au temps de communication des données vers la mémoire d'*ICURE* que ce soit à destination des successeurs matériels de la tâche *i* ou en tant que résultat de calcul final.

- Si la tâche *i* est matérielle :

$$ASAP(i) = \text{Sup}\,[début\text{–}ctx + \sum t_{com}(\text{mémoire} \rightarrow \text{i}),\ \text{Sup}_{(j \in prédéc_{HW}(i))}(ASAP(j) + T_{exe}(j))] \qquad \textbf{(EQ 2.11)}$$

où *début-ctx* est la date de début du contexte courant CTX_k et t_{com} (mémoire $\rightarrow$ *i*) le temps de communication depuis la mémoire d'*ICURE*. Ce dernier temps regroupe les temps de communication des données provenant des prédécesseurs logiciels (ou matériel appartenant à un contexte précédant) de la tâche *i* ainsi que des données provenant d'autres GFD.

Les prédécesseurs de la tâche *i* affectés à l'UCR sont notés $prédéc_{HW}(i)$. La tâche *i* ne peut commencer son exécution que lorsque toutes ses prédécesseurs en matériel finissent leurs exécutions.

Calcul de la date de fin *fin-ctx* du contexte CTX_k *:*

$$fin\text{-}ctx(CTX_k) = \text{Sup}_{(i \in CTX_k)}\,[ASAP(i) + T_{exec}(i) + \sum t_{com}\,(i \rightarrow \text{mémoire})] \qquad \textbf{(EQ 2.12)}$$

Calcul de la date *début-ctx* de début du contexte CTX_k *:*

$$début\text{-}ctx(CTX_k) = fin\text{-}ctx(CTX_{k-1}) + T_{reconf}(CTX_k) \qquad \textbf{(EQ 2.13)}$$

La reconfiguration se fait donc au plus tôt.

Pour l'exemple de la figure 2.26, on commence par ordonnancer la communication entrante vers la tâche 3, la tâche 3 sur le CPU, ensuite on ordonnance la reconfiguration du CTX_1 et par ordre de priorité, on ordonnance les tâches 1 et 2. La tâche 5 et ensuite ordonnancée, puis la communication sortante de la tâche 5, la tâche 4, la communication sortante de la tâche 4, la communication entrante vers la tâche 7, la tâche 7, sa communication sortante et ainsi de suite.

Cette procédure d'ordonnancement ne tient pas compte du traffic sur les bus ou de la taille mémoire encore disponible sur *ICURE*. Par exemple, les tâches 6 et 8 de la figure 2.26 ont la même priorité et demandent à accéder à leur

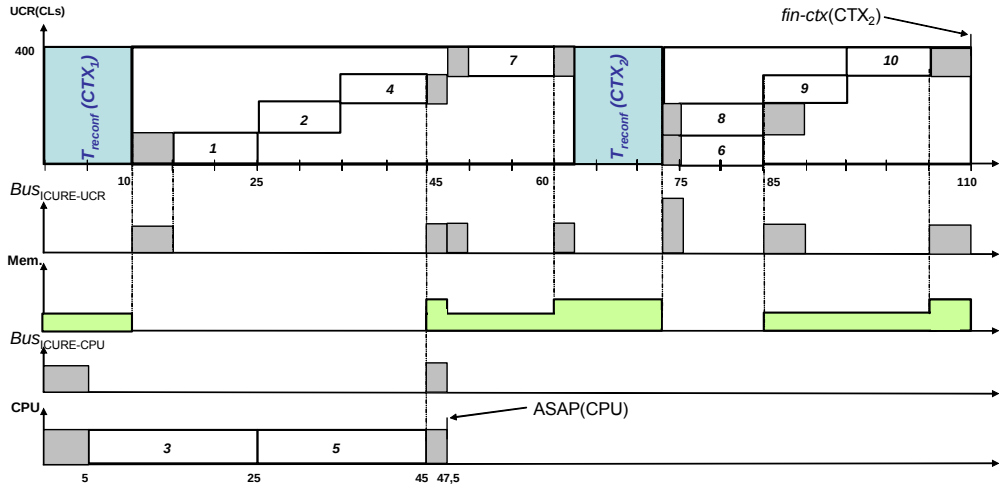


FIGURE 2.26. Ordonnancement et calcul du temps d'exécution total

données à partir de la mémoire d'*ICURE* au même instant. Une procédure de raffinement des communications est ici nécessaire pour ordonnancer ces communications en tenant compte du traffic sur le $bus_{ICURE\text{-}UCR}$.

Le coût en terme de temps d'exécution total de cette solution est donné par la formule suivante :

$$C_{Temps}(\text{Individu}) = \text{Sup}(ASAP(\text{CPU}), fin\text{–}ctx(CTX_{final})) \quad \textbf{(EQ 2.14)}$$

Pour l'exemple de la figure 2.26, le coût en temps de l'individu à évaluer est :

C_{Temps} (Individu) = *fin-ctx*(CTX_2) = 110 unités de temps.

Une fois les problèmes d'affectation, de partitionnement temporel et d'ordonnancement ainsi résolus, l'AG dispose à ce stade d'une population de solutions triées par rapport à notre fonction de coût (pouvant être multicritère comme présenté dans l'extension proposée au paragraphe suivant). Ensuite, l'AG procède à la sélection des parents qui vont être amenés à se reproduire.

3. 2. 2. Sélection des parents

La sélection des solutions par notre AG se fait par la technique de Tournois. On procède à $N_{parents}$ tournois, où chacun oppose un nombre fixé d'individus, aléatoirement choisis parmi la population courante. On sélectionne finalement le plus performant de chaque tournois pour faire partie des parents qui vont se reproduire à la génération suivante.

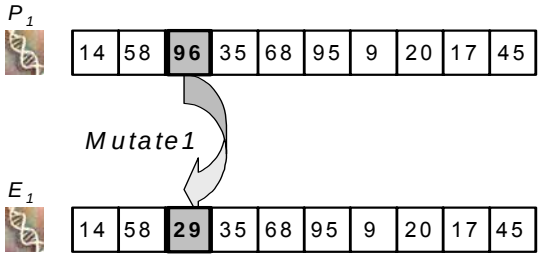


FIGURE 2.27. Exemple d'une mutation utilisant l'opérateur *Mutate1*

3. 2. 3. Génération

Plusieurs opérateurs génétiques sont appliqués aux $N_{parents}$ individus sélectionnés pour générer les $N_{enfants}$ solutions qui vont être intégrées dans la nouvelle génération.

Nous réalisons un contrôle dynamique du nombre d'individus créés par chaque opérateur (ou de la probabilité d'utilisation de tel ou tel opérateur) basé sur son efficacité à travers les générations précédentes. Les opérateurs génétiques utilisés dans notre AG sont :

3. 2. 3. 1. Les opérateurs de mutation

La mutation sélectionne aléatoirement un gène (ou un ensemble de gènes) et change sa valeur. L'affectation de la tâche correspondante peut changer d'une implantation SW à UCR, UCR à SW, ou la tâche peut rester sur l'UCR mais changer de point d'implantation. Dans notre algorithme, quatre opérateurs de mutation sont utilisés.

* *Mutate0* : qui change en *'0'* la valeur d'un gène aléatoirement choisi. Cet opérateur impose une implantation SW à la tâche sélectionnée.

* *Mutate1* (respectivement *Mutate2* et *Mutate3*) : change aléatoirement la valeur d'un (respectivement 2 et 3) gène(s) choisi(s) lui (eux) aussi de manière aléatoire.

La figure 2.27 illustre un exemple de mutation par l'opérateur *Mutate1* où le troisième gène (choisi de manière aléatoire) du parent P_1 mute pour donner naissance à l'enfant E_1.

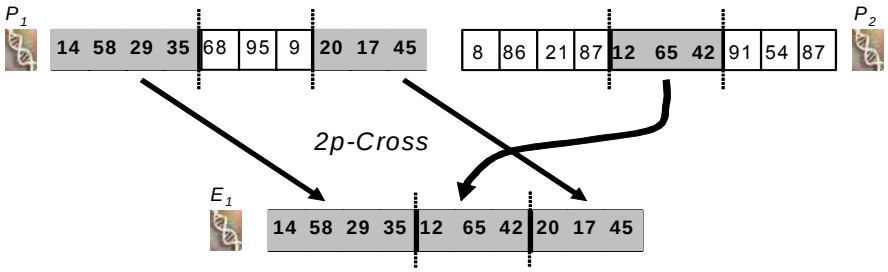


FIGURE 2.28. Exemple d'un croisement utilisant l'opérateur *2p-Cross*

3. 2. 3. 2. Les opérateurs de croisement

Deux sortes d'opérateurs de croisement sont utilisés dans notre algorithme : des opérateurs de croisement à partir d'un seul chromosome et des opérateurs de croisement à partir de deux chromosomes.

Les opérateurs de croisement à partir d'un seul chromosome comprennent les opérateurs suivants :

* *Scramble* : qui redistribue les gènes entre deux positions aléatoirement choisies dans le chromosome,

* *CutAndPaste* : qui coupe le chromosome en deux parties à un niveau aléatoirement choisi et les permute,

* *2-Opt* et *3-Opt* de la famille *k-Opt* : des opérateurs plus évolués qui font une recherche de voisinage.

Les opérateurs de croisement à partir de deux chromosomes sont des opérateurs qui coupent les deux chromosomes parents au même niveau (aléatoirement choisi) et les segments qui suivent la coupure sont interchangés. Dans

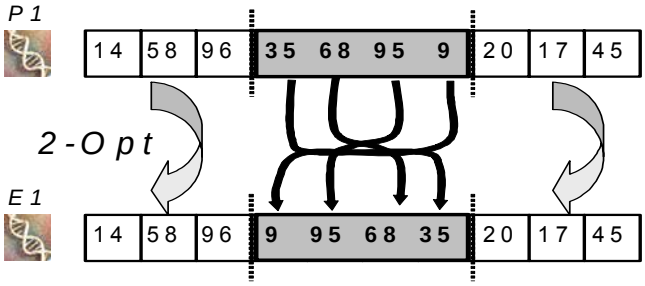


FIGURE 2.29. Exemple d'un croisement utilisant l'opérateur *2-Opt*

notre algorithme, un opérateur de croisement en un point (*1p-Cross*) et un autre à deux points (*2p-Cross*) sont utilisés.

La figure 2.28 illustre un exemple de croisement par l'opérateur *2p-Cross* qui découpe les chromosomes des deux parents P_1 et P_2 au niveau du $4^{ème}$ et du $7^{ème}$ gène (positions choisies de manière aléatoire) et les recombine de façon à produire l'enfant E_1. Un exemple de croisement à partir d'un seul chromosome et utilisant l'opérateur 2-Opt est illustré par la figure 2.29, où les segments des gènes compris entre deux positions choisies de manière aléatoire sont interchangées.

3. 2. 4. Renouvellement

Après la création de la nouvelle génération, le renouvellement de la population est effectué selon le principe élitiste. On combine donc les deux populations et on en garde les meilleurs individus. On appellera dans la suite cette procédure de renouvellement ELITISM_0. Une deuxième procédure de renouvellement a été évaluée dans le cadre de ce travail. Elle consiste à garder la totalité de la nouvelle génération et de ne garder que les meilleurs ($N_{parents}$ - $N_{enfants}$) parents en gardant une taille de population fixe. Cette procédure sera notée ELITISM_1 dans la suite. Les *Clones* ne sont pas autorisés dans ces deux procédure de renouvellement parce qu'ils peuvent envahir toute la population et produire ainsi une dérive génétique.

Lorsqu'un nombre de générations N_{gen} s'est écoulé sans aucune amélioration du meilleur individu, l'AG arrête son exécution et affiche la meilleure solution rencontrée.

Comme nous l'avons déjà mentionné, le partitionnement porte sur l'ensemble des GFD associés aux transitions de la machine d'états, que ces transitions soient prises individuellement ou regroupées dans des chemins. L'AG génère donc une solution unique pour chacun de ces GFD et le temps d'exécution total de l'application peut être calculé en sommant les temps d'exécution relatifs aux différents GFD se trouvant sur le chemin critique.

4. Limitations et comparaison

Parmis les limitations de notre méthode, on peut citer le fait que son principal objectif se limite à optimiser le temps d'exécution total de l'application sans tenir compte d'autres critères qui peuvent paraitre aussi importants comme la consommation d'énergie. La méthode actuelle aussi ne traite pas le cas con-

traint (cas des systèmes temps réel strict par exemple). Pour cela nous proposons un ensemble d'extensions qui sont actuellement à des stades plus ou moins avancés d'intégration.

4. 1. Extensions pour prendre en compte la consommation et les cas contraints :

D'autres critères d'optimisation ont été considérés dans un cadre prospectif, comme la consommation en terme d'energie et de puissance afin d'être intégrée dans la fonction de coût globale.

4. 1. 1. Calcul du coût en puissance :

A partir des résultats de partitionnement (affectation et ordonnancement sur l'architecture cible), nous avons utilisé un estimateur de consommation développé au sein de notre équipe qui indique l'énergie totale et l'ensemble des pics de puissance rencontrés lors de l'exécution.

La puissance d'exécution d'une tâche est estimée différemment selon qu'il s'agit d'une implantation logicielle ou matérielle.

* Pour les cibles logicielles, la puissance dissipée du circuit P dépend de la tension selon la formule classique :

$$P = \alpha \, . \, C \, . \, f \, . \, V^2 \qquad \textbf{(EQ 2.15)}$$

avec C : la capacité de charge et de décharge du circuit

f : la fréquence du circuit

V : la tension d'alimentation du circuit.

α : activité de commutation

Nous disposons pour le moment d'estimations de puissance pour les processeurs Palm [154], *OAK*, i8051, ARM 7, et TMS320C620 [156], que ce soit par nos travaux ou par ceux d'autres équipes (l'outil *SoftExplorer* du laboratoire *LESTER* par exemple).

On commence par construire une bibliothèque qui décrit les caractéristiques de chaque unité, comme sa puissance dissipée au repos P_{idle}, c'est-à-dire la puissance dissipée quand elle est en attente d'une tâche à exécuter.

Ceci est par exemple la description du DSP *OAK* de Philips [155] :

Processor: OAK

Total area: 3.4 mm2;

OnCore ROM size: 4 Kbyte;

OnCore RAM size: 8 Kbyte;

Clock cycle: 12.5 ns; // 80 MHz typ.

Pidle : 0.02 mW/MHz;

* Pour la cible matérielle, la puissance dissipée par l'UCR est estimée selon le modèle de consommation des FPGAs d'Altera donné par [157], puisque peu de modèles de consommation d'architectures reconfigurables sont disponibles à notre connaissance.

* Des estimations en terme de consommation mémoire faites au sein de notre équipe [158] peuvent aussi être considérées.

* Reste à considérer un modèle de consommation pour les supports de communications (bus, réseaux plus évolués...) qui restent peu étudiés jusqu'à maintenant.

Une fois que l'algorithme génétique a produit l'allocation et à l'ordonnancement des différentes tâches, les unités actives et au repos à chaque instant sont connues ainsi que le trafic sur les bus et les accès mémoire. Nous ciblons la recherche de la puissance maximale dissipée. Ainsi, quand plusieurs tâches ont des exécutions en recouvrement, il faut sommer leurs puissances respectives. Si dans ce même temps, il y a des unités en attente d'exécution de tâches, on ajoute la puissance que l'unité dissipe au repos : donc pour chaque date t, on ajoute les puissances des exécutions des tâches et les puissances de repos des unités au repos à cette même date ainsi que les coûts en consommation dus aux accès mémoires et aux interconnexions :

$$P_t = \sum_{\text{UnitesExec}} P_{exec} + \sum_{\text{UnitesRepos}} P_{repos} + \sum_{\text{com}} P_{com} \qquad \textbf{(EQ 2.16)}$$

Le coût en puissance $C_{Puissance}$ est la valeur maximale de P_t durant toute la période d'exécution.

4. 1. 2. Calcul du coût en énergie :

On estime l'énergie globale de l'application à partir des solutions d'allocation fournies par l'outil de partitionnement [163]. En repérant les instants de début et de fin des tâches, des communications et des accès mémoire, et en se

basant sur les calculs des P_t à chaque instant on parvient à calculer l'énergie dissipée pendant toute la période d'exécution en utilisant :

$$C_{Energie} = \int_t P_t \, dt \qquad \textbf{(EQ 2.17)}$$

Actuellement, l'outil de partitionnement produit des solutions indépendamment de l'outil d'estimation qui lui travaille en aval avec les résultats de l'AG sans qu'il y ait aucun bouclage afin d'optimiser la consommation. Une modification de la fonction de Clustering est nécessaire pour tenir compte de ce nouveau critère.

Une variante de l'algorithme génétique prenant en compte la consommation ainsi que des contraintes que ce soit au niveau de l'architecture ou de l'application elle même, est proposée dans [164] où on introduit des termes correcteurs de type erreur quadratique moyenne ou de type pénalité pour assurer la vérification de ces contraintes [135].

4. 1. 3. Contrainte temps réel :

Pour assurer les contraintes temps réel imposées par l'application, on doit assurer que le temps d'exécution maximal C_{Temps} calculé reste en dessous d'une contrainte de temps $Constraint_T$. Pour cela on utilise un terme de correction du type erreur quadratique moyenne Cor_T donné par l'équation suivante [135] :

$$Cor_T = \frac{(C_{Temps} - Constraint_T)^2}{Constraint_T^2} \qquad \textbf{(EQ 2.18)}$$

4. 1. 4. Contraintes sur l'énergie et la puissance dissipées :

Pour augmenter la durée de vie des batteries ou pour ajuster au mieux la batterie par rapport à la fonctionnalité recherchée, on doit s'assurer que pendant la période d'exécution, les pics de puissances recensés ainsi que l'énergie globale consommée ne dépassent pas les valeurs des contraintes $Constraint_P$ et $Constraint_E$.

Pour cela on utilise des termes de correction Cor_P et Cor_E de type pénalité. Le terme de correction correspondant à la contrainte sur les pics de puissance

(de même pour la contrainte sur l'énergie) est donnée par l'équation suivante [135] :

$$Cor_P = \left[\max \left\{ 0, \frac{C_{Puissance} - Constraint_P}{Constraint_P} \right\} \right]^2 \quad \textbf{(EQ 2.19)}$$

4. 1. 5. Calcul du coût global :

La fonction de coût générique est une somme pondérée des différentes fonctions de coût ainsi que des termes de correction. Cette fonction peut être exprimée de la façon suivante :

$$\hat{Coût}(\text{Individu}) = k_T \cdot \frac{C_{Temps}}{Constraint_T} + k_P \cdot \frac{C_{Puissance}}{Constraint_P} + k_E \cdot \frac{C_{Energie}}{Constraint_E} \quad \textbf{(EQ 2.20)}$$
$$+ k_{corrT} \cdot Cor_T + k_{corrP} \cdot Cor_P + k_{corrE} \cdot Cor_E$$

où les k_x sont des poids définis par l'utilisateur pour mettre plus ou moins l'accent sur les différents paramètres à optimiser.

Dans un premier temps, on pourrait prendre $k_T = k_{corrT}$, $k_P = k_{corrP}$, et $k_E = k_{corrE}$. Ces paramètres sont des pourcentages fixés par l'utilisateur comme détaillé dans le chapitre 1 de la partie 3.

4. 2. Comparaison avec d'autres méthodes de partitionnement

Dans ce paragraphe, nous allons essayer de comparer notre méthode de partitionnement avec trois méthodes rencontrées dans la littérature qui à notre sens sont les plus intéressantes :

- L'approche MOGAC [119] : cette approche part d'une spécification sous forme d'un graphe de tâches de périodes multiples. Elle cible une architecture multi-processeurs, multi-ASIC et traite le problème de partitionnement logiciel/matériel en utilisant un algorithme génétique multiobjectif (sa fonction de coût minimise le coût du circuit et la consommation d'énergie sous une contrainte temps réel stricte). La formulation du problème est intéressante dans le sens ou les auteurs traitent les problèmes d'allocation, d'affectation (partitionnement spatial) et d'ordonnancement. Le problème de partitionnement temporel n'est pas abordé car l'architecture ciblée n'est pas reconfigurable dynamiquement.

- L'approche détaillée dans [133] : cette approche part d'une spécification sous forme d'un graphe de tâches périodiques. Elle cible une architecture

formée d'un processeur connecté à un composant à reconfiguration dynamique partielle à chaud (le XC6264). Cette étude traite le problème de partitionnement de la spécification à l'aide d'un algorithme génétique de la manière suivante :

* l'affectation (le partitionnement spatial) des tâches sur les unités se fait, comme dans notre approche, implicitement dans le cadre de la procédure de codage des individus. Ce codage est binaire c'est à dire que chaque unité possède un seul point d'implantation.

* pas de partitionnement temporel puisque l'étude cible une architecture reconfigurable dynamiquement à chaud. Ce problème se transforme en un problème de 'placement contraint' : une fois qu'il existe de la surface disponible sur le composant, on procède à la reconfiguration. Dans cette étude, le placement en temps réel des tâches est simplifié par l'hypothèse que les configurations possèdent la même largeur.

* l'ordonnancement est exécuté par un algorithme de list-scheduling modifié avec une formulation intéressante de la priorité des tâches. La fonction de coût intègre un terme de pénalité pour chaque violation des contraintes de temps.

- l'approche détaillée dans [127] : cette approche part d'une spécification sous forme de graphe de tâches (graphe de dépendance de données acyclique et orienté). Elle cible une architecture formée d'un processeur connecté à un composant à reconfiguration dynamique totale à contexte unique. Cette étude traite le problème de partitionnement logiciel/matériel de la manière suivante :

* l'affectation (le partitionnement spatial) se fait en utilisant une technique de migration du SW vers le HW. Cette téchnique a tendance à mettre le plus de tâches sur le reconfigurable ce qui est logique lorsqu'on cible un composant reconfigurable totalement.

* le partitionnement temporel et l'ordonnancement sont résolus à l'aide d'un algorithme de list-scheduling modifié dont l'objectif est de minimiser le temps d'exécution total sous une contrainte de surface maximale.

Notre approche a la particularité de partir d'une spécification plus riche que celles citées précedemment (elle intégre du flot de données dans le flot de contrôle de l'application). Elle cible une architecture à reconfiguration dynamique partielle (non à chaud) ou totale (à contexte unique). Nos objectifs sont les mêmes que ceux de l'étude détaillée dans [127] : optimiser le temps d'exécution total de l'application sous une contrainte de surface maximale de l'UCR.

Conclusion

Dans cette partie, nous avons détaillé le modèle d'application choisi (le formalisme SSM d'*Esterel*). Une fois le système modélisé sous *Esterel*, il est nécessaire de vérifier a posteriori que l'hypothèse synchrone est vraiment respectée, c'est à dire que la durée de calcul est inférieure au temps de réaction du système.

PARTIE 3

Outils et Validation de la méthodologie

Sommaire

Objet

Dans le premier chapitre de cette partie, nous présentons l'ensemble des outils développés dans le cadre de ce travail ainsi que leur enchaînement dans la construction du flot de notre méthodologie de partitionnement.

Le second chapitre traite de la validation de notre méthodologie sur la base de deux application test : une caméra intelligente qui réalise l'étiquetage et le suivi d'objet en mouvement, et une application de robot mobile autonome sous-marin pour la cartographie de fonds marins.

CHAPITRE 1

Outils développés

Introduction

Dans ce chapitre, nous présentons l'ensemble des outils développés pour mettre en œuvre notre méthodologie détaillée dans la partie précédante. Le flot de ces outils est illustré par la figure 3.1 parallèlement au flot de la méthodologie proposée.

Rappelons que notre méthodologie part d'une spécification en *SSM* de l'application. Le compilateur *Esterel*, grâce à un ensemble de passerelles (détaillées dans la chaine de compilation illustrée par la figure 2.7) qu'on résumera par un outil ***Strl_oc***, génère un fichier au format **oc** (représentation explicite de la mahine de Mealy de l'application).

Une étape d'analyse de ce fichier **oc** est ensuite effectuée. Le but étant de construire les graphes de flots de données (GFD) correspondants aux appels de procédures externes définis dans la spécification et sur lesquels est effectué le partitionnement. Cette étape est effectuée à l'aide de l'outil ***OCDataExtractor*** tel qu'illustré sur la figure 3.1.

Le partitionnement opère sur ces GFD dont les tâches sont caractérisées par des estimations de temps d'exécution et d'utilisation de ressources. Le partitionnement utilise aussi et muni de l'ensemble de ces informations produit une affectation des tâches, leur ordonnancement ainsi que de la composition

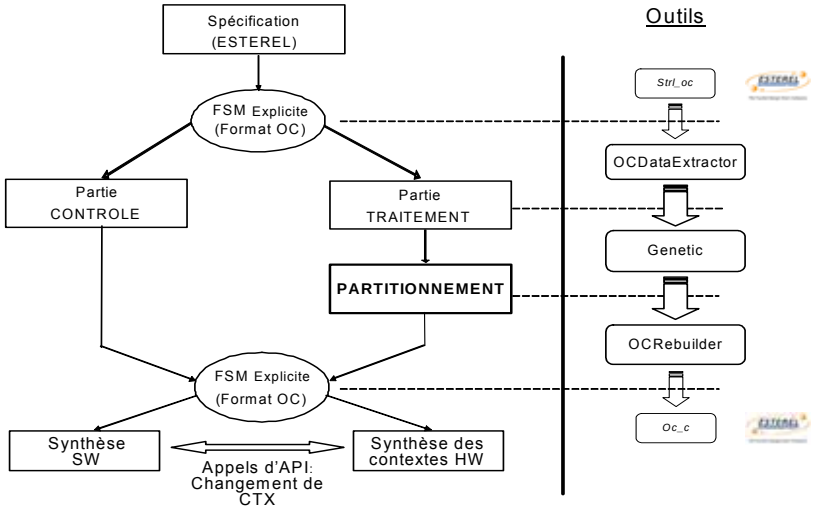


FIGURE 3.1. Rappel du flot global et de l'ensemble des outils utilisés dans notre méthodologie

des contextes de reconfiguration matériels. Cette étape est effectuée par l'outil ***Genetic***.

L'outil ***OCRebuilder*** intègre ensuite les résultats de partitionnement et reconstruit un fichier **oc** sémantiquement correct.

Une fois le fichier **oc** reconstruit, on utilise un autre outil de la chaine de compilation *Esterel*, l'outil ***oc_c*** qui génère du code C à partir du fichier **oc** contenant tout le contrôle et le traitement de l'application :

- le traitement logiciel sous forme d'appels logiciels.

- le traitement sur la partie reconfigurable par des appels (logiciels aussi) aux procédures matérielles via des APIs spécifiques.

FIGURE 3.2. Schéma de l'IHM principale *Epigui*

Détaillons maintenant les différents outils développés dans le cadre de ce travail et qui sont accessibles depuis une IHM principale appelée *Epigui* illustrée par la figure 3.2.

1. L'outil OCDataExtractor

Sur la base d'un *parser* du format **oc** (communiqué par l'INRIA Sophia Antipolis) et nous avons réalisé l'outil ***OCDataExtractor*** [160] qui traduit l'automate **oc** en une représentation interne (le *graphe de contrôle*) qui renferme toutes les informations dont a besoin le partitionnement.

Cet outil commence par analyser les dépendances entre les procédures externes pour ensuite générer un Graphe de Flot de Données (GFD) pour chaque chemin (ou transition) non-rédondant(e) de la machine d'états.

Essayons maintenant de récapituler les entrées / sorties de l'outil ***OCDataExtractor*** (l'IHM de l'outil est illustrée par la figure 3.3). Cet outil prend donc en entrée :

- le fichier **oc** généré à partir d'Esterel Studio.

- le fichier d'implantation global qui contient les points d'implantation de chacune des procédures présentes dans le fichier **oc** (en terme de temps d'exécution, de nombre de CL et de CD). Les valeurs de ces points d'implantation sont issues de la phase d'estimation.

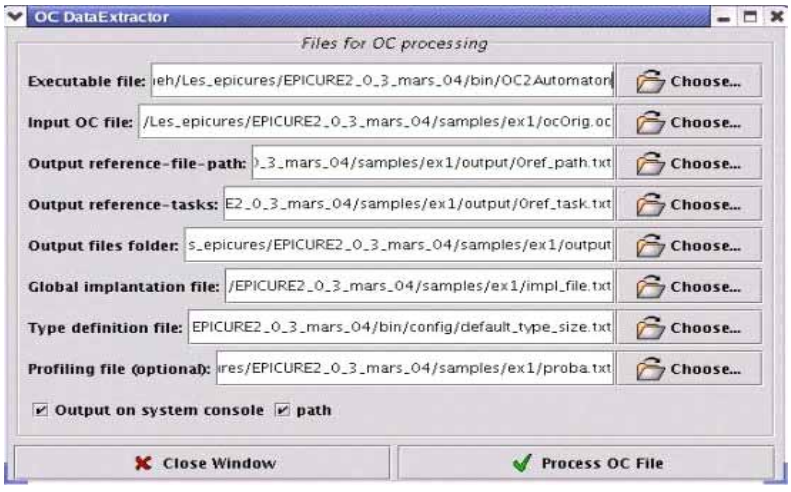


FIGURE 3.3. L'IHM de l'outil OCDataExtractor

- le fichier contenant la taille en octets des différents types (types utilisateurs inclus) utilisés dans la spécification en SSM.

- le fichier de '*profiling*', avec les probabilités de passage par les différentes transitions de la machine d'états, s'il est disponible.

- l'option de construction des GFD : par transition / par chemin.

Et l'outil génère en sortie :

- un fichier GFD pour chaque transition (ou chemin) avec les dépendances entre les différentes tâches et entre les tâches et les nœuds mémoire avec le nombre correspondant d'octets transférés sur chaque arc.

- un fichier implantation relatif à chaque GFD, construit à partir du fichier d'implantation global.

- un fichier DEP relatif à chaque GFD, explicitant les dépendances entre les tâches (fichier utile pour la reconstruction du fichier **oc**).

- un fichier de correspondance entre le *numéro_gfd* d'une tâche donnée dans un GFD et son numéro **oc** (le fichier reference-task de l'IHM). Ce dernier numéro est unique alors qu'une tâche peut avoir plusieurs *numéro_gfd* dans différents GFD.

Les informations en sortie de cet outil sont d'une grande utilité pour la phase de partitionnement ainsi que pour la phase de reconstruction du fichier **oc**.

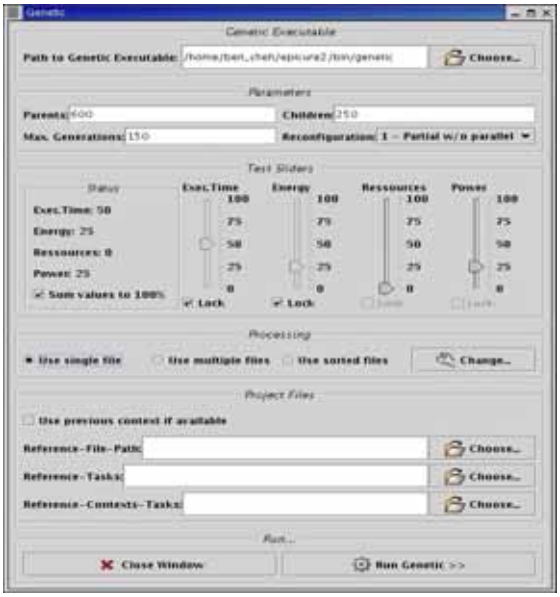


FIGURE 3.4. IHM de l'outil *Genetic*

2. L'outil Genetic

L'algorithme génétique utilisé au sein de l'outil ***Genetic*** réalise le partitionnement sur chcun des GFD non-redondants fournis par l'outil ***OCDataExtractor***.

Les paramètres et fichiers utilisés pour effectuer le partitionnement par ***Genetic*** sont :

- N_{Indiv} : la taille de la population initiale.

- $N_{enfants}$: le nombre d'enfants générés à chaque itération.

- N_{gen} : la condition d'arrêt qui représente le nombre maximum de générations sans amélioration du meilleur individu.

- *Reconf* : le paramètre relatif au type de reconfiguration utilisé. La reconfiguration peut être :

  . TOTALE : temps de reconfiguration fixe.

  . PARTIELLE : temps de reconfiguration calculé par l'équation (2.2).

  . PAS DE RECONFIGURATION : un seul contexte matériel équivalent à un ASIC.

- les poids utilisés dans la fonction de coût multicritère (détaillée dans le chapitre 3 de la partie 2 dans le pargraphe relatif aux extensions proposées

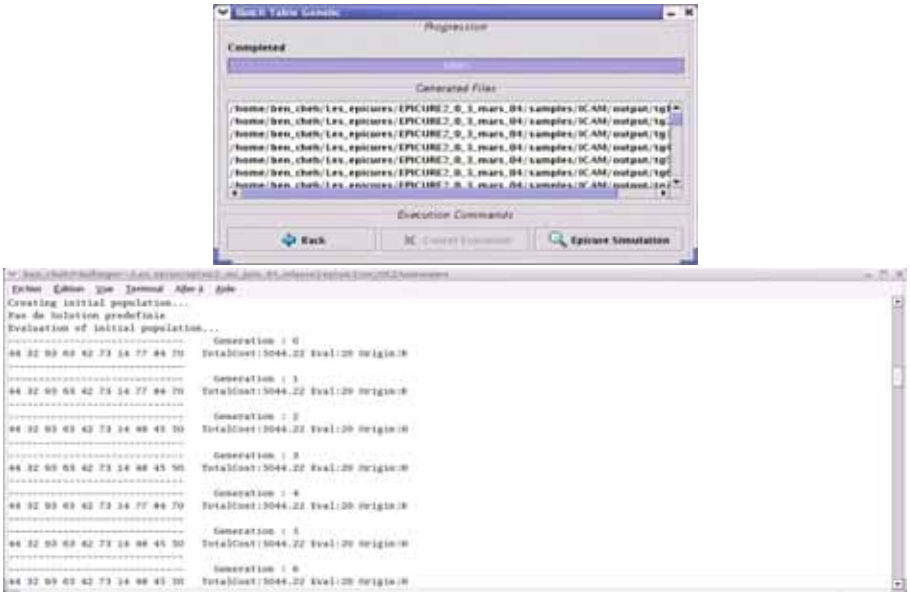


FIGURE 3.5. Progression de la convergence de l'outil genetic

pour prendre en compte la consommation) afin de mettre plus ou moins l'accent sur les différents paramètres à optimiser (le temps d'exécution, la puissance dissipée, l'énergie consommée, l'utilisation des ressources). Ces paramètres sont saisis à partir de l'interface graphique illustrée par la figure 3.4.

- les fichiers GFD générés par l'outil ***OCDataExtractor***.

- les fichiers d'implantations correspondants.

- un fichier de description de l'architecture comprenant des valeurs de paramètres architecturaux comme la taille de l'UCR en terme de CL et de CD, les largeurs des bus, leurs débits, le temps de reconfiguration par CL, la taille mémoire au niveau d'*ICURE*.

- une solution prédéfinie (optionnelle) sous forme d'un chromosome à injecter dans la population initiale (peut être le chromosome solution d'une exécution précédente de ***Genetic***). Le concepteur peut dans ce cas proposer une solution pour tenter de guider l'évolution de l'algorithme.

- le type d'élitisme à considérer :

  . ELITISM_0 : On mélange la population courante avec celle des enfants et on en garde les meilleurs.

  . ELITISM_1 : On garde les meilleurs (N_{Indiv} - $N_{enfants}$) de la population courante et on y ajoute les $N_{enfants}$ de la population des enfants.

- l'option EVOLUTION_DYNAMIQUE / STATIQUE du nombre d'individus produits par chaque opérateur selon sa performance à générer des bons individus.

L'outil de partitionnement ***Genetic*** opère sur chaque GFD de l'application comme illustré par la figure 3.5 et génère en sortie, pour chaque GFD :

- une table contenant l'ensemble des contextes ordonnancés sur le processeur[1] et le reconfigurable ainsi que les communications requises pour transférer les données, et les opérations de reconfiguration.

- un fichier, dit de *simulation* qui permet de visualiser le résultat de partitionnment sous forme d'un diagramme de Gantt. La figure 3.6 illustre un résultat de partitionnement avec l'ordonnancement sur l'UCR (trois contextes matériels sont représentés, chacun précédé d'un rectangle en oranger représantant le temps de reconfiguration associé), sur le CPU ainsi que l'occupation de la mémoire d'*ICURE*.

1. Afin de garder l'hypothèse qu'un seul contexte est actif à la fois sur une ressource donnée, on peut parler de contexte SW formé d'une seule tâche.

FIGURE 3.6. La *simulation* résultante après partitionnement

- l'énergie totale consommée ainsi qu'un fichier retraçant le chronogramme des pics de puissance (qui se greffe au diagramme de Gantt du fichier *simulation*, dans le cadre de l'extension proposée à la fin de la partie 2).

L'item EPICURE PROJECT de l'IHM de la figure 3.2 permet de récapituler les résultats de partitionnement de chaque GFD (qu'il soit construit sur une transition ou sur un chemin de la machine d'états). Son activation affiche des informations relatives à chaque GFD telles que :

- Le temps d'exécution total (en µs).

- Les nombres maximum de CL et de CD utilisés par contexte.

- La composition de chaque contexte (logiciel ou matériel).

La figure 3.7 donne l'exemple d'un résultat de partitionnement issu de l'application ICAM décrite au prochain chapitre. Le GFD #6 sélectionné dans cet exemple, possède un temps d'exécution total de 22954,4 µs et utilise un nombre maximum de 1292 CL et 52 CD par contexte. Le partitionnement de ce GFD donne deux contextes matériels (les contextes #2 et #5), le premier (#2) contenant par exemple les tâches IC_SUBSTRACT, IC_DECALAGE, IC_COPY_DATA, IC_ADD etc.

EPICURE PROJECT permet aussi d'exporter les résultats de partitionnement de toute l'application au format html et de *simuler* ce résultat (comme illustré dans la figure 3.6) GFD par GFD.

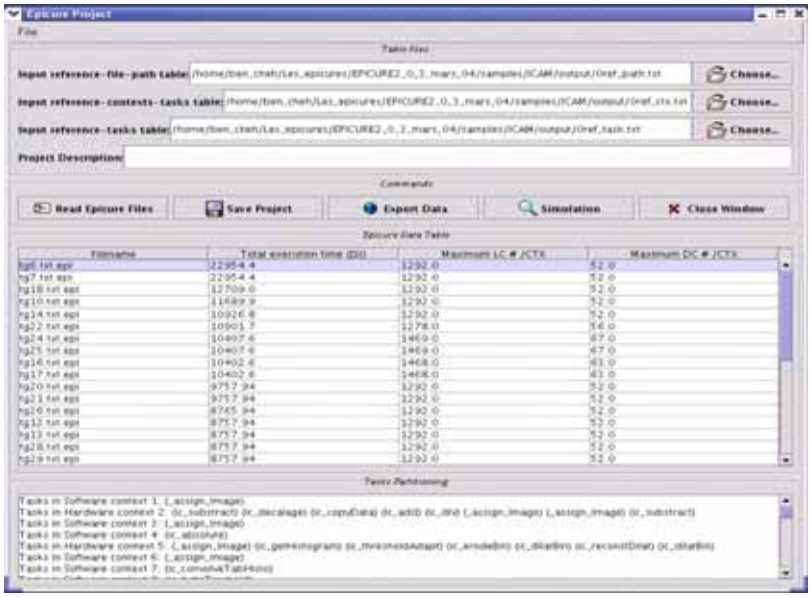


FIGURE 3.7. **Récapitulation des résultats au sein d'EPICURE PROJECT**

3. L'outil OCRebuilder

L'outil ***OCRebuilder*** a pour but d'enrichir le fichier **oc** original avec les informations issues de l'étape de partitionnement et de mettre en forme les tables d'ordonnancement selon le schéma d'exécution détaillé dans le chapitre 2 de la partie 2.

L'outil ***OCRebuilder*** prend en entrée :

- le fichier **oc** original.

- la table des contextes issue de l'outil ***Genetic*** et générée pour chaque transition (les chemins ne sont pas pour le moment pris en compte).

- le fichier DEP relatif à chaque transition (issu de l'outil ***OCDataExtractor***) contenant les dépendances entre les tâches.

Cet outil génère en sortie :

- le fichier **oc** reconstruit comme indiqué dans le chapitre 2 de la partie 2.

- la table d'ordonnancement relative à chaque transition sous forme d'un entête de fichier C.

Une fois le fichier **oc** reconstruit, on génère le code C correspondant grâce à l'outil ***oc_c*** du compilateur *Esterel*. Ce fichier C généré fait appel à la procé-

dure *Schedule()* avec, comme paramètre, un pointeur sur la table d'ordonnancement de la transition à ordonnancer. En incluant l'entête contenant les tables d'ordonnancement, cette procédure gère par interruptions l'ordonnancement du GFD correspondant selon le schéma détaillé dans le chapitre 2 de la partie 2.

Conclusion

Dans ce chapitre, nous avons présenté l'ensemble des outils conçus et utilisés dans le cadre de ce travail. Dans le chapitre suivant nous exposons quelques résultats de l'application de notre flot de partitionnement à deux exemples. Le premier est une application de type flot de donnée : la détection de mouvement exécutée sur une caméra intelligente (ICAM) et le deuxième exemple est une application plus complête (comprenant du traitement et du contrôle) de robotique sous-marine.

CHAPITRE 2
Validation de la Méthodologie

Introduction

Dans ce chapitre, des résultats permettant la validation de notre méthodologie sont exposés sur deux applications test. La première est une application de détection de mouvement sur fond d'image fixe. La deuxième est une application de robotique sous-marine.

1. Résultats de partitionnement sur une application de détection de mouvement

1. 1. Présentation de l'application de vidéo-surveillance

L'application de vidéo-surveillance considérée lors de ce travail a été proposée par le *CEA* dans le cadre du projet EPICURE. Elle est issue du projet ICAM[1] élaboré au sein du laboratoire LCEI du *CEA* en collaboration avec *Atmel*, *Thales*, *Thomson*, *Philips*, *WV*, *DC*, *Faurecia*, *Siemens*, *Alstom*, la *RATP*, *CMM* et le *TIMA*. Ce projet a pour but de développer une caméra intelligente à faible coût, à faible volume (cm^3) et ayant une très forte puissance de calcul embarquée. Les applications qui ont motivé ce projet sont :

1. ICAM : Intelligent CAMera

- L'assistance au conducteur : Embarquée au sein d'un véhicule, ICAM permet de détecter des piétons (figure 3.8 a), des obstacles et rappelle au conducteur la signalisation.

- La vidéo-surveillance : ICAM peut être utilisée aussi bien pour des opérations de vidéo-surveillance d'un quai de gare (figure 3.8 b) que pour compter les véhicules ou détecter les accidents.

Comme illustré sur la figure 3.9, l'application de vidéo-surveillance considérée assure la détection et l'étiquetage des objets en mouvement par rapport à un fond d'image fixe (la figure 3.8 c donne l'exemple de la vidéo-surveillance d'une portion d'autoroute où les voitures sont caractérisées et entourées par une enveloppe). Cette application possède une contrainte temps réel de 40 ms par image.

L'algorithme de l'application est illustré par la figure 3.10 et consiste en un ensemble (séquentiel) d'étapes :

- le moyennage sur N images: il permet au traitement d'être moins sensible au bruit. Cette étape consiste à faire N additions (appels à la procédure ADD) et une division par N (procédure DIV).

- la soustraction : fait une différence entre l'image de fond et l'image moyennée pour détecter les zones en mouvement (procédure SUB).

- la valeur absolue : pour garder une image résultat en niveaux de gris (entre 0 et 255) (appel à la procédure ABS).

- le seuillage adaptatif : permet de binariser l'image afin d'isoler les objets en mouvement. Le seuil correspond au niveau de gris qui annule le gradient de l'histogramme de l'image. Cette étape intègre les procédures GET_HISTO qui calcule l'histogramme de l'image, CONVOLTABHISTO qui calcule le gradient de l'histogramme et la procédure SEUIL qui procède au seuillage de l'image considérant la valeur calculée du seuil.

a b c

FIGURE 3.8. Exemple d'applications traitées dans le cadre du projet ICAM

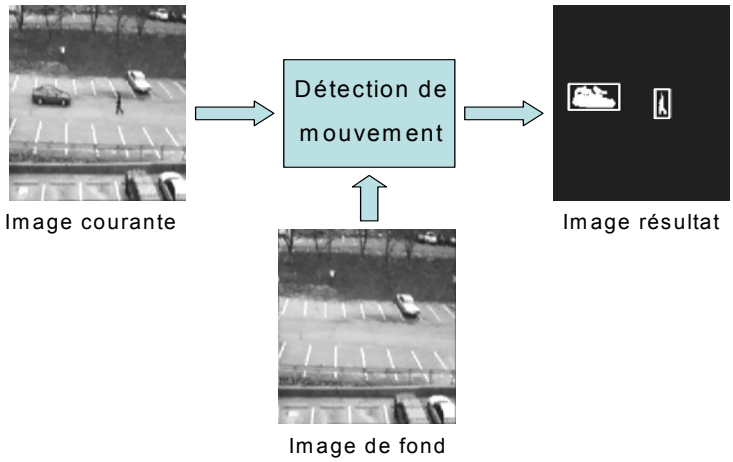


FIGURE 3.9. Principe de la procédure de détection de mouvement

- le traitement morphologique : permet de filtrer l'image seuillée pour faire disparaitre les points isolés. Cette étape fait appel aux procédures : ERODBIN qui procède à l'érosion de l'image, DILATBIN qui procède à sa dilatation (procédure appelée deux fois avec deux tailles différentes de masques), et RECONS à sa reconstruction.

- la mise à jour du fond de l'image : permet de calculer les centres de gravité des objets (procédure ETIQUET). Si au bout d'un certain temps, un centre de gravité garde les mêmes coordonnées, l'objet correspondant intègre le fond de l'image. Sinon, cet objet est considéré en mouvement et

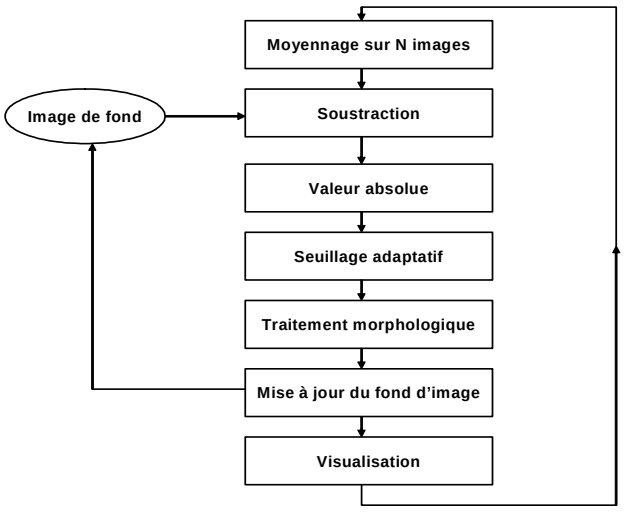


FIGURE 3.10. Schéma de principe de l'application

une enveloppe englobante est affichée tout autour comme illustré dans la figure 3.9 (procédure ENVELOP).

1. 2. Modélisation sous Esterel Studio et extraction des GFD

Le point de départ de notre spécification de l'application de détection de mouvement était un ensemble de procédures en C AINSI non optimisées pour une plateforme particulière. Cette application contient très peu de contrôle et beaucoup de traitement. L'algorithme de traitement de base est séquentiel et répétitif, ce qui limite de facto l'efficacité d'une architecture parallèle.

Nous avons donc introduit du parallélisme et du contrôle dans la spécification en Esterel Studio en exécutant en mode pipeline la partie de l'application qui correspond au moyennage, à la soustraction et à la valeur absolue. Ceci est possible si on se permet un décalage d'une image (ou si l'acquisition n'est pas faite en temps réel).

Ce pipeline fonctionne de la manière suivante : tant que l'image de fond reste inchangée, on peut exécuter en parallèle les fonctions de seuillage adaptatif, de traitement morphologique et de mise à jour de l'image de fond sur l'image courante avec le moyennage, la soustraction et la valeur absolue sur l'image suivante. Dans le cas contraire, il est necessaire de réexécuter la soustraction et la valeur absolue avec l'image de fond mise à jour. La représentation sous forme de SSM de l'application après ces transformations est donnée par la figure 3.11. Considérant une décomposition gros grain de l'application, on arrive à un total de 16 procédures C qui apparaissent dans les macro-états de la description SSM de la figure 3.11.

La machine d'états générée par le compilateur Esterel contient 16 états. L'outil ***OCDataExtractor*** est ensuite utilisé pour construire l'ensembles des GFD. En éliminant les GFD redondants, l'outil retient 14 GFD (sur un total de 20) pour une construction par transition et 10 (sur un total de 12) pour une construction par chemin. Ces GFD font apparaitre un nombre maximum de 5 appels de procédures lorsqu'on considère une approche par transition et 16 dans le cas de l'approche par chemin (ce chemin représentant le chemin nominal de l'application regroupant toutes les 16 procédures de l'application).

Pour effectuer le partitionnement de ces GFD, il est nécessaire de disposer de points d'implantation associés à chaque procédure.

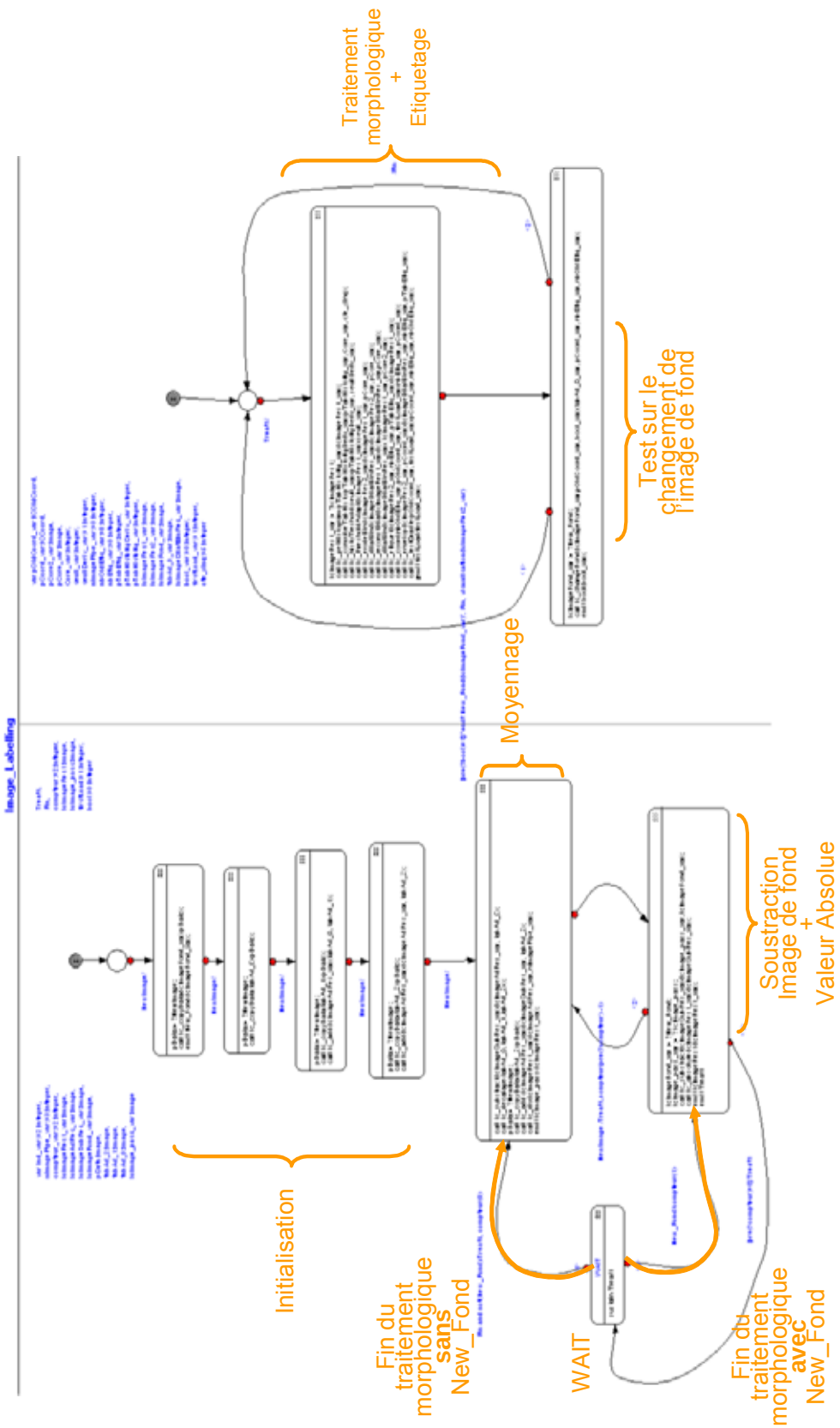


FIGURE 3.11. Spécification sous Esterel Studio de l'application ICAM

TABLEAU 3.1. Temps d'exécution SW des procédures de l'application ICAM

Nom Procédure	$T_{exe}(\mu s)$
IC_RECONSTDILAT	12689
IC_ETIQUET	11763
IC_DIV	10374
IC_ENVELOP	4086
IC_DILATBIN1	3965
IC_DILATBIN2	3965
IC_SUBSTRACT2	3659
IC_ABSOLUTE	3481
IC_SUBSTRACT1	3476
IC_ADD2	3473
IC_ADD1	3471
IC_ERODBIN	3298
GET_HISTOGRAM	3101
THRES_ADAPT	3074
CONVOLTABHISTO	825
READ_IMAGE	471
SEUIL	3
TOTAL	76448

Les points d'implantation HW sont issus d'une phase d'estimation menée au laboratoire LESTER sur la base de l'architecture Virtex-E de Xilinx alors que les points d'implantation SW sont issus de mesures effectuées au CEA sur un ARM 922 (en moyennant sur une séquence de 500 images). Les temps d'exécution par procédure sur le ARM 922 sont donnés dans le tableau 3.1 et le poids relatif de chaque procédure dans l'exécution globale est illustré sur la figure 3.12. Une sélection de quelques points d'implantations des procédures de l'application ICAM est donnée en Annexe I de ce manuscrit.

Un extrait du fichier d'implantation global est illustré par la figure 3.13. La première implantation (ligne) de chaque procédure correspond au point d'implantation logicielle.

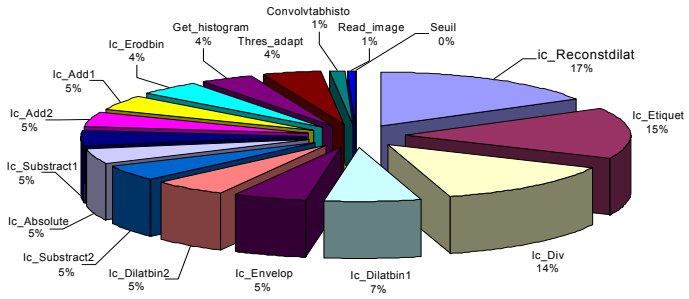


FIGURE 3.12. Pourcentage des temps d'exécution SW des procédures de l'application de détection de mouvement

Numéro OC	Nom de la tache	Temps d'exécution (ns)	CL #	CD #
5	ic_absolute	3481000	0	0
5	ic_absolute	270393.6	316	6
3	ic_add	3471000	0	0
3	ic_add	83270.4	88	12
3	ic_add	83270.5	88	8
3	ic_add	83270.8	70	8
3	ic_add	92523.1	70	6
17	ic_changeFond	2000	0	0
17	ic_changeFond	190.1	316	10
17	ic_changeFond	226.9	316	6

FIGURE 3.13. Extrait du fichier d'implantation global de l'application de détection de mouvement

1. 3. Résultats de Partitionnement

L'outil ***Genetic*** procède au partitionnement des différents GFD et dans le cas de cette application, nous avons considéré les paramètres suivants :

- une taille de population initiale N_{Indiv} = 80 individus générés aléatoirement.

- un nombre d'enfants générés à chaque itération $N_{enfants}$ = 40 individus.

- la condition d'arrêt N_{gen} = 50 générations sans l'amélioration du meilleur individu.

- l'option *Reconf* = PARTIELLE.

- un fichier de description de l'architecture illustré par la figure 3.14 avec un temps de reconfiguration de 5µs/CL, un nombre de CL de 43200, et un nombre de CD (ici sous forme de Block RAM) de 150 correspondant aux caractéristiques du Virtex XCV2000E de Xilinx. (Notons que le paramètre *Cache_size* n'est pas pris en compte).

- l'option ELITISM_0 est utilisée dans l'exécution illustrée par la figure 3.15 et l'option ELITISM_1 est utilisée dans celle illustrée par la figure 3.16.

Le temps d'exécution de l'algorithme génétique sur l'ensemble des 10 chemins est d'environ 2 minutes. Au cours de l'évolution de l'AG pendant le partitionnement d'un de ces chemins, on peut constater que, jusqu'à la fin de l'exécution, une convergence est observée comme illustré sur la figure 3.15 où la courbe rouge correspond à l'évolution du coût moyen sur l'ensemble des générations et la courbe verte à celle du meilleur individu.

Rec_time/CL(µs)		*Cache_size*		
5		*1*		
Max_Ress(#CL)		*Max_Ress(#CD)*		
43200		*150*		
$Bus_{ICURE-UCR}$ width(bytes)		*$Bus_{ICURE-UCR}$ cycle(µs)*		*$Bus_{ICURE-UCR}$ access_time (µs)*
256		*0.02*		*0.1*
$Bus_{CPU-ICURE}$ width(bytes)		*$Bus_{CPU-ICURE}$ cycle(µs)*		*$Bus_{CPU-ICURE}$ access_time (µs)*
32		*0.01*		*0.1*
Mem_width(bytes)		*Mem_access_time(µs)*		*ICURE_Mem_size(bytes)*
4		*0.05*		*200000*

FIGURE 3.14. Exemple de fichier de description de l'architecture

Sur cette figure (ainsi que la figure 3.16) on remarque bien qu'une fois que le coût du meilleur individu (courbe verte) est resté stable pendant 100 générations (condition d'arrêt utilisée), l'exécution de l'algorithme s'arrête.

Le fait de garder tous les enfants et de les inclure dans la population courante (l'option ELITISM_1 de la figure 3.16) permet de garder une diversité importante jusqu'à la fin de la convergence. Ceci est illustré par un écart type (la courbe en bleu foncé) et une moyenne (la courbe en rouge) assez fluctuants.

Vu l'irrégularité de l'espace des solutions du problème que nous traitons, il est important de permettre de faire des sauts dans cet espace pour explorer de nouvelles zones et échapper ainsi aux minima locaux. Ceci est réalisé en utilisant l'option ELITISM_1 en réglant les paramètres relatifs à la taille de la population de départ N_{Indiv} et celle des enfants générés $N_{enfants}$ (ce qui revient en général à régler le nombre d'*élites* de la population courante à garder dans la population suivante).

L'utilisation de l'option ELITISM_0 permet (selon les tailles des populations N_{Indiv} , $N_{enfants}$) d'avoir une convergence plus rapide (la moyenne se rapproche du coût du meilleur individu au fur et à mesure de la convergence et l'écart type tend vers zéro comme l'indique la figure 3.15).

Comme la communication joue un rôle de plus en plus important dans les systèmes sur puce d'aujourd'hui, il est intéressant aussi de voir la variation du temps de communication du meilleur individu sur plusieurs générations. La courbe en bleu ciel (temps COMM) dans les figures 3.15 et 3.16 correspond au temps de communication *effectif* sans recouvrement avec les traitements sur le CPU et l'UCR.

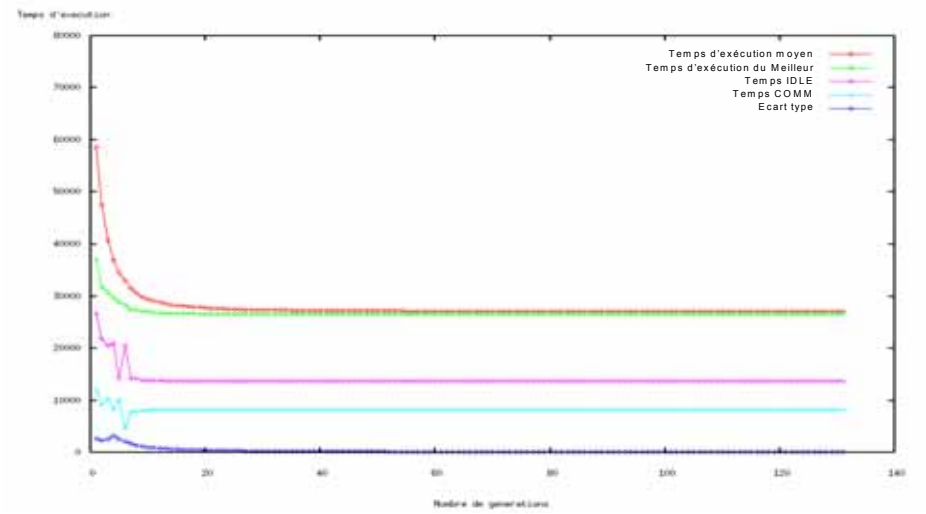


FIGURE 3.15. Coût moyen, coût du meilleur individu, temps 'IDLE', temps 'COMM' en utilisant l'option ELITISM_0

Il reste significatif par rapport au temps d'exécution total de l'application et ceci est dû principalement aux transferts de données vers (et à partir de) la mémoire d'*ICURE*. Dans le cas de l'application de traitement d'image que nous traitons et selon la taille des images considérées, ceci peut entraîner des temps de communication importants qui ne peuvent pas être en recouvrement avec des traitements.

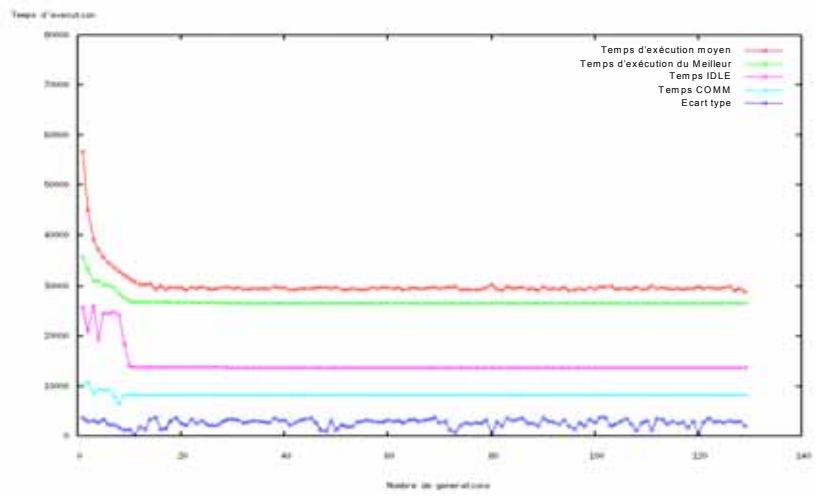


FIGURE 3.16. Coût moyen, coût du meilleur individu, temps 'IDLE', temps 'COMM' en utilisant l'option ELITISM_1

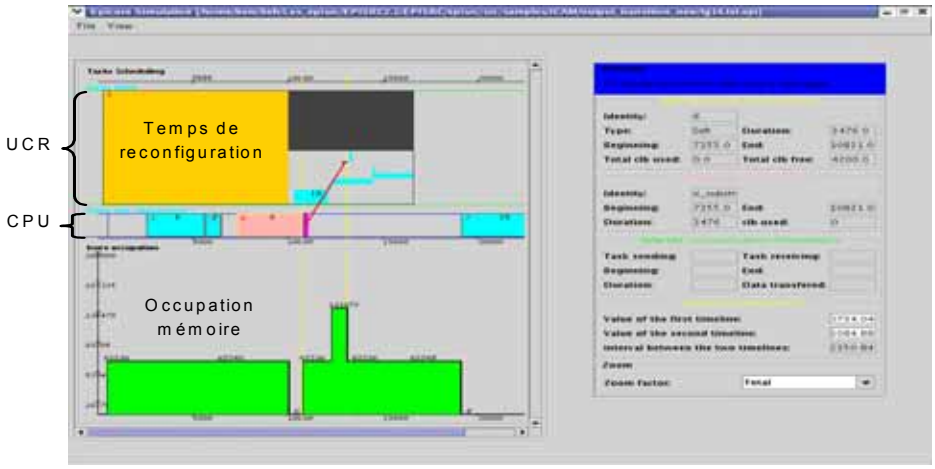


FIGURE 3.17. Une solution de partitionnement d'une transition

Les figures 3.15 et 3.16 montrent aussi la variation du temps IDLE du meilleur individu (courbe en magenta). Ce temps représente le temps de non traitement (ni exécution ni communication) sur les deux unités : l'UCR et le processeur. Ce temps décroît grâce à la capacité de la procédure de raffinement de l'AG d'utiliser au mieux les intervalles de temps inutilisés dans les ordonnancements des deux unités de traitement.

L'outil ***Genetic*** fournit en sortie, pour chaque GFD, un fichier de simulation pour la visualisation du résultat de partitionnement. La figure 3.17 donne l'exemple d'une solution de partitionnement sur une transition avec un seul contexte matériel. La tâche sélectionnée (en rose) est la tâche IC_SUBSTRACT. Elle est affectée au processeur et fait partie de l'étape de 'moyennage'. Elle est active en parallèle avec l'étape de 'traitement morphologique' comprenant des tâches (affectées au matériel) comme IC_DILATBIN et GET_HISTOGRAM. L'occupation mémoire d'*ICURE* est illustrée sur la partie basse du diagramme avec, dans le cas de la figure 3.17, des pics qui ne dépassent pas la taille spécifiée dans le fichier architecture (figure 3.14) de 200 Ko.

Dans un premier temps, nous nous sommes intéressés à l'accélération des traitements sans aborder le problème de la localité des données nécessaires à l'exécution des différentes procédures. On obtient dans ce cas les résultats suivants.

Le chemin nominal de cette application correspond au chemin #6 parmi les 10 chemins extraits et à la concaténation de 12 transitions parmi les 14 transitions extraites.

En considérant le chemin nominal dans le partitionnement, on construit un plus grand GFD que ceux extraits sur les simples transitions, et donc de plus larges contextes de configuration. Ces contextes nécessitent un plus grand

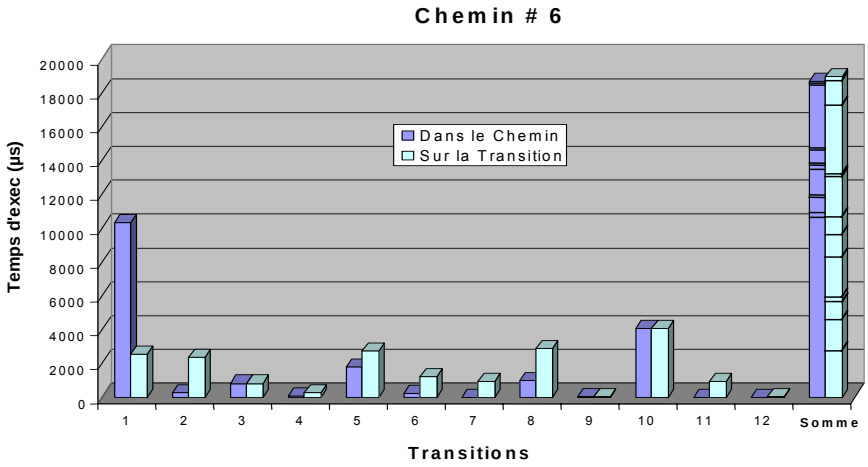


FIGURE 3.18. Détails des temps d'exécution sur le chemin nominal et des temps d'exécution sur les transitions correspondantes

temps de reconfiguration au début de l'exécution pour pouvoir accélérer plus tard les tâches finales. La figure 3.18 montre d'une part, la grande pénalité due à ce temps de reconfiguration dans la première transition au sein du chemin #6 et d'autre part, que l'accélération induite globalement est juste suffisante pour masquer ce temps de reconfiguration.

En sommant les temps d'exécution sur les transitions formant le chemin nominal, on atteint un temps d'exécution global de 18.2 ms (*OC_transition* sur la figure 3.19) et 18 ms en considérant le temps d'exécution du chemin #6 (*OC_chemin*). On compare ces résultats avec les résultats de partitionnement de l'application entière considérée comme un GFD unique (*GFD_unique*) sans pipeline et avec le temps d'exécution *(Tout_soft)* de l'application sur une cible entièrement logicielle (l'ARM 922).

Au niveau de l'occupation des ressources, le partitionnement du chemin nominal (*OC_chemin*) a donné deux contextes de configuration en ciblant le Virtex XCV2000E (43200 CL et 150 Blocks RAM). Le partitionnement par transition nécessite moins de ressources materielles car les GFD extraits sont de taille plus petite. Ce partitionnement donne dix contextes sur le XCV2000E qui sont remplis à moins de la moitié de la capacité totale du

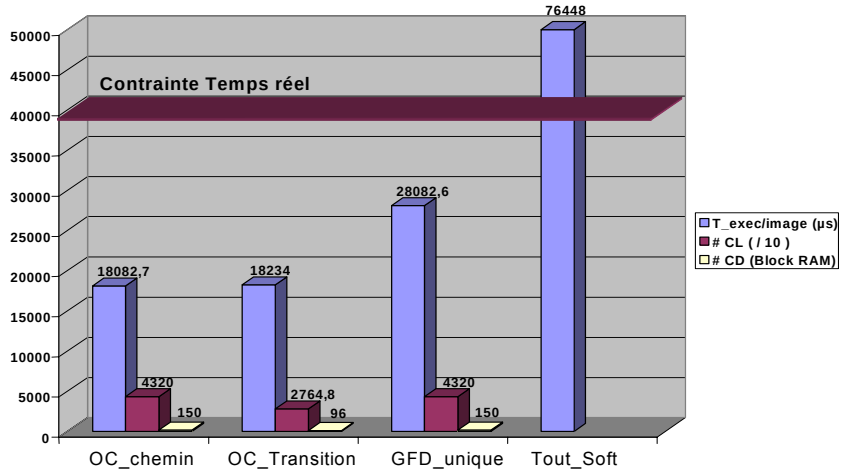


FIGURE 3.19. Temps d'exécution par image et taux d'utilisation des ressources

composanrt ce qui peut conduire à utiliser un plus petit composant de la même famille : le Virtex XCV1000E (27648 CL et 96 Blocks RAM).

Les résultats sur l'approche par chemin mettent en évidence que la stratégie utilisée dans le *Clustering* n'est sans doute pas la meilleure pour constituer les contextes aussi il serait nécessaire de reconsidérer cette fonction de *Clustering*.

Dans un deuxième temps, en adoptant le choix de lire les données en entrée à partir de la mémoire d'*ICURE* et de les y stocker une fois l'exécution terminée, on pénalise beaucoup les temps d'exécution des GFD construits par transition en imposant de multiple lectures/écritures qui peuvent être faites en recouvrement avec le traitemement dans le cas de GFD construits par chemin.

En sommant les temps d'exécution sur les transitions formant le chemin nominal, on atteint un temps d'exécution global de 36.1 ms (*OC_transition* sur la figure 3.20) et 32.4 ms en considérant le temps d'exécution du chemin #6 (*OC_chemin*). La figure 3.20 montre aussi la part de communication *effective* dans le temps d'exécution total.

Il est aussi intéressant de voir le temps que prennent la récupération de l'image moyennée, sa soustraction par rapport à la nouvelle image de fond et la valeur absolue dans le cas d'une rupture de pipeline.

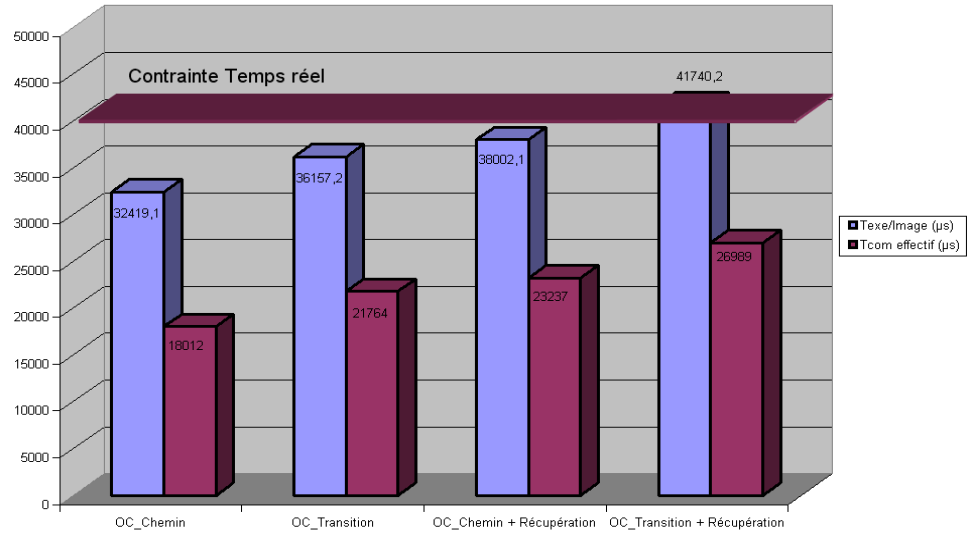


FIGURE 3.20. Temps d'exécution et contribution des communications

Ce temps est de l'ordre de 5600 µs avec 5200 µs de temps de communication. Ces temps sont additionnés aux temps d'exécution et de communication effectif de *OC_Chemin* et *OC_transition* pour créer les graphes *OC_Chemin + Récupération* et *OC_transition + Récupération*.

Nous remarquons par exemple, qu'en utilisant les paramètres architecturaux donnés par la figure 3.14, que le temps de communication *effectif* est toujours supérieur à 50% du temps d'exécution total, même dans le cas *OC_Chemin.* Ceci peut être expliqué par le fait que l'utilisation du pipeline nécessite de travailler sur deux images à la fois et de stocker en mémoire en fin d'exécution les résultats des traitements intermédiaires sur ces deux images.

Puisque les estimations dont nous disposons pour cette application ciblent principalement la famille *Virtex-E* de Xilinx (ceci fixe la taille maximale de l'UCR et son temps de reconfiguration/CL), nous avons essayé de faire varier des paramètres architecturaux comme le temps d'accès mémoire et la largeur du banc mémoire.

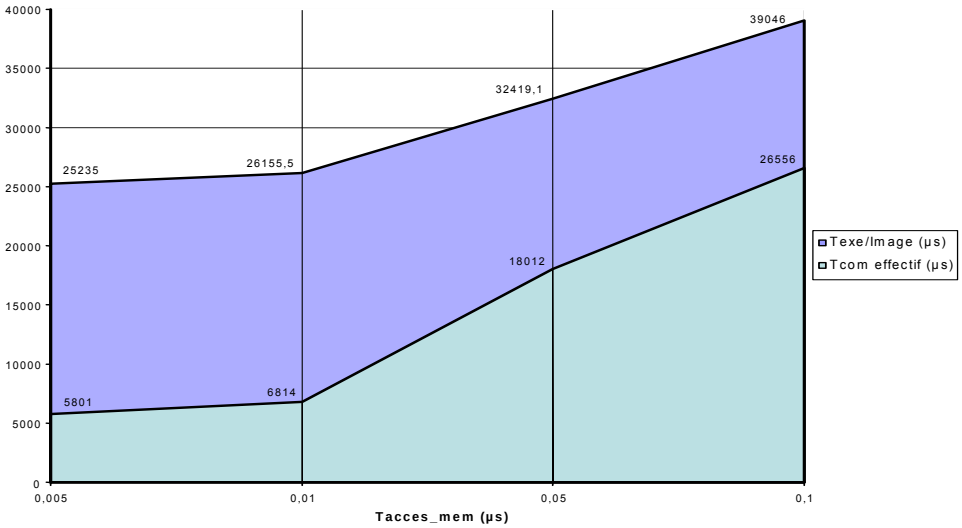


FIGURE 3.21. Variation de la contribution du temps de communication effectif au temps d'exécution total en fonction du temps d'accès mémoire

La figure 3.21 donne la variation du temps d'exécution et du temps de communication effectif en fonction du temps d'accès mémoire. Nous remarquons que le temps d'accès mémoire a un fort impact sur le temps de communication effectif total et de ce fait sur le temps d'exécution global.

En fixant le temps d'accès mémoire à 0.01 µs, nous avons fait varier la largeur du banc mémoire. Les résultats sont présentés dans le tableau 3.2.

TABLEAU 3.2. Variation du temps d'exécution et du temps de communication effectif en fonction de la largeur du banc mémoire

	T_{exe} (µs)	$T_{com\_effectif}$ (µs)	% de Communication dans T_{exe}	% de Gain par rapport à T_{exe_16bits}
Largeur_banc = 16 bits	27798,8	10031	36 %	-
Largeur_banc = 24 bits	26647	8180	30,7 %	4 %
Largeur_banc = 32 bits	26155,5	6814	26,05 %	5,9 %
Largeur_banc = 64 bits	25236	5801	22,98 %	9,2 %

2. Résultats de partitionnement sur une application de robotique sous-marine

2. 1. Présentation de l'application de robotique sous-marine

Cette application nous a été communiquée par l'équipe *SAM* de l'I3S. Elle consiste en un système temps-réel embarqué sous forme d'un robot qui porte le nom de *M.A.U.V.E* (Mini Autonomous Underwater VEhicule) [159]. Il a été conçu dans le cadre d'un projet européen MAST III qui regroupait cinq différents pays : *le CNIM, Thomson Marconi Sonar, I3S/CNRS, IFREMER* pour la France, l'*ISR/IST* (Instituto Superior Technico) pour le Portugal, le *CNR* (Consiglio Nazionale delle Ricerche) pour l'Italy, le *GMI* (Geological and Marine Instrumentation) pour le Danemark et le *MUMM* (Management Unit of the north sea Mathematical Models) pour la Belgique. Il est actuellement utilisé dans le cadre du projet SUMARE afin d'effectuer des relevés bathymétriques sur les côtes belges en mer du nord.

Il s'agit d'un robot mobile autonome sous-marin (schématisé par la figure 3.22) renfermant une électronique lui permettant de remplir une mission confiée par une interface IHM avant sa mise à l'eau. Le véhicule peut ainsi suivre une route prédéfinie tout en tenant compte des contraintes de l'environnement telles que l'irrégularité des fonds, les obstacles et les courants, grâce à la mise en place de divers capteurs et de lois d'asservissement.

A part la partie mécanique et les capteurs que contient le *MAUVE* actuel, toute la partie 'intelligente' qui s'occupe de la gestion de la mission et des traitement sous-jacents se trouve au niveau des blocs *MMNS* et *UCV* illustrés par la figure 3.22 :

- L'*UCV* : Effectue l'asservissement nécessaire afin de respecter les consignes. Il traite également les mesures des capteurs de sécurité (tels que les capteurs de tension, d'intensité, de température moteur, de présence d'eau...) afin de déclencher la remontée en urgence si nécessaire. C'est le contrôleur de bas niveau qui tourne à une fréquence de 10 Hz.

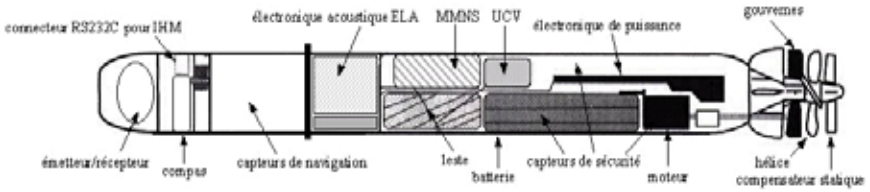


FIGURE 3.22. Schéma des principales parties du véhicule *MAUVE*

- Le *MMNS* : (Mission Management and Navigation System) est le cerveau du véhicule, il prend en compte la mission au départ, fournit les commandes à l'*UCV* en tenant compte des contraintes du monde sous-marin (courants marins, obstacles...). Le *MMNS* tourne à une fréquence de 1 Hz sur un processeur 68020. Il est formé principalement de deux modules :

  * le système de positionnement : qui lit les valeurs des capteurs non-acoustiques depuis l'*UCV*, le capteur d'évitement d'obstacles, l'altimètre et les distances aux balises pour mettre à jour la position du véhicule et estimer les courants marins.

  * le système de gestion de mission : qui est responsable du contrôle de l'exécution de la missio. Il prend les décisions à partir des valeurs actuelles des capteurs ainsi que de la position estimée. Cette partie a été modélisée en SSM par l'équipe SAM et le code C généré par *Esterel* tourne actuellement sur le *MAUVE*.

Le partitionnement dans le *MAUVE* a été réalisé empiriquement lors de la conception : Le contrôle (ainsi que quelques traitements) de haut niveau est affecté au bloc *MMNS*. Les contrôles et traitements (asservissements...) de bas niveau sont affectés à l'*UCV*. La dichotomie est faite de manière à respecter les contraintes imposées par l'application ainsi que par l'architecture.

Les différentes missions qui sont supportées pour l'instant sont :

- Garder un cap jusqu'à atteindre la cible où dépasser un temps maximum (*task_open_loop* : figure 3.23 -a-).

- Suivre une trajectoire prédéfinie constituée d'une suite de points à atteindre (en coordonnées géographiques absolues) (mission *closed-loop* ou *task_point_list* : figure 3.23 -b-). Cette mission, contrairement à la première, tient compte des courants marins et calcule en permanance les nouvelles coordonnées du point cible visé afin d'atteindre le point cible réel.

- Retourner au point de départ de la mission (*do_homing*).

Nous avons aussi ajouté à *MAUVE* une nouvelle mission propre à un autre véhicule étudié par l'équipe SAM de l'I3S : le *Phantom*. Un véhicule sous-marin actionné à distance (ROV[1]).

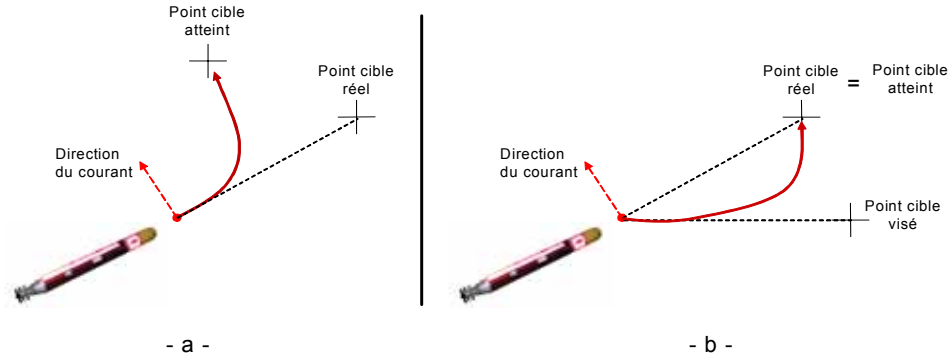


FIGURE 3.23. Missions -a- : *task_open_loop* et -b- : *task_point_list*

Il s'agit d'une mission de suivi de contours utilisant des retours de sonar (*sonar_tracking*). Le principe est de différencier deux zones à partir des retours sonar sur la partie du fond marin couverte par le robot à un instant donné. La figure 3.24 donne l'exemple d'un contour d'un banc d'algues sur un fond de sable. Après une phase d'apprentissage, où on stocke les retours de sonar sur le banc d'algues et sur le sable séparément, le véhicule suit un cap donné jusqu'à ce qu'il ait réussi à différencier deux zones distinctes puis il essaye de maintenir le contour au milieu de sa zone couverte. Pour cela, il calcule les commandes de vitesse moteur et de vitesse de rotation pour les gouvernes.

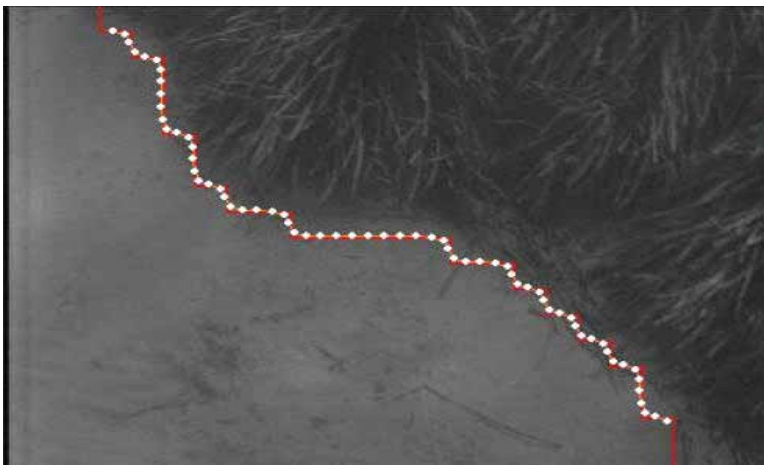


FIGURE 3.24. Mission de suivi de contour

1. ROV : Remotely Operated Vehicle

2. 2. Modélisation sous Esterel Studio

Le point de départ de notre spécification de l'application *MAUVE* était une modélisation en SSM du MMNS, faite au sein de l'équipe SAM. C'est cette spécification qui est traduite en C par le compilateur *Esterel* et embarquée dans le *MMNS* de la version actuelle du *MAUVE*.

Elle est formée principalement d'un module de gestion de la mission (*task_execution*), d'un module de surveillance des urgences (*emergency_surveillance*) et d'un module de contrôle du guidage vertical et horizontal du véhicule (*guidance*). Ces modules sont présentés dans l'**annexe 2**.

Dans cette modélisation, tous les traitements sont externalisés et le contrôle est réalisé en utilisant des signaux purs (non-valués). La SSM de départ a pour entrée des signaux dont les *status* de présence sont les résultats de calcul des procédures externes au cycle précédant de réaction du système. Elle émet en sortie d'autres signaux dont la présence (ou abscence) peut lancer l'exécution de procédures externes ou activer une remontée en urgence.

Nous avons eu accès aussi à l'ensemble des procédures C de l'application dont les temps moyens d'exécution sur le processeur *Nios* d'Altera sont repris dans le tableau 3. 2. Nous avons considéré le processeur *Nios* car il n'était pas aisé d'obtenir ces temps d'exécution sur le processeur embarqué dans *MAUVE*.

Notre travail, au niveau de la spécification a consisté dans un premier temps à intégrer les traitements relatifs aux différentes missions en y ajoutant les traitements de bas niveau effectués par l'*UCV* pour construire un modèle complet de l'application.

TABLEAU 3.3. Temps d'exécution SW des procédures de l'application *Mauve*

Nom Procédure	$T_{exe}(\mu s)$
POSITIONNEMENT	160103
INIT_POSITIONNEMENT	23770
GO_TO_POINT_UCV_ORDERS	16375
GETSONARCLASSIFICATION	7929
AUTO_HEADING_UCV_ORDER	7301
SETHOMINGPOINTASTARGET	1318
SELECTNEXTREFERENCE	1201
SELECTNEXTSONARTRACKINGREF	1201
ATTAINED	1172
SELECTNEXTPOINT	1031
GENERATETRACKINGCOMMAND	727
SELECTNEXTTASK	609
EMERGENCES	185
F_SECURE_ALTITUDE	93
F_SECURE_DEPTH	92
ACQUI_UCV	3
ACQUI_SONAR	1
ACQUI_ELA	1
CONTROL_DURATION	1

FIGURE 3.25. Représentation sous forme de boite noire de l'application *MAUVE*

Le système peut être représenté sous forme d'une boite noire, comme illustré par la figure 3.25, qui prend en entrée :

- une mission à partir de l'IHM.

- les valeurs des capteurs de navigation tels que l'assiette, la profondeur, la vitesse linéaire, la vitesse de rotation (gyro), le roulis et le cap.

- les valeurs des capteurs de sécurité tels que les capteurs de tension, d'intensité, de température moteur et de présence d'eau.

- les distances par rapport à deux balises sous-marines, positionnées à des endroits précis avant le début de la mission, ainsi que les distances par rapport à des obstacles à partir des acquisitions sonars (ELA).

et donne en sortie :

- les commandes de gouvernes (profondeur et direction).

- la propulsion (vitesse moteur).

- l'ordre d'activation du mode 'Pinger' en cas d'une remontée en urgence du véhicule. Ce mode permet de forcer le sonar du véhicule en émission afin de pouvoir facilement le localiser pour sa récupération.

- l'historique de la mission : Au cours de la mission, toutes les valeurs des capteurs, des consignes ainsi que des commandes sont enregistrées à des instants réguliers. Une fois la mission terminée, cet historique (sorte de boite noire du véhicule) est récupéré via l'IHM pour analyser la mission.

Deux horloges coexistent dans le *MAUVE*. L'horloge du bloc MMNS qui, à une fréquence d'1 Hz, réalise les fonctions suivante :

- acquisition des données provenant des capteurs acoustiques et non-acoustiques.

- estimation de la position, de la vitesse et des courants marins.

- envoi des commandes relatives à la mission (Cap, profondeur, vitesse).

Et l'horloge de l'UCV qui, à la fréquence de 10 Hz, a pour rôle de procéder à :

- l'acquisition des données capteurs non-acoustiques et des commandes venant du MMNS.

- les asservissements de direction, d'immersion et de propulsion relatifs à la mission traitée.

- l'envoi des commandes au moteur et aux gouvernes.

Nous avons modélisé le système sous Esterel Studio (voir **Annexe 2**) sous forme d'un Macro-état global qui rejoint la représentation donnée par la figure 3.25 (avec des signaux valués en entrée et en sortie du système) et pour lui procurer le comportement détaillé précédemment, nous avons fait les choix de modélisation suivants.

Nous avons choisi comme cycle de réaction du système, le cycle d'horloge de l'UCV (de 10 Hz). Ce qui permet de s'assurer qu'entre deux tick *Esterel*, le système procède à l'acquisiton des données capteurs, le traitement et l'envois des commandes.

Nous avons regroupé les tâches d'acquisition des données des capteurs et de positionnement (regroupant l'estimation de la position, de la vitesse et des courants marins) exécutées à la fréquence d'1 Hz dans un module (ou STG) *acquisition_positionnement.* Nous avons donc imposé à ce module d'être exécuté tous les 10 tick *Esterel*.

Ce module fait appel aux procédures suivantes :

- ACQUI_SONAR : acquisition des données sonar pour la mission *sonar_tracking.*

- ACQUI_ELA : acquisition des données ELA pour calculer les distances balises.

- ACQUI_UCV : acquisition des données des capteurs non-acoustiques.

- POSTITIONNEMENT : estimation de la position actuelle du véhicule, de sa vitesse et des courants marins.

Le module *acquisition_positionnement* est exécuté en parallèle avec le module *task_execution* qui gère le séquencement du déroulement de la mission globale (chargée à partir de l'IHM) en appelant à chaque fin d'exécution d'une mission la procédure SELECTNEXTTASK. Il contrôle aussi la durée d'exécution de la mission en cours et envoie un signal d'arrêt si elle dépasse la durée qui lui est allouéee (procédure CONTROLDURATION). Les missions supportées sont les suivantes :

- la mission *task_point_list* : elle procède au suivi d'une liste de points en faisant appel aux procédures suivantes :

  * ACQUI_UCV : acquisition des données des capteurs non-acoustiques.

  * SELECTNEXTPOINT : sélection du prochain point à atteindre.

  * GO_TO_POINT_UCV_ORDERS : calcul des ordres à donner à l'UCV.

  * ATTEINED : test si le point est atteint.

  Cette mission regroupe aussi le cas particulier de la mission *do_homing* en imposant comme coordonnées du prochain point de destination les coordonnées du point de départ (appel à SETHOMINGPOINTASTARGET).

- la mission *task_open_loop* : elle consiste à suivre un cap jusqu'à atteindre la cible où dépasser un temps maximum en faisant appel aux procédures suivantes :

  * ACQUI_UCV.

  * SELECTNEXTREFERENCE : sélection de la prochaine référence (en terme de cap, profondeur, vitesse, temps de mission) à considérer.

  * AUTO_HEADING_UCV_ORDERS : transmission des ordres directement à l'UCV.

- la mission *sonar_tracking* : elle consiste à suivre un cap donné (*task_open_loop*) jusqu'à ce que l'on détecte deux zones distinctes selon les retours sonar obtenus. Une fois les deux zones distinguées, on cherche à garder le contour au milieu de la zone couverte par le sonar. Elle regroupe les procédures suivantes :

  * ACQUI_UCV.

  * ACQUI_SONAR.

  * SELECTNEXTSONARTRACKINGREF : sélection de la prochaine référence de suivi sonar en terme de cap, vitesse, profondeur.

  * READNEXTSONARPROFILE : lecture de la prochain profil sonar.

* GETSONARCLASSIFICATION : classification des retours sonar.

* GENERATETRACKINGCOMMAND : génération des ordres de commande pour le suivi de contour.

Un troisième module *emergency_surveillance* est exécuté en parallèle aux deux premiers. C'est un module qui contrôle les urgences qui peuvent se produire pendant une mission en vérifiant que les valeurs des capteurs de navigation et de sécurité se situent bien dans les intervalles de fonctionnement normaux fixés au départ. Les procédures appelées sont :

- EMERGENCES : regroupement de plusieurs contrôleurs d'urgence sur les valeurs des capteurs de sécurité.

- F_SECURE_DEPTH : sécurisation de la profondeur du véhicule.

- F_SECURE_ALTITUDE : sécurisation de l'altitude.

2. 3. Génération du fichier oc et réduction du nombre d'états

Une modélisation modulaire et parallèle permet d'avoir une meilleure visibilité du système pour une réutilisation plus aisée. Mais elle a l'inconvénient d'augmenter considérablement le nombre d'états de l'automate explicite généré.

La génération du code C à partir d'*Esterel* permet de connaître le nombre de variables registres nécessaires pour coder les états de l'automate. La version *Esterel* de départ de l'application comprenait 2^{67} états.

```
/* REGISTER VARIABLES */
static __SSC_BIT_TYPE __Mauve_R[67]
```

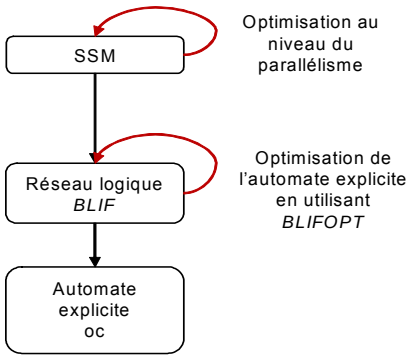


FIGURE 3.26. Schéma d'optimisation considéré

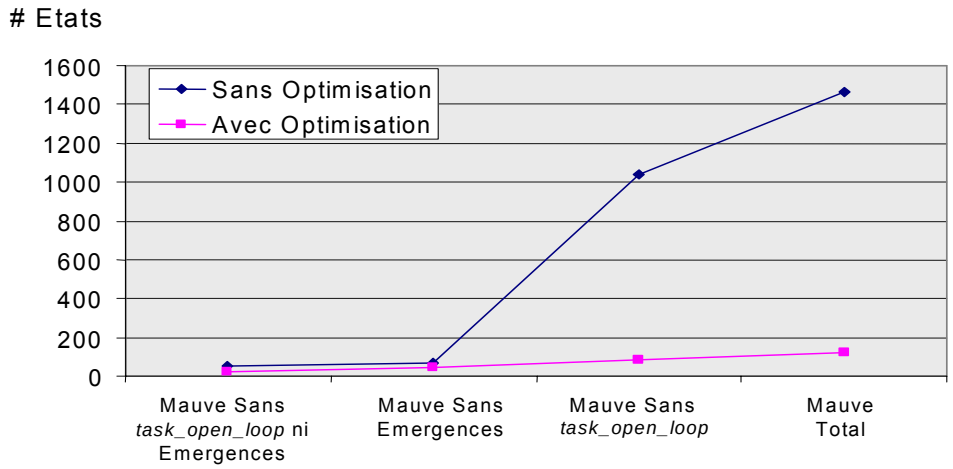


FIGURE 3.27. Variation du nombre d'états de l'automate dans les versions optimisée et non-optimisée en fonction du nombre de modules

Une optimisaton de ce nombre d'états s'imposait. Dans la chaîne de compilation d'*Esterel* V5 illustrée par la figure 2.7, cette optimisation est faite en passant par le format *BLIF* et en utilisant *BLIFOPT*. Dans le cas où on utilise des signaux valués, un fichier trace est généré en parallèle avec le fichier *BLIF* pour pouvoir ensuite réintégrer les informations sur la valeur des signaux, car *BLIF* n'opère que sur des signaux purs. Cependant, la version dont nous disposons (V5-9) ne nous permettait pas de regénérer le format **oc** une fois l'automate optimisé en passant par le fromat **ssc**.

Une autre manière de procéder est de générer le format **oc** directement à partir de *BLIF*. Un travail dans ce sens est déjà en cours au sein de l'équipe SPORTS d'I3S mais il ne couvre pour l'instant que le cas des signaux purs.

Nous avons choisi d'attaquer ce problème de réduction du nombre d'états sur deux fronts comme illutré par la figure 3.26. Le premier porte sur la SSM elle même, en essayant de diminuer le nombre de modules en parallèle. Le deuxième est l'optimisateur d'ESTEREL STUDIO en modifiant quelques scripts afin d'utiliser *BLIFOPT*.

Le résultat de cette stratégie de réduction du nombre d'états est illustré par la figure 3.27. Le nombre d'états de l'application totale passe de 1460 états avant optimisation à 123 états après optimisation. Cette figure montre aussi la variation du nombre d'états en fonction du nombre de modules considérés (avec ou sans le module Emergences) et du nombre de missions considérées (avec ou sans *task_open_loop*). Nous remarquons que le module de contrôle des urgences (Emergences) a un fort impact sur le nombre d'états total de la machine de Mealy générée.

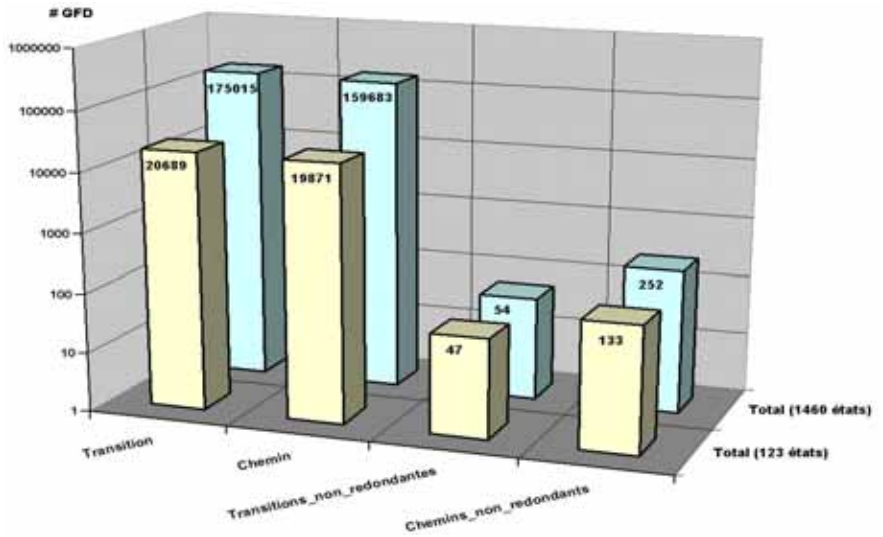


FIGURE 3.28. Gain en terme de nombre de GFD traités en éliminant les GFD redondants

2. 4. Extraction des GFD

La machine d'état générée par le compilateur *Esterel* contient 25 états si on ne considère que la mission *Point_tracking*, 27 états si on ne considère que la mission *Sonar_tracking* et 123 états pour le *Mauve_total*. L'outil ***OCDataExtractor*** est ensuite utilisé pour construire l'ensembles des GFD.

La figure 3.28 montre les nombres de ces GFD obtenus pour le Mauve_Total à 1460 états (sans optimisation du nombre d'états) et pour le Mauve_Total à 123 états (avec optimisation).

Cependant, la figure 3.28 montre que l'optimisation du nombre d'états ne suffit pas dans le cas de cette application à diminuer le nombre de GFD construits. On remarque en effet que le nombre de GFD construits pour le *Mauve_total* (que ce soit par chemin ou par transition) est de l'ordre de 20000 GFD. Ceci résulte non pas du nombre total d'états de l'application mais du nombre de transitions dans la machine de Mealy équivalente. En appliquant la technique d'élimination des GFD redondants (suite d'actions identiques sur différentes transitions) on réduit de façon spectaculaire (jusqu'à 0,03%) le nombre de GFD sur lesquels il devient nécessaire d'effectuer le partitionnement. On passe ainsi de 20689 GFD dans le cas de *Mauve_Total* avec analyse par transition à 47 GFD non-redondants. Dans le cas d'une analyse par chemin, la réduction permet de passer de 19871 à 133 GFD.

Le nombre de GFD non-redondants est supérieur dans le cas d'une analyse par chemin du fait que les suites d'actions sont plus longues à cause de la

concaténation des transitions diminuant ainsi les possibilités de retrouver des suites d'actions identiques.

La figure 3.28 montre aussi que le travail de réduction du nombre d'états par modification de la structure initiale de la représentation SSM de *MAUVE* n'a pas de grande influence sur le nombre de GFD final après la suppression des GFD redondants et le rend de ce fait superflu. Ce résultat est intéressant puisqu'il permet d'obtenir, directement, une taille de problème tout à fait tractable pour le partitionnement alors que l'explosion combinatoire du nombre d'états dans la description initiale rendait impossible le partitionnement.

L'outil ***OCDataExtractor*** utilise en entrée un fichier d'implantation global. Les points d'implantation SW sont issus de mesures que nous avons effectuées sur le processeur *Nios* d'Altera. Les temps d'exécution et nombres de ressources sur l'UCR sont fixés manuellement et de façon empirique et non pas issus d'une phase d'estimation architecturale comme dans le cas de l'application ICAM.

2. 5. Résultats de Partitionnement

L'outil ***Genetic*** procède au partitionnement des GFD non-redondants et dans le cas de l'application *MAUVE*, nous avons considéré les valeurs des paramètres suivantes :

- une taille de population initiale N_{Indiv} = 80 individus générés aléatoirement.
- un nombre d'enfants générés à chaque itération $N_{enfants}$ = 40 individus.
- la condition d'arrêt N_{gen} = 50 générations sans amélioration du meilleur individu.
- l'option *Reconf* = PARTIELLE.
- un fichier de description de l'architecture.
- l'option ELITISM_1.

Le temps d'exécution de l'algorithme génétique sur l'ensemble des 133 chemins est d'environ 15 minutes.

Puisque les missions sont exclusives, les GFD que l'on extrait en considérant une à une les missions se retrouvent dans le Mauve_Total. Pour cela, nous allons considérer dans ce qui suit deux GFD pour représenter au mieux chacune des deux missions suivantes :

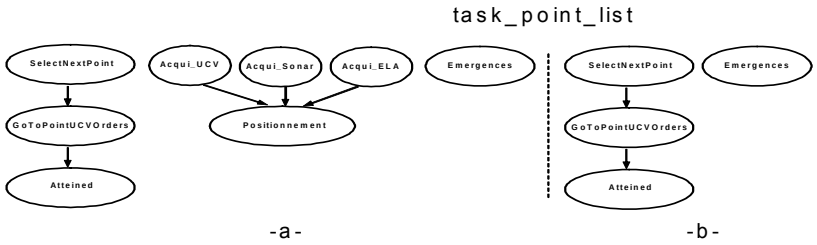


FIGURE 3.29. Quelques GFD représentatifs des différentes missions

- *task_point_list* :

  * GFD 1 : correspondant au chemin #111 de Mauve_Total (représenté par la figure 3.29 -a-). Ce chemin intègre les traitements de la mission *task_point_list* en parallèle avec les traitements effectués dans les modules *acquisition_positionnement* et *emergency_surveillance*. Il est appelé à chaque cycle du MMNS (à la fréquence d'1 Hz).

  * GFD 2 : correspondant au chemin #16 de Mauve_Total (représenté par la figure 3.29 -b-). Il intègre seulement les traitements de la mission *task_point_list* et est appelé à chaque cycle de l'UCV (à la fréquence de 10 Hz).

- *sonar_tracking* :

  * GFD 1 : correspond au chemin #113 de Mauve_Total (représenté par la figure 3.30 -a-). Il intègre les traitements de la mission *sonar_tracking* en parallèle avec les traitements effectués dans les modules *acquisition_positionnement* et *emergency_surveillance*. Il est aussi appelé à chaque cycle du MMNS.

  * GFD 2 : correspond au chemin #117 de Mauve_Total (représenté par la figure 3.30 -b-). Il est appelé à chaque cycle de l'UCV.

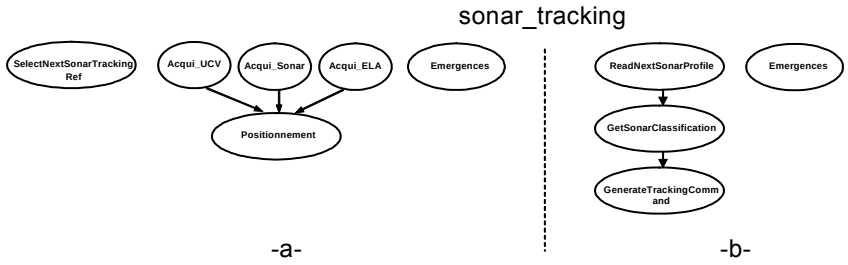


FIGURE 3.30. GFD représentants la mission Sonar_Tracking

Notons que, pour des raisons de lisibilité, ni les noeuds mémoire, ni les temps de communication sur les arcs ne sont représentés sur les figure 3.29 et 3.30.

TABLEAU 3.4. Temps d'exécution des GFD relatifs à chaque mission

	T_{exe} (µs) GFD 1	T_{exe} (µs) GFD2
task_point_list	21728,6	1922,63
sonar_tracking	20589,9	952,8

En utilisant le fichier de description de l'architecture tel qu'illustré par la figure 3.14, nous obtenons les temps d'exécution listés dans le tableau 3.4. Nous remarquons que les temps d'exécution des GFD 1 des missions *task_point_list* et *sonar_tracking* sont bien inférieurs à la contrainte de temps de 1 s fixée par l'horloge du MMNS.

De même, les temps d'exécution des GFD 2 relatifs à ces deux missions sont inférieurs à la contrainte de temps de 100 ms imposée par l'horloge de l'UCV.

Dans le cadre de cette application, nous avons essayé de faire varier des paramètres architecturaux autres que le temps d'accès mémoire ou la largeur du banc mémoire au niveau d'*ICURE*. La figure 3.31 montre par exemple la variation du temps d'exécution des GFD 1 et 2 de la mission *sonar_tracking* en fonction du temps de reconfiguration par CL de l'UCR.

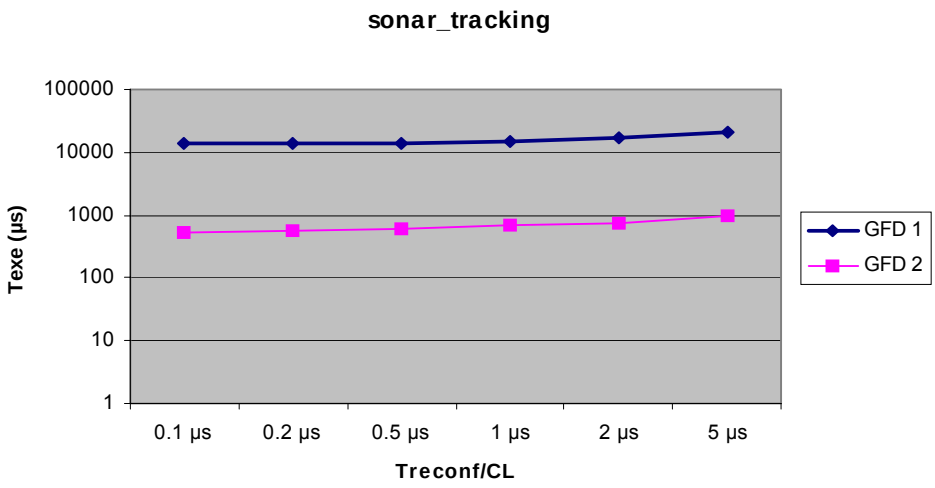


FIGURE 3.31. Variation du temps d'exécution des GFD 1 et 2 de la mission sonar_tracking en fonction du temps de reconfiguration par CL

On remarque que le temps d'exécution dépend largement du temps de reconfiguration par CL et que la tendance des deux courbes est très similaire. Ceci est dû au fait que les GFD sont de faible taille et qu'il y ait une forte disparité entre les temps d'exécution des tâches (ceci est par exemple illustré par la figure 3.32) ce qui entraîne l'apparition de larges contextes qui dictent le temps d'exécution total du GFD. Une variation du temps de reconfiguration par CL se répercute donc directement sur le temps d'exécution total.

La figure 3.32 présente une solution de partitionnement du GFD 1 de la mission *sonar_tracking*. On remarque que la tâche POSITIONNEMENT occupe à elle seule tout le contexte matériel sur l'UCR. Les tâches ACQUI_UCV, ACQUI_SONAR, ACQUI_ELA, SELECTNEXTSONARTRACKINGREF et EMERGENCE sont allouées au CPU ont des temps d'exécution très faibles par rapport au temps d'exécution de la tâche POSITIONNEMENT. De plus, les exécutions de ces tâches s'effectuent en parallèle avec la reconfiguration masquant ainsi leurs contributions au temps d'exécution global.

Nous remarquons que le fait de forcer la tâche POSITIONNEMENT à s'exécuter en un tick *Esterel* fait apparaître des GFD (contenant cette tâche) avec des temps d'exécution énormes par rapport à la moyenne des temps d'exécution des autres GFD. Ceci entraîne aussi une sous-utilisation des ressources disponibles puisque toute tâche ajoutée au contexte matériel augmente le temps de reconfiguration de ce dernier et de ce fait le temps d'exécution total du GFD. Ceci nous ammène à reconsidérer la granularité de cette tâche et de la prendre en compte au niveau de la spécification de départ.

Nous avons donc découpé l'exécution de la tâche POSITIONNEMENT sur deux ticks *Esterel* (on les nommera par la suite POSITIONNEMENT_1 et

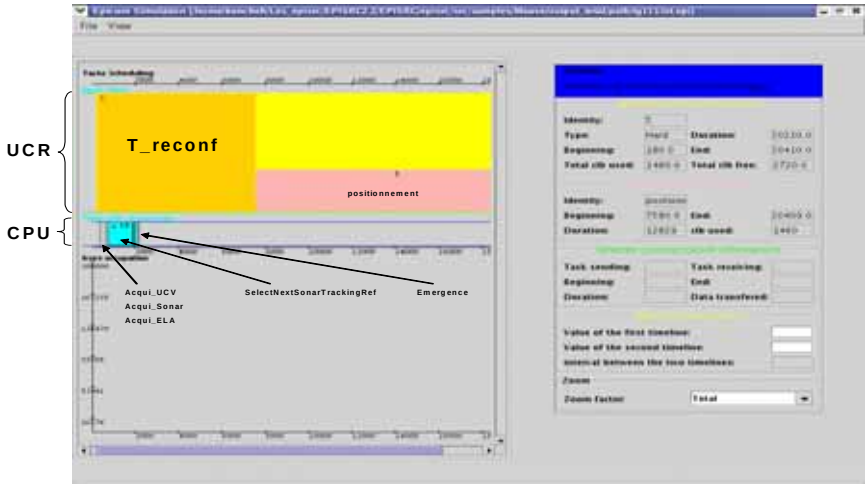


FIGURE 3.32. Résultat de partitionnement du GFD 1 de la mission *sonar_tracking*

POSITIONNEMENT_2) en gardant l'hypothèse que ces deux nouvelles procédures s'exécutent tous les 10 ticks.

Les GFD générés ne sont pas les mêmes que ceux générés à partir de la dernière spécification car les configurations (ou les états internes) ont aussi changé. Nous avons donc considéré le GFD équivalent au GFD 1 de la mission sonar_tracking pour la nouvelle spécification (on le nommera par la suite GFD 1_prim). Ce GFD présente en plus des tâches qui forment le GFD 1, les procédures READNEXTSONARPROFILE et GETSONARCLASSIFICATION.

Le résultat de partitionnement du GFD 1_prim de la mission *sonar_tracking* selon cette nouvelle spécification est donné par la figure 3.33. On remarque dans ce cas, que la tâche POSITIONNEMENT_1 reste assez prédominante sur l'ensemble des autres tâches, mais le fait qu'elle soit moins importante que la tâche POSITIONNEMENT fait que d'autres tâches (comme GETSONARCLASSIFICATION sur l'exemple de la figure 3.33) partagent le même contexte qu'elle. On remarque aussi que les périodes d'inutilisation du CPU sont moins importantes que celles illustrées par la figure 3.32.

D'autres investigations pourraient être faites pour trouver la meilleure granularité de la fonction positionnement, pour ensuite les reconduire vis à vis d'autres procédures, jusqu'à ce qu'on arrive à trouver une granularité 'convenable' pour la majorité des tâches de l'application.

Mais on peut se poser les questions suivantes :

- qu'est ce qu'une granularité 'convenable' ?

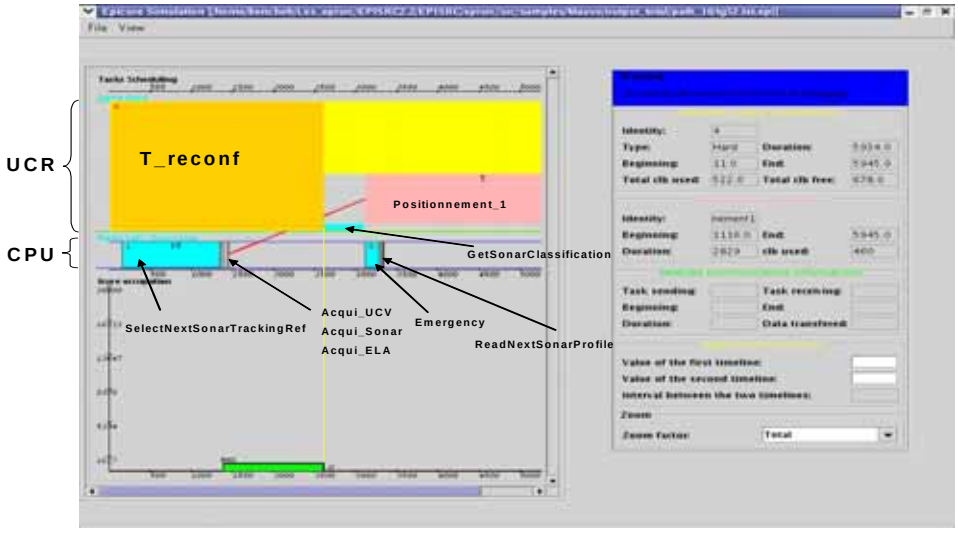


FIGURE 3.33. Résultat de partitionnement du GFD 1_prim de la mission *sonar_tracking*

- et surtout est-ce nécessaire de pousser les recherches pour trouver cette fameuse granularité ?

Une chose est sûre, c'est que tant que l'hypothèse synchrone est vérifiée, c'est à dire que la durée de calcul est inférieure au temps de réaction du système, nous pouvons être sûr de la bonne exécution de l'application issue de notre flot de partitionnement.

Dans le cas contraire, on pourrait imaginer un couplage plus important entre estimation (DesignTrotter) et partitionnement pour ajuster la bonne granularité. Mais ceci dépasse le cadre de notre travail de thèse et constitue un axe de recherche intéressant pour des études futures.

Conclusion :

Dans ce chapitre, nous avons présenté les résultats de notre méthodologie de partitionnement logiciel/matériel sur deux applications test :

- l'application *ICAM* : une application de traitement d'images orientée flot de données qui, une fois modélisée en SSM en introduisant un parallélisme de type pipeline dans les traitements sur les images, devenait intéressante à traiter du point de vue du partitionnement vu l'importance des GFD construits.

- l'application *MAUVE* : une application de robot mobile autonome sous-marin mixant du contrôle et du traîtement embarqué (nous avons aussi ajouté une mission de suivi sonar de contours pour augmenter la part de traitement). Les résultats de partitionnement de cette application montrent que notre méthodologie est capable de traîter des systèmes complets et de couvrir tous les *états internes* (ou *configurations*) dans lesquels ce système peut se trouver en fournissant un partitionnement des tâches actives dans ces différentes *configurations*. Les résultats montrent que l'élimination des GFD redondants permet de limiter le nombre de ces *configurations*.

Les résultats de partitionnement sur l'application *ICAM* en considérant les chemins montrent que la fonction de *Clustering* utilisée dans l'étape d'évaluation des individus n'est sans doute pas la meilleure pour constituer les contextes. Elle tend à regrouper le plus de tâches matérielles possible au sein d'un même contexte, ce qui augmente le temps de reconfiguration du contexte (cas ou l'on considère une reconfiguration partielle de l'UCR) et retarde les tâches logicielles dépendantes de tâches matérielles présentes dans le contexte considéré.

Nous avons réfléchi à une deuxième fonction de *Clustering* qui construit itérativement les contextes en regroupant les tâches en partant de la tâche la plus prioritaire et en calculant pour chaque tâche successeur le coût de l'intégration potentielle de cette nouvelle tâche dans le contexte courant. Ce coût correspond au temps d'exécution total en considérant que toutes les tâches *non-clusterisées* forment chacune un *cluster* avec tâche unique. Tant qu'il y a un regroupement avec une tâche successeur qui améliore le temps d'exécution total (sans dépasser la taille restante de l'UCR), on intègre cette tâche dans le *cluster* courant. Sinon, on ferme le cluster et on construit le prochain de la même manière.

Des résultats sont en cours d'élaboration concernant cette nouvelle méthode de *Clustering*. Elle a l'intérêt d'être plus spécifique au problème de partitionnement dans le cas d'architectures à reconfiguration partielle, mais elle présente l'inconvénient de rajouter de la complexité de calcul à l'intérieur de la boucle de l'AG. Des expérimentations sont nécessaires pour affirmer la pertinence de cette méthode.

Les résultats de partitionnement de l'application *ICAM* ont pu mettre en évidence aussi que le choix de résolution du problème de localité des données (détaillé dans le chapitre 2 de la partie 2) en imposant que chaque GFD récupère ses données à partir de la mémoire d'*ICURE* et réécrit les données en fin d'exécution, pénalise énormément le temps d'exécution total si on considère une construction des GFD par transitions.

Ceci peut être résolu par une étude sur la localité des données en mémoire. En commençant par une caractérisation des GFD critiques en temps, le partitionnement choisit ensuite la meilleure localité des données qui minimise le temps d'exécution total et fige (force) cette localité pour le reste des GFD utilisant les mêmes données.

Concernant l'application *MAUVE*, le fait que le véhicule soit autonome et qu'il change de mission (de fonctionnement) en cours de route est très intéressant du point de vue de la reconfiguration dynamique. Le véhicule change de mission en appelant la procédure SELECTNEXTTASK qui va lire la prochaine mission à exécuter, mais on peut imaginer que le véhicule change de mission (ou passe à une version améliorée ou dégradée) par rapport à la valeur de capteurs de sécurité ou de résultats de calculs intermédiaires.

Si l'on détecte par exemple un faible niveau de batterie, on passerait à une version *low_power* de la mission en cours, ou si les résultats d'estimation montrent la présence d'un fort courant marin, on passerait à une version de l'estimateur (ou du calculateur des ordres à transmettre à l'UCV : GO_TO_POINT_UCV_ORDERS) plus précise.

Il serait intéressant de changer la procédure de construction des chemins dans la machine d'états pour générer par exemple tous les chemins partant de la réception d'un signal INPUT jusqu'à l'émission d'un OUTPUT (ou d'un signal local donné). En partitionnant ces chemins, nous pouvons nous assurer que le temps mis entre la réception et l'émission de ces deux signaux ne dépasse pas une certaine contrainte de temps imposée au départ.

Les résultats de partitionnement de l'application *MAUVE* ont aussi montré que le changement de granularité d'une tâche peut permettre une meilleure utilisation des unités de traitement. Ce résultat est intéressant du fait qu'il permet d'avoir une évaluation rapide des changements effectués au niveau de la spécification. Il serait intéressant de proposer une aide au concepteur en lui proposant la granularité de ces procédures en entrée de sa spécification.

Enfin, signalons que nous avons validé le principe du schéma d'exécution sur une plateforme *Nios*-Altera avec le code **oc** reconstruit et une fonction *schedule()* qui gère le séquencement des tâches sur le processeur *Nios*. Cette validation a été effectuée sur un exemple test afin de s'assurer d'un fonctionnement correct par rapport au comportement attendu de l'exécution de l'automate *Esterel*.

Conclusion générale

OBJECTIFS

Les objectifs que nous nous sommes fixés dans cette étude sont principalement issus du projet *EPICURE* qui étaient les suivants :

- Apporter des solutions d'aide au développement d'applications basées sur une plateforme reconfigurable constituée d'un processeur connecté à une unité reconfigurable dynamiquement.
- Proposer un environnement intégrant la méthode et les outils permettant d'effectuer automatiquement le partitionnement logiciel/matériel.

BILAN DES TRAVAUX

Nous avons développé dans cette étude un flot de conception homogène qui cible des architectures reconfigurables dynamiquement. Ce flot (rappelé sur la figure 1) part d'une spécification au niveau système en SSM (formalisme graphique d'*Esterel*) intégrant des traitements de granularité importante (sous forme d'appels à des procédures C externes) dans un environnement flot de contrôle. Cette spécification permet de vérifier formellement des propriétés du système et de simuler son fonctionnement. Une fois que le fonctionnement du système est satisfaisant, on procède à l'extraction des différentes *configurations* (ou *états internes*) du système : ce sont les différentes combinaisons de contrôle et de traitement que peut effectuer le système à un instant donné de son fonctionnement. Ceci est réalisé en considérant la compilation au format automate d'*Esterel* (format **oc**).

Les traitement contenus dans une *configuration* donnée sont groupés sous forme d'un graphe de flot de données (GFD) explicitant ainsi le parallélisme potentiel entre les procédures C appelées dans cette *configuration*. La partie contrôle reste inhérente à la machine d'états.

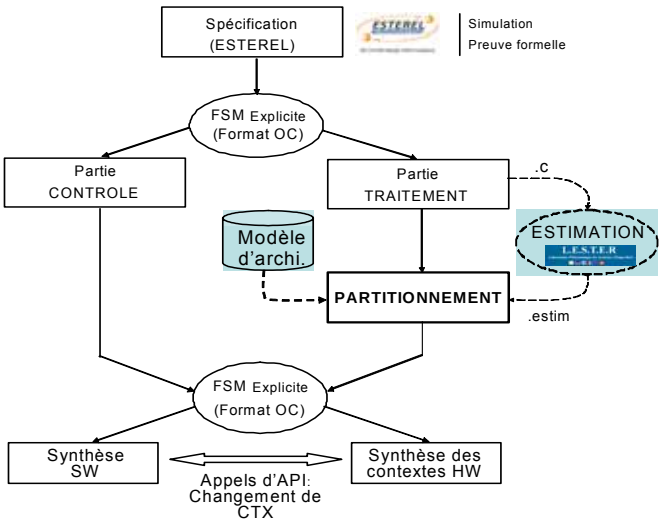


Figure 1. Flot de partitionnement développé

Après une étape d'estimation et d'exploration architecturale (étudiée au laboratoire LESTER), les procédures C d'un GFD sont estimées en terme de temps d'exécution et d'utilisation des ressources ce qui conduit à un ensemble de points d'implantation possibles pour chaque procédure (une implantation logicielle et plusieurs implantations matérielles).

L'outil de partitionnement logiciel/matériel développé dans le cadre de cette étude est basé sur un algorithme génétique. Il prend en entrée un GFD avec l'ensemble des points d'implantations des tâches qui le constituent ainsi qu'un modèle d'architecture décrit par un ensemble de paramètres pertinents qui caractérisent l'architecture. L'outil a pour objectif de minimiser le temps d'exécution total du GFD sous la contrainte d'un ensemble de ressources maximum de l'Unite de Calcul Reconfigurable (UCR). Le résultat du partitionnement est le suivant :

- une affectation (ou partitionnement spatial) des tâches du GFD sur les unités de traitement en sélectionnant pour chaque tâche un point d'implantation.
- un partitionnement temporel des tâches affectées à l'UCR en les regroupant au sein de contextes matériels qui vont se succéder sur l'UCR suivant un ordre bien défini.
- un ordonnancement des différentes tâches en tenant compte des temps de communication et des temps de reconfiguration de l'UCR.

Après l'étape de partitionnement, on reconstruit la machine d'états en y intégrant les informations issues du partitionnement pour disposer à ce niveau d'un séquenceur des *configurations* du système (la machine d'états) où chacune des *configurations* est caractérisée, partitionnée et optimisée en terme de temps d'exécution.

Les procédures destinées à une implantation matérielle nécessitent une traduction en un lan-

gage synthétisable sur l'architecture matérielle de destination. Des outils commencent à émerger pour automatiser ce processus.

La machine d'états (le format automate **oc**) est ensuite traduite en C à l'aide des outils de compilation standard d'*Esterel*. Nous avons fait le choix d'allouer son exécution au processeur qui aura pour charge le lancement des procédures qui lui sont affectées et des requêtes de reconfiguration.

Nous avons validé ce flot sur deux applications test :

- *ICAM* : une application de traitement d'images orientée flot de données.
- *MAUVE* : une application de robot mobile autonome sous-marin mixant du contrôle et du traitement embarqué.

Les résultats de partitionnement de ces deux applications montrent que notre méthodologie est capable de traiter des systèmes complets (cas de *MAUVE*) et de couvrir tous les *états internes* (ou *configurations*) dans lesquels ce système peut se trouver en fournissant un partitionnement des tâches actives dans ces différentes *configurations*. Les résultats montrent que l'élimination des GFD redondants permet de limiter le nombre de ces *configurations*.

Bien que respectant les objectifs fixés au début de cette étude, notre flot peut être optimisé de plusieurs façons. Dans la suite, nous discutons des évolutions possibles de ce flot.

PERSPECTIVES

Une des extensions à considérer dans notre flot serait d'étoffer notre fonction de coût avec des nouveaux objectifs, comme la minimisation de la consommation d'énergie ou des pics de puissance. Des réflexions à ce sujet ont été présentées à la fin de la partie 2 de ce manuscript et qui seraient intéressantes à développer.

Une autre perspective serait d'étendre cette étude au cas des architectures à reconfiguration dynamique partielle à chaud. Ces architectures ont l'avantage de minimiser l'influence des temps de reconfiguration sur le temps d'exécution total de l'application en permettant son recouvrement avec les traitements effectués sur d'autres zones de l'UCR. Ce point pose également le problème de l'ordonnancement dont nous avons montré l'importance dans la qualité des solutions obtenues. Avec des architectures à reconfiguration dynamique partielle à chaud, il de-

vient primordial de considérer une technique d'ordonnancement efficace.

Il serait également important de pouvoir étendre notre modèle pour prendre en compte une panoplie plus large d'architectures. Ceci peut être rendu possible par un travail de caractérisation préalable qui aurait pour but d'uniformiser les modèles en mettant l'accent sur les caractéristiques communes des différentes architectures (des travaux de ce type sont en cours d'étude aux laboratoires LIRMM et LESTER). Ce travail permettrait par exemple au concepteur de déterminer d'une part le type d'architecture le mieux adapté vis à vis de l'application cible mais également de la dimensionner au mieux compte tenu de contraintes multi-critères.

Annexe I

TEMPS D'EXÉCUTION LOGICIELS ET POINTS D'IMPLANTATION DE L'APPLICATION ICAM

Cette annexe détaille les données utilisées dans le partitionnement de l'application *ICAM* relativement aux caractéristiques des tâches qui composent cette application.

1. Temps d'exécution logiciels des procédures de l'application ICAM

Les temps d'exécution moyens (sur une séquence de 500 images) des procédures de l'application *ICAM* sur un processeur de type ARM 922 sont listés dans le tableau I. 1

On remarque que le temps d'exécution total sur ce processeur dépasse la contrainte de 40 ms pour traiter 25 images par seconde.

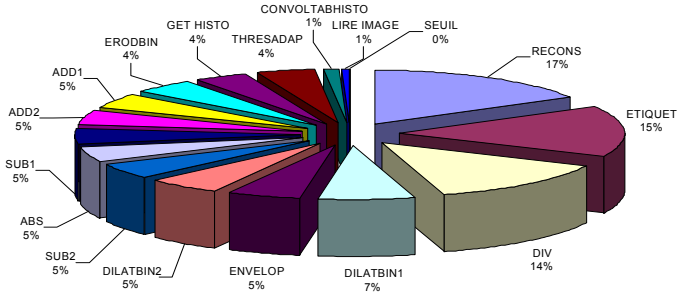


FIGURE I.1. Pourcentage des temps d'exécution SW des procédures de l'application de détection de mouvement

La contribution des ces procédures au temps d'exécution global de l'application est illustré par la figure I.1.

2. Points d'implantations de l'application ICAM utilisés par l'algorithme de partitionnement

Les figures I.2 et I.3 donnent quelques uns des points d'implantation utilisés par notre algorithme de partitionnement en terme de temps d'exécution, nombres de CL et nombres de CD. Ces valeurs sont issus des travaux effectués par le laboratoires LESTER dans le cadre du projet EPICURE sur l'outil DesignTrotter.

TABLEAU I.1 Temps d'exécution SW des procédures de l'application ICAM

Nom Procédure	$T_{exe}(\mu s)$
RECONS	12689
ETIQUET	11763
DIV	10374
ENVELOP	4086
DILATBIN1	3965
DILATBIN2	3965
SUB2	3659
ABS	3481
SUB1	3476
ADD2	3473
ADD1	3471
ERODBIN	3298
GET HISTO	3101
THRESADAPT	3074
CONVOLTABHISTO	825
LIRE IMAGE	471
SEUIL	3
TOTAL	76448

On remarque que le reconfigurable permet des accélérations de significatives par rapport au processeur sur toutes les procédures hormis le cas de ic_envelop. Par ailleurs, l'ajout de ressources de type CL et CD n'a pas toujours d'impact marqué sur les temps d'exécution.

Nom	Ic_substract	ic_decalage	ic_copyData	ic_add	ic_div	Ic_substract2	Ic_absolute	Ic_getHistogram	Ic_convolveTabHisto	Ic_histoTreshold
Temps (µs)										
Processeur	3476	8	8	3471	10374	3659	3481	3101	825	3
FPGA1	83.2704	0.1372	0.1372	83.2704	196.6502	83.2704	270.3936	83.5231	6.1449	0.03
FPGA2	83.2705			83.2705	221.2313	83.2705		92.7752	7.1857	0.00155
FPGA3	83.2708			83.2708		83.2708			7.296	
FPGA4	92.5231			92.5231		92.5231			7.4111	
FPGA5									7.4288	
FPGA6									7.5248	
Ress (CL)										
Processeur	0	0	0	0	0	0	0	0	0	0
FPGA1	104	54	54	88	316	104	316	18	976	70
FPGA2	104			88	316	104		18	968	70
FPGA3	86			70		86			670	
FPGA4	86			70		86			356	
FPGA5									348	
FPGA6									322	
Ress (CD)										
Processeur	0	0	0	0	0	0	0	0	0	0
FPGA1	12	2	2	12	8	12	6	4	32	10
FPGA2	8			8	6	8		3	31	6
FPGA3	8			8		8			30	
FPGA4	6			6		6			27	
FPGA5									26	
FPGA6									24	

FIGURE I.2. Selection de quelques points d'implantations des procédures de l'application ICAM

Nom	lc_substract	ic_decalage	ic_copyData	ic_add	ic_div	lc_substract2	lc_absolute	lc_getHistogram	lc_convolveTabHisto	lc_histoTreshold
Temps (µs)										
Processeur	3476	8	8	3471	10374	3659	3481	3101	825	3
FPGA1	83.2704	0.1372	0.1372	83.2704	196.6502	83.2704	270.3936	83.5231	6.1449	0.03
FPGA2	83.2705			83.2705	221.2313	83.2705		92.7752	7.1857	0.00155
FPGA3	83.2708			83.2708		83.2708			7.296	
FPGA4	92.5231			92.5231		92.5231			7.4111	
FPGA5									7.4288	
FPGA6									7.5248	
Ress (CL)										
Processeur	0	0	0	0	0	0	0	0	0	0
FPGA1	104	54	54	88	316	104	316	18	976	70
FPGA2	104			88	316	104		18	968	70
FPGA3	86			70		86			670	
FPGA4	86			70		86			356	
FPGA5									348	
FPGA6									322	
Ress (CD)										
Processeur	0	0	0	0	0	0	0	0	0	0
FPGA1	12	2	2	12	8	12	6	4	32	10
FPGA2	8			8	6	8		3	31	6
FPGA3	8			8		8			30	
FPGA4	6			6		6			27	
FPGA5									26	
FPGA6									24	

FIGURE I.3. Selection de quelques points d'implantations des procédures de l'application ICAM (suite)

Annexe II

MODÉLISATION EN SSM DE L'APPLICATION MAUVE

1. La modélisation de départ

La modélisation en SSM de départ réalisé par l'équipe SAM d'I3S est illustrée par la figure II.1. Elle se compose principalement d'un contrôleur de l'exécution des missions (*task_execution*), d'un contrôleur des urgences

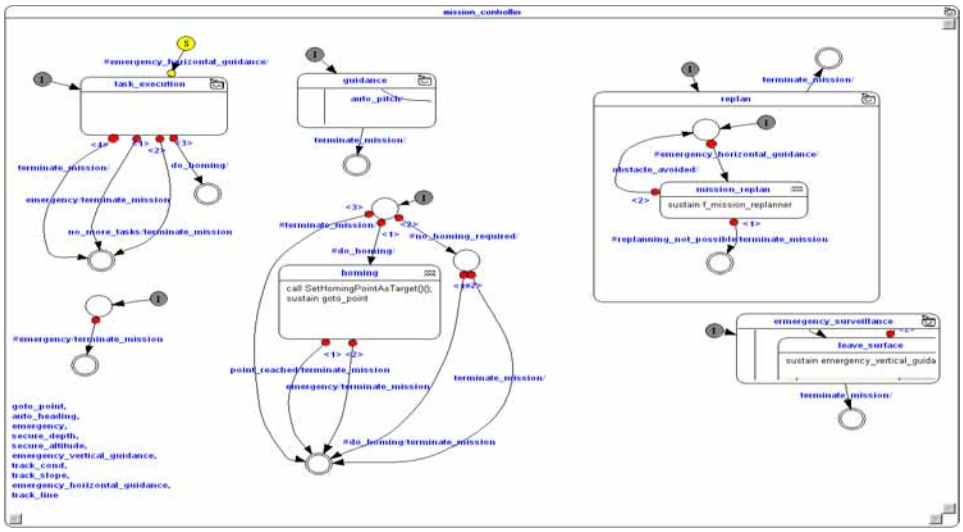


FIGURE II.1. Le contrôleur de mission de la modélisation de départ

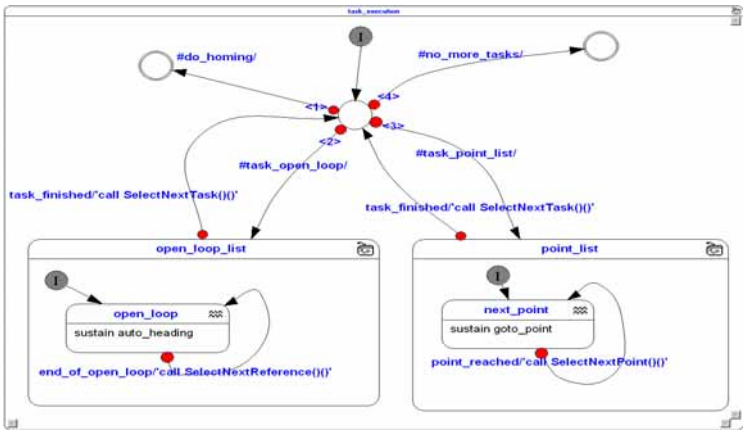


FIGURE II.3. Le module task_execution de la modélisation de départ

(*emergency_surveillance*), d'un bloc *Homing* et d'un contrôleur de guidage (*guidance*).

Le module *replan* replanifie la mission lorsqu'il détecte un obstacle (contrôleur horizontal) mais ce module n'est pas encore supporté par le véhicule qui procède en revanche à une remontée en urgence dans ce cas.

Le module *task_execution* est illustré par la figure II.3, et celui relatif a *guidance* est détaillé par la figure II.2.

Comme le montre ces figures, la modélisation de l'application fait apparaître un important parallélisme qui a pour conséquence une explosion combinatoire de l'automate après compilation *Esterel*.

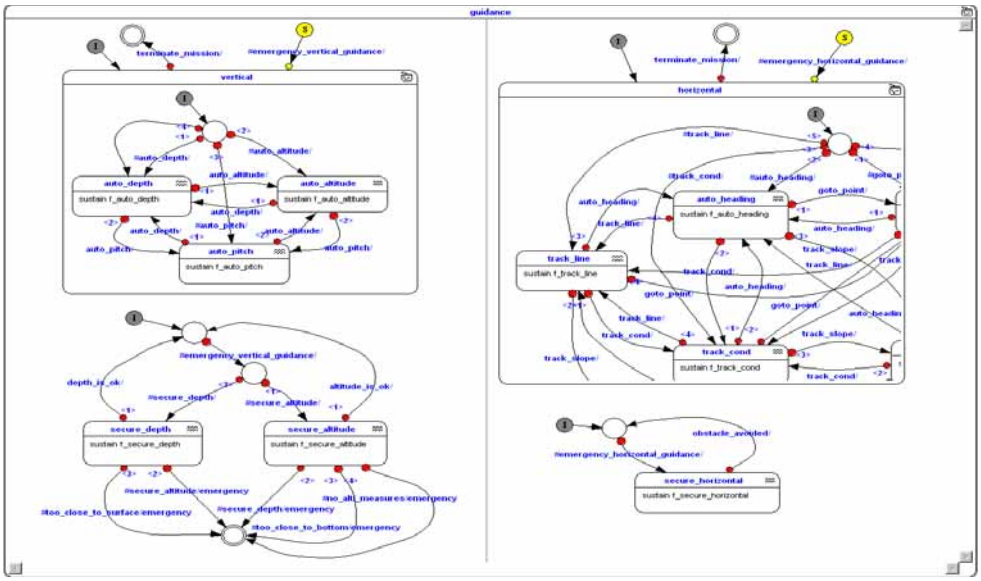


FIGURE II.2. Le module guidance de la modélisation de départ

Nous avons donc choisi une approche différente pour modéliser l'application.

2. Notre modélisation en SSM de l'application MAUVE

La modélisation que nous avons choisie de l'application *MAUVE* se compose de trois modules :

- le module de surveillance des urgences (*emergency_surveillance*).

- le module de gestion de l'exécution des tâches (*task_execution*).

- le module d'acquisition et de positionnement (*acquisition_positionnement*).

Ces trois modules (ou STG) sont toujours actifs au même instant. L'ensemble est illustré par la figure II.4.

Le module *task_execution* détaillé sur la figure II.6 regroupe les missions :

- *Sonar_tracking.*

- *Point_tracking.*

- *Open_loop.*

On peut noter que ces trois missions ne peuvent s'exécuter que de manière exclusive.

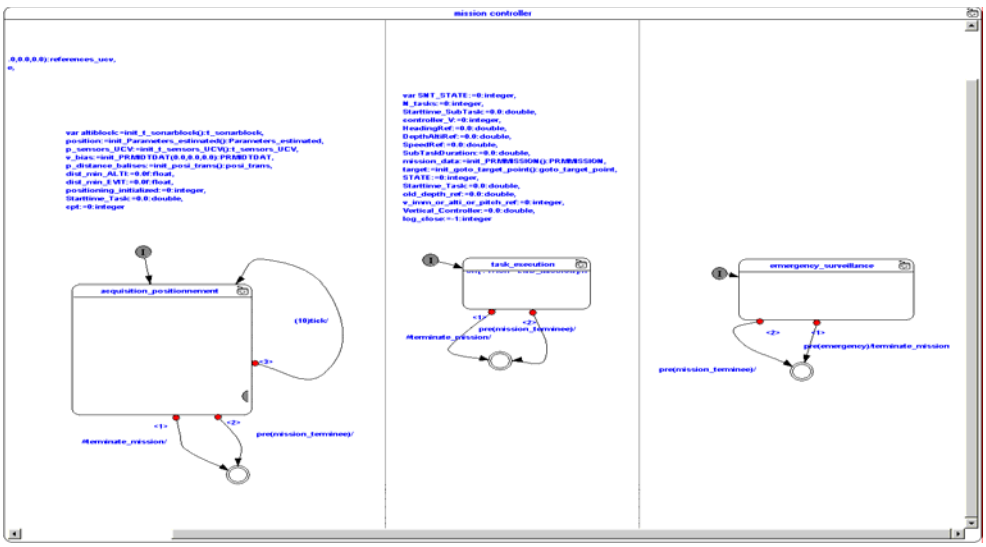


FIGURE II.4. Le contrôleur de mission de notre spécification de l'application MAUVE

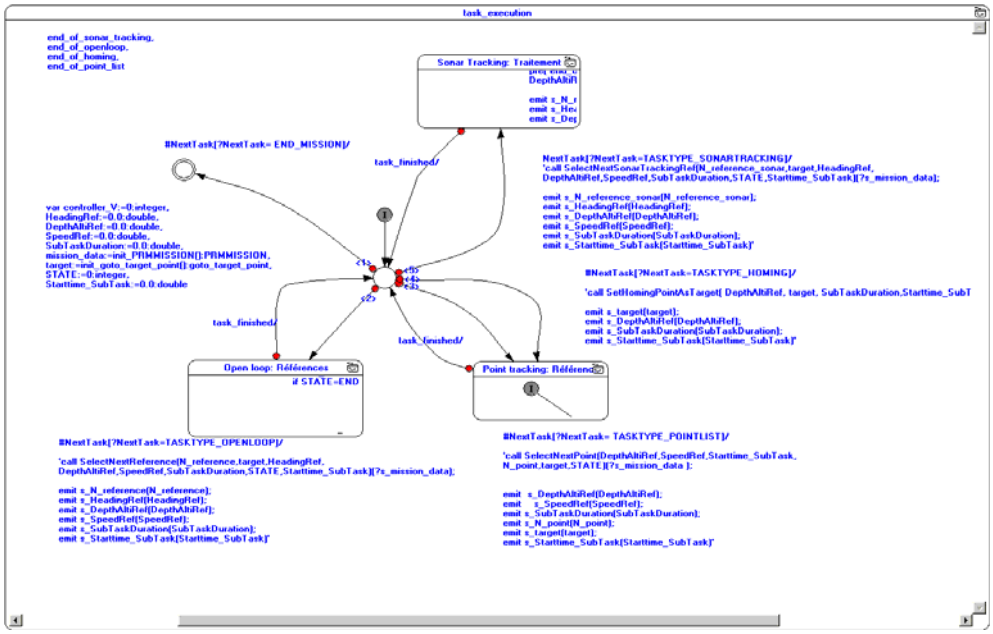


FIGURE II.6. Le module task_execution de notre modélisation

Le module relatif à la mission Sonar_tracking est illustré par la figure II.5.

Cette mission a été ajoutée dans cette étude par rapport au système *MAUVE* réel car l'équipe SAM de l'I3S envisage de l'intégrer effectivement dans ce véhicule. Ce point illustre l'intérêt de l'approche de partitionnement qui per-

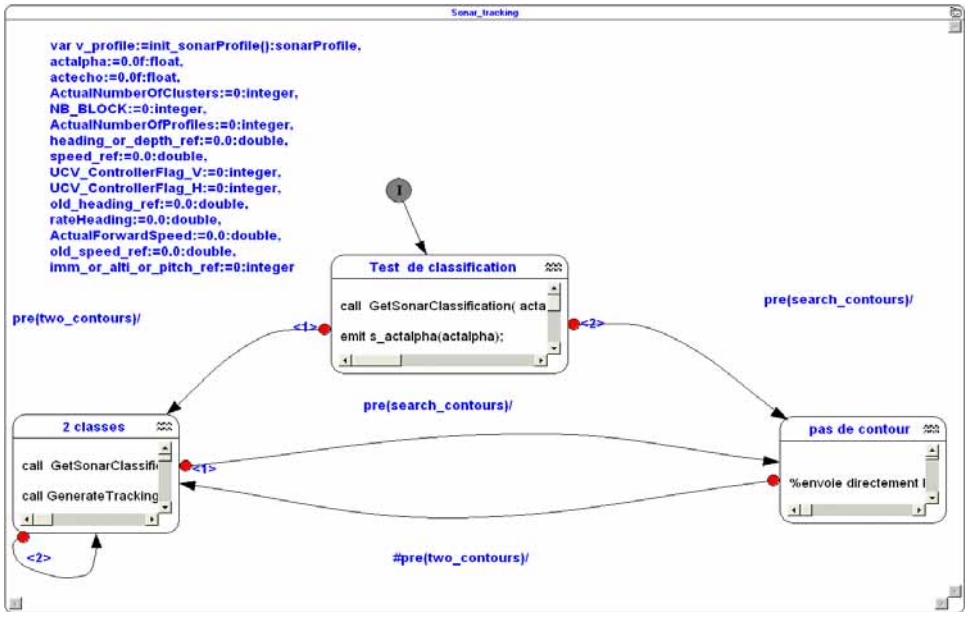


FIGURE II.5. Le module Sonar_tracking de notre modélisation

met d’évaluer une solution architecturale suivant les fonctionnalités ciblées et l’algorithmique considérée pour les réaliser.

Bibliographie

[1] E. Lee, What's ahead for embedded software, IEEE Computer, septembre 2000.

[2] D. Gajski, F. Vahid, Specification and design of embedded software/hardware systems, *IEEE Design & Test of Computers*, Vol. 12, No.1, Spring 1995.

[3] R. Ernst, J. Henkel, T. Benner, Hardware-Software Cosynthesis for Microcontrollers, *IEEE Journal Design and Test of Computers*, december, pp 64-75, 1993.

[4] R. Gupta, G. De Micheli, Hardware-Software Cosynthesis for Digital Systems, *IEEE Journal Design and Test of Computers*, september, pp 29-41, 1993.

[5] F. Vahid, Modifying Min-Cut for hardware and software functional partitioning, *Workshop on Hardware/Software Codesign*, Braunschweig, Germany, 1997.

[6] J. Teich, T. Blickle, L. Thiele, An evolutionary approach to system level synthesis, *Codes/CASHE'97*, Braunschweig, Germany, march 1997.

[7] I. Karkowski, H. Corpooral. Design space exploration algorithm for heterogeneous multi-processor embedded system design. *DAC'98*, San Francisco, CA, june 1998.

[8] P. Knudsen, J. Madsen, PACE: a dynamic programming algorithm for hardware/ software partitioning, *CODES/CASHE'97*, Pittsburgh, Pennsylvania, march 1996.

[9] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, J. Stockwood, Hardware-software co-design of embedded reconfigurable architectures, *Design Automation Conference (DAC'00)*, Los Angeles, june 2000.

[10] B. Kienhuis, E. Rijpkema, E. Deprettere, Compaan: deriving process networks from Matlab for embedded signal processing architectures, *International Conference on Hardware Software Codesign, Proceedings of the eighth international workshop on Hardware/software codesign*, San Diego, California, Pages: 13 - 17, May 3-5, 2000.

[11] P. Coste, F. Hessel, A.A. Jerraya, Multilanguage Codesign Using SDL and Matlab, *SASIMI'00*, Kyoto, Japan, april 2000.

[12] D. Gajski, N. Dutt, A. C. -H. Wu and S.Y-L Lin, High-level synthesis: Introduction to chips and systems. *Kluwer*, 1992.

[13] D. Desmet and D. Genin, Assynt: Efficient assembly code generation for digital signal processors starting from a data flowgraph, (e.d.c.-mentor graphics corp./ic group, abdijstraat 34, b-30001 leuven, belgium). *ISCAS'93*, 1993.

[14] T. Grandpierre, C. Lavarenne, Y. Sorel, Optimized rapid prototyping for real time embedded heterogeneous multiprocessors. *7th International Workshop on Hardware/ Software Co-Design* (*CODES'99)*, Rome, may 1999.

[15] R. Szymanek, K. Kunchcinski, Design space exploration in system level synthesis under memory constraints, *Euromicro Conference'99*, Milan, Italy, 1999.

[16] L. Bianco, M. Auguin, G. Gognat, A. Pegatoquet, A Path Based Partitioning Algorithm for Time Constrained Embedded Systems Design, *Int. Workshop IEEE/ ACM Codes/ CASHE'98*, Seattle, 15-18 March, 1998.

[17] P. Bjorn-Jorgensen, J. Madsen, Critical path driven cosynthesis for heterogeneous target architecures, *International Conference on Hardware Software Codesign, Proceedings of the 5th International Workshop on Hardware/Software Co-Design*, Braunschweig, Germany, March 24-26, 1997.

[18] M. C. McFarland, A. Parker and R. Camposano. The high-level synthesis of digital systems. *IEEE Proceedings*, February 1990, pp. 301-318.

[19] J.P. Diguet, G. Gogniat, P. Danielo, M. Auguin, J.L. Philippe, The SPF Model, *Proceedings of International Forum on Design Languages of (FDL'00)*, Tübingen, Germany, September 2000.

[20] Y. Le Moullec, J. P. Diguet and J. L. Philippe, Design-Trotter: a Multimedia Embedded Systems Design Space Exploration Tool, *IEEE Workshop on Multimedia Signal Processing (MMSP'02)*, St. Thomas, US Virgin Islands, December 9-11, 2002.

[21] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Proc. of the IFIP Congress 74*, North-Holland Publishing Co., 1974.

[22] E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, K.A. Vissers, YAPI: Application Modeling for Signal Processing Systems, *Design Automation Conference (DAC'00)*, Los Angeles, june 2000.

[23] Cadence Design Systems, "SPW user's manual", Foster City, CA 94404, USA.

[24] Mentor Graphics, "DSP station user's manual", San Jose, CA 95131, USA.

[25] N. Smyth, Communicating Sequential Processes Domain in Ptolemy II, *MS Report, UCB/ERL Memorandum M98/70, Dept. of EECS, University of California, Berkeley*, CA 94720, December 1998. (http://ptolemy.eecs.berkeley.edu/publications/papers/98/CSPinPtolemyII/).

[26] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8, August 1978.

[27] R. Milner, Communication and Concurrency, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[28] T. Murata, Petri Nets: Properties, analysis and applications. *Proc. IEEE*, vol.77 (4), April 1989, pp. 541-580.

[29] P. Merlin. A Study of the recoverability of computer systems. *PhD thesis, Dep. Comp. Sc.*, Univ. California, Irvine, 1974.

[30] J. Sifakis. Use of Petri nets for performance evaluation. *In: Measuring, Modelling and Evaluating Computer Systems*, ed. H. Beilmer and E. Gelenbe. North Holland, 1977.

[31] C. Ramchandani. Analysis of asynchronous concurrent systems by timed Petri Nets. *Project mac, tech. rep. 120,* Massachusetts Inst. Technol., February 1974.

[32] F. Symons. Introduction to numerical Petri Nets, a general graphical model of concurrent processing systems. *Australian Telecomm. Res.*, Vol. 14, January 1980.

[33] J. Dunagan, K. Trivedi, R. Geist and V. Nicola, Extended Stochastic Petri nets: Applications and analysis. *Proc. Performance 84*, Paris (France), December 1984, pp. 507-519.

[34] W. van der Aalst. Interval timed coloured petri nets and their analysis. Application and theory of Petri Nets 1993, *Proceedings 14th International Conference*, Chicago, Illinois, USA, vol. 691, 1993, pp. 453-472.

[35] G. Berry and G. Gonthier, The Esterel synchronous programming language: Design, semantics, implementation, *Science of Computer Programming*, 19(2) : 87-152, 1992.

[36] A. Benveniste and P. Le Guernic, Hybrid Dynamical Systems Theory and the SIGNAL Language *IEEE Tr. on Automatic Control*, Vol. 35, No. 5, pp. 525-546, May 1990.

[37] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, LUSTRE: A Declarative Language for Programming Synchronous Systems, *Conference Record of the 14th Annual ACM Symp. on Principles of Programming Languages*, Munich, Germany, January, 1987.

[38] F. Maraninchi, The Argos Language: Graphical Representation of Automata and Description of Reactive Systems, *in Proc. of the IEEE Workshop on Visual Languages*, Kobe, Japan, Oct. 1991.

[39] T. Y. Yen, W. Wolf, Sensitivity-driven cosynthesis of distributed embedded systems, *Internation Symposium on System Synthesis (ISSS'95)*, 1995.

[40] B. Dave, G. Lakshminarayana, N. K. Jha, COSYN: hardware/software co-synthesis of embedded systems, *Design Automation Conference (DAC'97)*, Anaheim, june 1997.

[41] H. Oh, S. Ha, A hardware/software co-synthesis technique based on heterogeneous multiprocessor scheduling, *Codes'99*, Rome, 1999.

[42] D. Kirovski, M. Potkonjak, System level synthesis of low power hard real time systems, *Design Automation Conference (DAC'97)*, Anaheim, june 1997.

[43] I. D. Bates, E. G. Chester, D. J. Kinniment, A statechart based HW/SW codesign system, *International Conference on Hardware Software Codesign, Proceedings of the seventh international workshop on Hardware/software codesign*, Rome, Italy, May 3-5, 1999.

[44] F. Balarin et al, Hardware-software codesign of embedded systems, the POLIS approach, *Kluwer Academic Publishers*, 1997.

[45] S. Saracco, J. R. W. Smith, and R. Reed, Telecommunications Systems Engineering Using SDL, *North-Holland - Elsevier*, 1989.

[46] N. E. Zergainoh, G. F. Marchioro, J. M. Daveau, A.A. Jerraya, Using SDL for Hardware/Software Co-design of an ATM Network Interface Card, *1st Workshop of the SDL Forum Society an SDL and MSC*, Berlin, june 1998.

[47] D.D. Gajski, F. Vahid, S. Narayan, and J. Gong. SpecSyn: An environment supporting the specify-explore-refine paradigm for hw/sw system design. *IEEE transactions on VLSI Systems*, 6(1) : 84-100, 1998.

[48] F. Thoen and F. Catthoor, Modeling, Verification and Exploration of task-level concurrency in real-time embedded systems. *Kluwer Academic Publishers*, 2000.

[49] SystemC 2.0.1 Langage reference Manual, Revision 1.0, http://www.systemc.org/projects/systemc/documents/SystemC_v201_LRM/

[50] SystemVerilog 3.1 - Accellera's Extensions to Verilog, http://www.eda.org/sv/SystemVerilog_3.1_final.pdf

[51] W. Rosenstiel, S. Swan, F. Ghenassia, P. Flake, and J. Srouji, SystemC and SytemVerilog: Where do they fit? Where are they going?, *Panel at the Design Automation and Test in Europe Conference (DATE'04)*, Paris, France, 2004.

[52] J. Rumbaugh, I. Jacobson, and G. Booch, The Unified Modeling Language User Guide, Addison-Wesley, 1998.

[53] G. Martin, L. Lavagno, J. Louis -Guerin, Embedded UML: a merger of real-time UML and co-design, *Proceedings of CODES 2001,* Copenhagen, pp.23-28, April 2001.

[54] P. Green, M. Edwards, S. Essa, UML for System-Level Design, Forum on Design Languages, *Proceedings of FDL 2001,* Lyon, France, Sept. 3-7, 2001.

[55] R. Chen, M. Sgroi, G. Martin, L. Lavagno, A. Sangiovanni- Vincentelli, J. Rabaey, Embedded System Design Using UML and Platforms, *Proceedings of FDL'02*, Marseille, France, September 2002.

[56] G. de Jong, A UML-Based Design Methodology for Real-Time and Embedded Systems, *In Proceedings of the Design Automation and Test in Europe Conference (DATE'02)*, Paris, France, 2002.

[57] M.R. Mousavi, P. Le Guernic, J.-P., Talpin, S.K. Shukla, T. Basten, Modeling and Validation of Globally Asynchronous Design in Synchronous Frameworks, *Proceedings of the Conference on Design Automation and Test in Europe (DATE'04)*, Paris, France, pp. 384--389, February 2004.

[58] V. D'Silva and S. Ramesh, Synchronous protocol automata for modelling and verification of system-on-chip bus architectures, *Proceedings of the Conference on Design Automation and Test in Europe (DATE'04)*, Paris, France, pp. 384--389, February 2004.

[59] S. Lee, S. Yoo, and K. Choi, Reconfigurable SoC Design with Hierarchical FSM and Synchronous Dataflow Model, *Tenth International Workshop on Hardware/Software Codesign,* Estes Park, Colorado, May 6-8, 2002.

[60] The PTOLEMY project : http://ptolemy.eecs.berkeley.edu

[61] M. Auguin, L. Bianco, L. Cappella, E. Gresset, Partitioning conditional data flow graphs for embedded system design, *International Conference on Application Specific Systems, Architectures and Processors (ASAP'2000)*, Boston, july 2000.

[62] A. Kalavade, P. Subrahmanyam, Hardware/software partitioning for multi-function systems, *ICCAD'97*, San José, novembre, 1997.

[63] P. Schaumont, S. Vernalde, L. Rijnders, M. Engels, I. Bolsens, A programming environment for the design of complex high speed ASICs, *Design Automation Conference (DAC'98)*, San Francisco, 1998.

[64] PTOLEMY II, Heterogeneous Concurrent Modeling And Design In Java, Volume 1: Introduction To Ptolemy II, *University of California at Berkeley*, July 16, 2003 http://ptolemy.eecs.berkeley.edu

[65] http://www.gigascale.org/metropolis/

[66] www.sldl.org

[67] http://www.eda.org/slds-rosetta/

[68] Xilinx, Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet, *Product Specification*, June 2004.

[69] Xilinx, Virtex-4 Family Overview, *Advanced Product Sprcification, Data Sheet DS 112 (v1.1)*, September 10, 2004.

[70] Altera, Excalibur Device Overview, *Data Sheet*, May 2002.

[71] R. Hartenstein, A Decade of Reconfigurable Computing: A Visionary Perspective, *Proc. DATE '01*, Munchen, March 13-16, 2001.

[72] P. Schaumont I. Verbauwhede K. Keutzer M. Sarrafzadeh, A Quick Safari Through the Reconfiguration Jungle, *Design Automation Conference (DAC'01)*, Las Vegas, June 2001.

[73] S. Hauck, The Roles of FPGAs in Reprogrammable Systems, *Proceedings of the IEEE*, Vol. 86, No. 4, pp. 615-639, April 1998.

[74] J-Y. Mignolet, S. Vernalde, D. Verkest, R. Lauwereins, Enabling hardware-software multitasking on a reconfigurable computing platform for networked portable multimedia appliances, *Proceedings of the International Conference on Engineering Reconfigurable Systems and Architecture (ERSA'02)*, pages 116-122, Las Vegas, June 2002.

[75] F. Ghaffari, M. Benjemaa, M. Auguin. Algorithms for the Partitioning of Applications containing variable duration tasks on reconfigurable architectures. *IEEE Int. Conf. AICCSA'03*, Tunis, Tunisia, 14 – 18 July 2003.

[76] S. Trimberger, Scheduling Designs into a Time-Multiplexed FPGA, *FPGA'98 Proceedings*, Monterey, CA, February 22-24, 1998, p. 153-160.

[77] T. Fujii et al., A Dynamically Reconfigurable Logic Engine with a Multi-Context/ Multi-Mode Unified-Cell Architecture", *Proc. of the IEEE International Solid State Circuits Conference (ISSCC'99)*, San Francisco, CA, February 15-17, 1999. See : http://www.nec.co.jp/press/en/9902/1502.html.

[78] J. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. *In IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, April 1997.

[79] H. Singh, M. H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, E. M. C. Filho, MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation – Intensive Applications," *IEEE Trans. Computers*, vol. 49, No. 5, pp. 465-481, May 2000.

[80] Chameleon Systems. http://www.chameleonsystems.com/

[81] S. Hauck, Configuration Prefetch for single context reconfigurable coprocessors, *ACM Int. Symposium on FPGA*, 1998.

[82] S. Brown and J. Rose. FPGA and CPLD Architectures: A Tutorial. *IEEE Design & Test of Computers*, Summer 1996.

[83] Altera. Stratix FPGA Family. Device Handbook, volume 1, January 2004.

[84] Xilinx. XC6200 Field Programmable Gate Arrays, 1996.

[85] Atmel, AT40K FPGAs with FreeRAM, january 1999.

[86] Atmel, AT6000 Series Configuration, Configuration Guide, September 1999.

[87] Atmel, Implementing Cache Logic with FPGAs, Application Note, September 1999.

[88] D. Demigny, M. Paindavoine, and S. Weber. Architecture à reconfiguration dynamique pour le traitement temps réel des images. *Technique et Science de l'Information Numéro Spécial Architectures Reconfigurables*, 18(10) : 1087-1112, December 1999.

[89] Xilinx Inc.(www.xilinx.com). Virtex Series FPGAs.

[90] Xilinx Inc.(www.xilinx.com). Two flows for partial reconfiguration: module based or difference based. *Application Note XAPP290: Virtex, Virtex E, Virtex-II, Virtex-II Pro Families*, November 2003.

[91] Edson L. Horta, John W. Lockwood, David E. Taylor, David Parlour, Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration. *Design Automation Conference (DAC'02)*, New Orleans, LA, June 10-14, 2002.

[92] E. Tau, D. Chen, I. Eslick, J. Brown and A. DeHon, A First Generation DPGA Implementation, *FPD'95*, *Canadian Workshop of Field-Programmable Devices*, May 1995.

[93] H. Singh, M. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. Filho, MorphoSys, An Integrated Reconfigurable System for Data-Parallel Computation-Intensive Applications, *IEEE Transactions on Computers* 2000.

[94] G. Venkataramani, W. Najjar, F. Kurdahi, N. Bagherzadeh and W. Bohm, A Compiler Framework for Mapping Applications to a Coarse-grained Reconfigurable Architecture, *Proceedings of International Conference on Compiler, Architecture and Synthesis for Embedded Systems (CASES'01)*, Atlanta GA, November 2001.

[95] A. Abnous and J. Rabaey, Ultra-Low-Power Domain-Specific Multimedia Processors, *IEEE VLSI Signal Processing Workshop*, October 1996.

[96] H. Zhang, V. Prabhu, V. George, M. Wan, M. Benes, A. Abnous, and J. M. Rabaey, A 1-V Heterogeneous Reconfigurable DSP IC for Wireless Baseband Digital Signal Processing, *IEEE JOURNAL OF SOLID-STATE CIRCUITS*, VOL. 35, NO. 11, November 2000.

[97] M. Wan, H. Zhang, V. George, M. Benes, A. Abnous, V. Prabhu, and J. M. Rabaey, Design Methodology of a Low-Energy Reconfigurable Single-Chip DSP System, *JVLSISP'00*, 2000.

[98] A. Kalavade, System Level Codesign of Mixed Hardware-Software System, Tech Report UCB/ERL 95/88, *Ph.D. Dissertation, Department of EECS, University of California, Berkeley*, CA 94720, September 1995.

[99] R. Subrammanian, U. Jha, J. Medlock, C. Woodthorpe, K. Rieken, Novel Application-Specific Signal Processing Architectures for Wideband CDMA and TDMA applications, *IEEE Vehicular Technology Conference*, Tokyo, Japan, May 2000.

[100] J. Wawrzynek and T. J. Callahan, Instruction-Level Parallelism for reconfigurable computing, *8th International Workshop on Field-Programmable Logic and Applications*, Berlin, 1998.

[101] J. Liang, S. Swaminathan, R. Tessier. aSoC : A scalable, Single-Chip Communications Architecture. *IEEE Conference on Parallel Architectures and Compilation Technique, Philadelphia, USA*, October 2000.

[102] A. Laffely, J. Liang, P. Jain, N. Weng, W. Burleson, R. Tessier. Adaptive System on a Chip (aSoC) for Low-Power Signal Processing. *Asilomar Conference on Signals, Systems and Computers*, November 2001.

[103] L. Bossuet, W. Burleson, G. Gogniat, V. Anand, A. Laffely, J.L. Philippe, Targeting Tiled Architectures in Design Exploration, *In 10th Reconfigurable Architectures Workshop, (RAW '03)*, Nice, France, April 22, 2003.

[104] R. David, D. Chillet, S. Pillement, and O. Sentieys. DART: A Dynamically Reconfigurable Architecture dealing with Next Generation Telecommunications Constraints. *In 9th Reconfigurable Architecture Workshop (RAW'02)*, Fort Lauderale, USA, April 2002.

[105] R. David, D. Chillet, S. Pillement, and O. Sentieys. A compilation framework for a dynamically reconfigurable arcitecture. In *12th International Conference on Field Programmable Logic and application (FPL'02)*, La grande Motte, France, September 2002.

[106] S. Chevobbe, N. Ventroux, F. Blanc, and T. Collette, RAMPASS: Reconfigurable And Advanced Multi-Processing Architecture for future Silicon System. *in Proceedings of Samos III Workshop*, July, 2003.

[107] G. Sassatelli, L. Torres, J. Galy, G. Combon, and C. Diou. The systolic ring: A dynamically reconfigurable architecture for embedded systems. In *International Workshop on Field Programmable Logic and Applications (FPL'01)*, pages 409-419. Lecture Notes in Computer Science 2147, 2001.

[108] S. Prakash, and A. Parker, Synthesis of application-specific multiprocessor architectures, *in Proceedings of the Design Automation Conference (DAC'91)*, pp. 8-13, 1991.

[109] M. Auguin, L. Capella, F. Cuesta, and E. Gresset, CODEF: a system level exploration tool, *ICASSP*, Salt Lake City, 7-11 mai 2001.

[110] K. M. Gajjalapurna, and D. Bhatia, Partitioning in Time: A Paradigm for Reconfigurable Computing, *In Proceedings of the IEEE Int. Conference on Computer Design (ICCD'98)*, Austin, Texas, pp. 340-345, October 5-7, 1998.

[111] S. Tsasakou, C. Dre, H. Kharatanasis, A. Birbas, Combined assessment of an industrial current practice and CoWare's methodologu to the codesign/cosimulation problem, *MEDEA/ESPRIT conference on HW/SW codesign*, 1998.

[112] M. Auguin, K. Ben Chehida, J. P. Diguet, X. Fornari, A. M. Fouilliart, C. Gamrat, G. Gogniat, P. Kajfasz, and Y. Le Moullec, "Partitioning and CoDesign tools & methodology for Reconfigurable Computing: the EPICURE philosophy", *Proceedings of the Third International Workshop on Systems, Architectures, Modeling Simulation (SAMOS'03)*, July 2003, Samos, Greece.

[113] J. M. P. Cardoso, and H. C. Neto, An Enhanced Static-List Scheduling Algorithm for Temporal Partitioning onto RPUs, *In Proc. of the IFIP International Conference on Very Large Scale Integration (VLSI'99)*, Lisbon, 1999.

[114] H. Liu, D. F. Wong, Network flow based circuit partitioning for time-multiplexed FPGAs, *Proceedings of the IEEE/ACM international conference on Computer-aided design (ICCAD'98)*, San Jose, California, United States, 1998.

[115] H. Liu, D. F. Wong, Circuit partitioning for dynamically reconfigurable FPGAs, *Proceedings of the ACM/SIGDA seventh international symposium on Field programmable gate arrays*, February 1999.

[116] Z. Huang, and S. Malik, Exploiting Operation Level Parallelism through Dynamically Reconfigurable Datapaths, *In Proceedings of the Design Automation Conference (DAC'02)*, New Orlean, June 2002.

[117] B. Dave, CRUSADE: Hardware/Software Co-Synthesis of Dynamically Reconfigurable Heterogeneous Real-Time Distributed Embedded Systems, *In Proceedings of the Design Automation and Test in Europe Conference (DATE'99)*, Munich, Germany, March 1999.

[118] B.P. Dave and N.K. Jha, CASPER: Concurrent Hardware-Software Co-Synthesis of Hard Real-Time Aperiodic and Periodic Specifications of Embedded System Archi-

tectures, *In Proceedings of the Design Automation and Test in Europe Conference (DATE'98)*, Paris, France, February 1998.

[119] R. P. Dick and N. K. Jha, MOGAC: a multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems, *in Proceedings of the 1997 IEEE/ ACM International Conference on Computer-Aided Design (ICCAD'97)*, November 1997.

[120] R.P. Dick and N.K. Jha, MOCSYN: Multiobjective Core-Based Single-Chip System Synthesis, *in Proc. Design, Automation and Test in Europe (DATE'99)*, March 1999.

[121] T. Blickle, J. Teich and L. Thiele, System-level synthesis using evolutionary algorithms. *TIK Report-Nr. 16*, 1996.

[122] P. Eles, Z. Peng, K. Kuchcinski, A. Doboli, System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search, *In Proc. Design Automation for Embedded Systems*, 1997.

[123] M. Lòpez-Vallejo and J. C. López, On the hardware-software partitioning problem: System modeling and partitioning techniques, *ACM Transactions on Design Automation of Electronic Systems (TODAES'03)* , Vol. 8 , No. 3, pp: 269 - 297, July 2003.

[124] M. Kaul, R. Vemuri, Optimal Temporal Partitioning and Synthesis for Reconfigurable Architectures, *In Proceedings of the Design Automation and Test in Europe Conference (DATE'98)*, Paris, France, February 1998.

[125] V. Srinivasan, S. Radhakrishnan, R. Vemuri, Hardware Software Partitioning with Integrated Hardware Design Space Exploration, *In Proceedings of the Design Automation and Test in Europe Conference (DATE'98)*, Paris, France, February 1998.

[126] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, and R. Vemuri, An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures, *Reconfigurable Architectures Workshop (RAW'98)*, pp. 31-36, March 1998.

[127] K. S. Chatha and R. Vemuri, Hardware-Software Codesign for Dynamically Reconfigurable Architectures, *9th International Workshop on Field Programmable Logic and Applications (FPL'99)*, Glasgow, Springer Verlag, September, 1999.

[128] M. Kaul, R. Vemuri, S. Govindarajan, and I. E. Ouaiss, An Automated Temporal Partitioning and Loop Fission approach for FPGA based Reconfigurable Synthesis of DSP Applications, *In Design Automation Conference (DAC'99)*, New Orleans, LA, June 1999.

[129] S. Govindarajan, R. Vemuri, Tightly Integrated Design Space Exploration with Spatial and Temporal Partitioning in SPARCS, *10th International Conference on Field Programmable Logic and Applications (FPL'00)*, Villach, Austria, 2000.

[130] S. Ganesan and R. Vemuri, An Integrated Temporal Partitioning and Partial Reconfiguration Technique for Design Latency Improvement, *In Proceedings of the Design Automation and Test in Europe Conference (DATE'00)*, Paris, France, March 2000.

[131] K. S. Chatha and R. Vemuri, MAGELLAN: Multiway Hardware-Software Partitioning and Scheduling for Latency Minimization of Hierarchical Control-Dataflow

Task Graphs, *in Proceedings of 9th International Symposium on Hardware-Software Codesign (CODES'01)*, Copenhagen, Denmark, April, 2001.

[132] J. Noguera, R. Badia, Run-Time HW/SW Codesign for Discrete Event Systems using Dynamically Reconfigurable Architectures, *13th International Symposium on System Synthesis (ISSS'00)*, Madrid, Spain, September, 2000.

[133] B. Mei, P. Schaumont and S. Vernalde, Hardware-Software Partitioning and Scheduling Algorithm for Dynamically Reconfigurable Embedded Systems, *11th ProRISC workshop on Circuits, Systems and Signal Processing*, Veldhoven, Netherlands, Nov. 2000.

[134] F. Vahid, Partitioning Sequential Programs for CAD Using a Three Step Approach, *ACM Transactions on Design Automation of Electronic Systems (TODAES'03)*, Vol. 7, No. 3, pp. 413-429, July 2003.

[135] M. L. Vallejo, J. Grajal, J. C. Lopez, Constraint-driven System Partitionning, *In Proceedings of DATE'00*, Paris, March 2000.

[136] M. L. Vallejo and J. C. Lopez, On the Hardware-Software Partitioning Problem: System Modeling and Partitioning Techniques, *In ACM Transactions on Design Automation of Electronic Systems (TODAES'03)*, Vol. 8, No. 3, pp. 269-297, July 2003.

[137] B. Miramond, Méthodes d'optimisation pour le partitionnement logiciel/matériel de systèmes à description multi-modèles, *Ph.D. Thesis*, December 2003.

[138] R. Maestre, J. Kurahi, N. Bagherzadeh, H. Singh, R. Hermida, M. Fernandez, Kernel Scheduling in reconfigurable scheduling, *In Proceedings of the Design Automation and Test in Europe Conference (DATE'99)*, Munich, March, 1999.

[139] D. N. Rakhmatov , B. K. Vrudhula, Hardware-Software Bipartitionning for dynamically reconfigurable systems, *Tenth International Workshop on Hardware/Software Codesign (CODES'02)*, Estes Park, Colorado, May 6-8, 2002.

[140] M. Vasilko, DYNASTY: A Temporal Floorplanning Based CAD Framework for Dynamically Reconfigurable Logic Systems, *In Patrick Lysaght, James Irvine, and Reiner W. Hartenstein, editors, Field-Programmable Logic and Applications*, pages 124-133. Springer-Verlag, Berlin, August/September 1999.

[141] J. Noguera, R. M. Badia, Dynamic Run-Time HW/SW Scheduling Techniques for Reconfigurable Architectures, *Tenth International Workshop on Hardware/Software Codesign (CODES'02)*, Estes Park, Colorado, May 6-8, 2002.

[142] G. Stitt, R. Lysecky, and F. Vahid, Dynamic Hardware/Software Partitioning: A First Approach, *In Proc. Design Automation Conference (DAC'02)*, Anaheim, CA, June, 2003.

[143] J. Mignolet, V. Nollet, P. Coene, S. Vernalde, D. Verkest, and R. Lauwereins, Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-On-Chip, *In Proceedings of the Design Automation and Test in Europe Conference (DATE'03)*, Munich, Germany, March 2003.

[144] R. Bergamaschi, J. Cohn (embedded tutorial): The A to Z of SoCs, *Proceedings of the IEEE/ACM International Conference on Computer-aided Design (ICCAD'02)*, San Jose, California, USA, November 10-14, 2002.

[145] C. Andre, Representation and analysis of reactive behaviors: A synchronous approach, *in Proc. CESA'96*, Lille, France, July 1996.

[146] C. Andre, Semantics of S.S.M, *Esterel-Technologies/I3S, rapport final*, April, 2003. http://www.esterel-technologies.com, section Downloads/Scientific Papers.

[147] C. Andre, H. Boufaied, S. Dissoubray, SyncCharts : un modèle graphique synchrone pour systèmes réactifs complexes, *Proceedings of the International Conference on Real Time Systems (RTS'98)*, Paris, January 1998.

[148] G. Berry, The foundations of Esterel, *Esterel-Technologies/Ecole des Mines de Paris/ INRIA*, http://www.esterel-technologies.com, section Downloads/Scientific Papers.

[149] N. Messina, La conception de systèmes complexes et critiques, *Conférencier invité (Texas Instruments) des rencontres SympAAA'03*, La Colle sur Loup, octobre 2003.

[150] G. Berry, M. Kishinevsky, S. Singh, System Level Design and Verification Using a Synchronous Langage, *Tutorial of the IEEE/ACM international conference on Computer-aided design (ICCAD'03)*, San Jose, CA, November 2003.

[151] The LUSTRE-ESTEREL portable format Version oc5, January 17, 2001.

[152] P. V. Knudsen and J. Madsen, Integrating Communication Protocol Selection with Partitioning in Hardware/Software Codesign, *In Proceedings of the 11th ISSS*, pages 111 -- 116, 1998.

[153] J. Holland, Adaptation in natural and artificial systems, Ann Arbor : *University of Michigan Press*, 1975.

[154] P. Guitton-Ouhamou, C. Belleudy, M. Auguin, A Data path consumption model for the Palm DSPTM, IASTED Power Energy Systems 2001, Tampa, Florida, USA, November 2001.

[155] Guitton-Ouhamou Patricia, Belleudy Cécile, Auguin Michel : 'Power consumption Model for the DSP OAK Processor', proceedings of Very Large Scale Integration-System-On Chip 2001, pp. 73-78, December 3-5 2001, Montpellier, France.

[156] Johann Laurent, Nathalie Julien, Eric Senn, Eric Martin, 'Estimation de la consommation d'un algorithme C par analyse fonctionnelle', 3ème Colloque de CAO de Circuits et Systèmes Intégrés, 15, 16-17 Mai 2002, Paris.

[157] Evaluating Power for Altera Devices, Application note. Altera Corporation, July 2001.

[158] H. Ben Fradj, Estimation et Optimisation de la consommation des accès mémoires dans un système embarqué, *rapport de DEA SiCom, Université de Nice Sophia-Antipolis*, 2003.

[159] Mini Autonomous Underwater VEhicule, Rapport technique, première partie, EURO PROJECT MAS-CT95-0036, E.C / D.G.12-MAST III, *Sciences and Marine Technology*, 1995.

[160] N. Benech, J. fuchet and Y. Cellè, Présentation du principe de fonctionnement du logiciel Genetic, Rapport interne, I3S, Octobre 2002.

Publications personnelles

Revues nationales

[161] K. Ben Chehida, M. Auguin et S. Raimbault : Partitionnement Logiciel Matériel ciblant une architecture reconfigurable dynamiquement, *RSTI Architecture des ordinateurs, Série Techniques et Sciences Informatiques*, Volume 22, n° 6/2003.

Conférences internationales

[162] K. Ben Chehida, M. Auguin et S. Raimbault : Architecture and application partitioning for reconfigurable system design, *11th European Signal Processing Conference EUSIPCO*, 3 - 6 September 2002, Toulouse.

[163] M. Auguin, K. Ben Chehida. HW/SW Partitioning Approach For Reconfigurable System Design, *International conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES'02*, Grenoble, 8 - 11 October 2002.

[164] K. Ben Chehida, P. Guitton-Ouhamou, C. Belleudy and M. Auguin. A multiobjective hardware software partitioner for dynamically reconfigurable system design. *WSEAS International Conference on Hardware / Software Codesign*. Rio de Janeiro 15 - 17 October 2002.

[165] K. Ben Chehida, M. Auguin : Design space exploration approach for reconfigurable platforms, *Co-design in Embedded Real-time Systems CERTS'03, workshop of the EUROMICRO Conference on Real-Time Systems*, Porto, 1 - 4 july 2003.

[166] M. Auguin, K. Ben Chehida, J. P. Diguet, X. Fornari, A. M. Fouilliart, C. Gamrat, G. Gogniat, P. Kajfasz, and Y. Le Moullec, Partitioning and CoDesign tools & methodology for Reconfigurable Computing: the EPICURE philosophy, *Proceedings of the Third International Workshop on Systems, Architectures, Modeling Simulation (SAMOS'03)*, July 2003, Samos, Greece.

[167] K. Ben Chehida, M. Auguin : Partitioning reactive Data Flow Applications On Dynamically reconfigurable Systems, I*FIP Conference on Very Large Scale Integration of System-on-Chip VLSI-SOC'03* - Darmstadt, Germany 1 - 3 December 2003.

Conférences nationales

[168] M. Auguin, K. Ben Chehida, S. Raimbault, Approche de partitionnement logiciel matériel ciblant une architecture reconfigurable, *8ème Symposium en Architecture Sympa8*, Hammamet, 10 - 13 avril 2002.

[169] P. Guitton-Ouhamou, K. Ben Chehida, C. Belleudy and M. Auguin. Exploration architecturale d'une application du type flot de données par partitionnement et estimation de la consommation, *Journées francophones de l'adéquation algorithme architecture - JFAAA 2002*, Monastir, Tunisie, 16 - 18 Décembre 2002.

[170] K. Ben Chehida, M. Auguin : Partitionnement pour la conception de systèmes réactifs à flot de données sur architectures reconfigurables, *Symposium en Architecture et Adéquation Algorithme Architecture SympAAA'03* - La Colle Sur Loup 15 - 17 Octobre 2003.

Rapports de recherche internes

[171] K. Ben Chehida, Méthodologie de Partitionnement Logiciel/Matériel pour une architecture reconfigurable dynamiquement: Etat d'avancement et perspectives, Rapport interne, I3S, Juillet 2003.

[172] K. Ben Chehida, Partitionnement Logiciel/Matériel pour des architectures reconfigurables utilisant une approche Génétique, Mémoire de DEA, Université de Nice-Sophia Antipolis, I3S, Juillet 2001.