Partitionnement en ligne d'applications flots de données pour des architectures temps réel auto-adaptatives

Fakhreddine Ghaffari

UNIVERSITE DE NICE-SOPHIA ANTIPOLIS

ECOLE DOCTORALE STIC

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

T H E S E

pour obtenir le titre de

Docteur en Sciences

de l'Université de Nice-Sophia Antipolis

Mention : Electronique

Présentée et soutenu par :

AUTEUR : ***Fakhreddine GHAFFARI***

Partitionnement en ligne d'applications flots de données pour des architectures temps réel auto-adaptatives

---

Thèse dirigée par : ***Michel AUGUIN et Mohamed ABID***

Soutenue le : 30 Novembre 2006

Jury :

M. Pascal SAINRAT, Professeur, IRIT Université Paul Sabatier	**Examinateur**
M. Jean Luc PHILIPPE, Professeur, Université de Bretagne Sud	**Rapporteur**
M. Habib YOUSSEF, Professeur, ISITC Université de sousse Tunisie	**Rapporteur**
M. Michel AUGUIN, Directeur de recherche CNRS, I3S, UNSA	**Directeur de thèse**
M. Mohamed ABID, Professeur, ENIS, Université de Sfax,	**Directeur de thèse**

Je dédie ce travail à :
Mes parents: Bechir et Moghlia
Ma femme: Aicha
Mon fils: Mohammed
Ma sœur: Yemna

Remerciements

Ce travail a été effectué en cotutelle entre l'équipe MOSARTS (Modélisation et Synthèse d'Architectures de Traitement de Signal) du laboratoire I3S à Sophia Antipolis et le laboratoire CES (Computer, Electronics & Smart engineering systems design) à l'Ecole Nationale d'Ingénieurs de Sfax en Tunisie.

Je remercie en tout premier lieu mes directeurs de thèse M. Michel AUGUIN et M. Mohamed ABID, pour la confiance qu'ils m'ont témoignée tout au long de ces années de travail, leurs disponibilités, leurs encouragements quotidiens et leurs précieux conseils.

Mes remerciements vont également à M. Maher BEN JEMAA qui a suivi de près ce travail.

Un grand merci aux professeurs M. Jean Luc PHILIPPE et M. Habib YOUSSEF pour l'intérêt qu'ils ont porté à ce travail et d'avoir accepté d'être les rapporteurs de ma thèse.

Tous mes remerciements à M. Pascal SAINRAT pour avoir accepté de présider ce jury.

Je tiens à remercier chaleureusement tous les membres du groupe MOSARTS pour leurs aides précieuses et pour la bonne ambiance qu'ils sont parvenus à instaurer au sein de l'équipe.

Je tiens à remercier également les membres du laboratoire CES avec qui j'ai eu beaucoup de plaisir à collaborer durant plusieurs mois.

Un grand merci à Khalil Moussa, sa femme Habiba et ses enfants Mohammed et Fakher. Merci pour votre soutien moral et pour vos encouragements.

Mes remerciements les plus chaleureux vont aux membres de ma famille, Béchir, Moghlia, Yemna, Haj Mohammed, Fatma,

Mabrouka, Souad, Ali, Kheria, Mohamed, Imed, Noura, Kamel, Lazhar, Khaled, Besma et Ibtisem, qui m'ont soutenu durant toutes mes études et ont toujours su être présents à coté de moi.

Finalement, un merci particulier à ma femme aicha, qui a eu la lourde tâche de me soutenir durant la dernière année de mon travail, sa patience et ses encouragements m'ont largement aidé à mener à bien cette thèse.

Fakhreddine GHAFFARI
Nice le 18 novembre 2006

Table des Matières

Introduction

Le marché des systèmes embarqués est étroitement lié au marché de l'industrie électronique, dont l'évolution et la progression dessinent les tendances de l'embarqué. Avec 1069 milliards d'euros en 2005, l'industrie de l'électronique a dépassé son précédent record (plus de 1000 milliards d'euros en 2000). La crise des télécommunications après l'éclatement de la bulle Internet appartient désormais au passé : l'évolution sur les 5 dernières années est marquée par le renforcement de marchés tels que la téléphonie mobile, les télécommunications et l'électronique automobile (Source: Cabinet DECISION)[1].

Les perspectives à moyen terme de l'industrie électronique sont plus intéressantes encore. Selon la même étude, le marché pourrait progresser de 6% en moyenne par an de 2005 à 2010, tiré par le développement des équipements électroniques dans tous les domaines d'application incluant des marchés de masse et des équipements électroniques professionnels. L'industrie de l'électronique est devenue mature et étroitement liée au développement de l'économie mondiale. La croissance globale de l'industrie de l'électronique devrait se répartir entre tous les domaines d'application comme illustré sur la figure 1 :

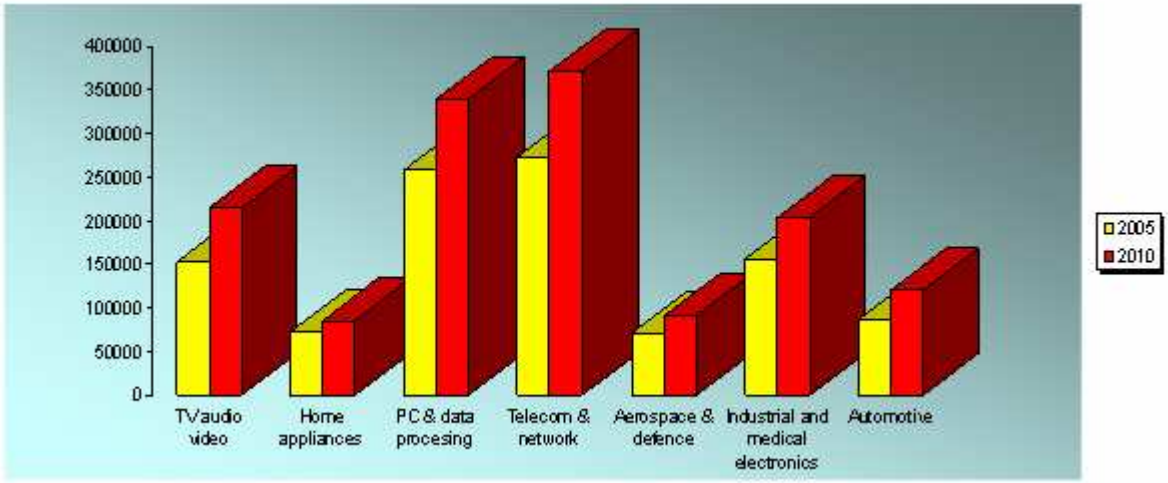


Figure 1: La production électronique mondiale par application, 2005-2010, (million euros)

[1] http://www.decision-consult.com

La tendance est bien sûr aux marchés de masse et cela entraîne une rationalisation des méthodes et des outils de développement des systèmes embarqués.

La complexité des applications présentes sur le domaine des systèmes embarqués est en pleine croissance. Aujourd'hui les besoins des consommateurs tendent de plus en plus vers des applications riches où l'on demande à être connecté à internet, d'avoir un aspect sécurité et une interface graphique étendue... Et cette complexité devrait croître toujours plus.

Les défis sont grands et l'activité autour des méthodes, des outils de développement des systèmes embarqués temps réel est très riche :

- L'automobile se mobilise aussi autour d'une initiative de standardisation des architectures matérielles et logicielles des systèmes embarqués,
- L'aéronautique explore les ressources de l'Open Source,

L'innovation technologique dans l'industrie électronique a également un impact sur l'évolution des systèmes embarqués dans tous les secteurs. Par exemple, le secteur spatial est confronté à la disparition du marché des composants "durcis" indispensables pour un bon fonctionnement des calculateurs dans l'espace et doit développer ses propres générations de processeurs. Ainsi étudie-t-il une nouvelle génération de processeurs multi-cœurs (quadriprocesseurs) appelé GINA.

Les défis actuels du développement des systèmes complexes tels que les systèmes de communication, les systèmes embarqués ou les systèmes de traitement d'images consistent à réaliser avec succès de nouveaux produits fiables, performants et peu coûteux. Relever ces défis passe par un bon choix des méthodes et outils adaptés aux applications visées et aux technologies cibles. En particulier, les composants reconfigurables offrent des niveaux de performances et une flexibilité qui les rendent très attractifs dans un nombre croissant de développements. La reconfiguration dynamique (partielle ou complète) offre la possibilité de réutiliser les mêmes ressources matérielles pour une succession de traitements, et ce de façon analogue à une réalisation logicielle. Ce type d'implémentation est donc particulièrement adapté aux objectifs des systèmes embarqués qui imposent des niveaux de performances parfois élevés avec un nombre réduit de ressources de calcul. Cependant l'exploitation de la reconfiguration dynamique est rendue difficile par le manque d'outils adaptés. Des solutions spécifiques sont proposées mais peu adaptées au cas général.

CONTRIBUTION

De nombreuses applications, en particulier en télécommunication et multimédia, ne nécessitent pas des réalisations temps réel strict c'est à dire des

implémentations visant à obtenir une qualité de service adaptée aux besoins donc suffisante (temps réel souple ou mou). On peut citer par exemple le cas d'une caméra fixe de télésurveillance qui adapte ses traitements en fonction de la nature des images acquises ou un terminal mobile multimodal qui change de norme de transmission si la qualité du canal de communication l'exige ou encore la couche applicative d'un mobile dans un réseau ad hoc.

Par conséquent, au lieu d'effectuer un travail de conception sur la base des temps d'exécutions dans le pire cas (Worst Case Execution Time) des tâches (approche requise pour le temps réel strict) le travail de cette thèse consiste à profiter du fait que les temps d'exécution réel des tâches sont souvent inférieurs au cas pire. D'un point de vue de l'architecture, le premier cas implique des ressources souvent non nécessaires liées au pessimisme de l'analyse et de la conception basées sur les valeurs des WCET.

Dans l'approche proposée nous introduisons la possibilité d'allouer et d'ordonnancer dynamiquement les tâches en fonction d'une estimation de leurs temps d'exécution afin de respecter au mieux les contraintes de temps. Ceci se traduit au niveau de l'architecture par une plate-forme auto-adaptative, c'est à dire capable de déterminer au cours de l'exécution de l'application (donc en ligne) l'allocation et l'ordonnancement des tâches qui peuvent induire des reconfigurations partielles du circuit.

L'architecture visée est un RSoC (Reconfigurable System on Chip) composé d'un processeur, d'un composant reconfigurable dynamiquement et d'une interface intelligente entre le processeur et le reconfigurable. D'un point de vue fonctionnel, outre les ressources de calcul, le système se compose d'une unité d'estimation des temps d'exécution, d'une unité de partitionnement (allocation) et d'un ordonnanceur. Ce type de système est adapté aux applications qui opèrent sur des flots de données avec des temps d'exécution dépendants des données traitées. Par exemple, ce type de situation se rencontre fréquemment en traitement d'images. La reconfiguration dynamique est également bien adaptée en télécommunication avec en particulier le problème de la Radio Logicielle qui vise à développer des systèmes multi-modes ou multi-normes pour les communications sans fil.

La méthode et l'outil de partitionnement développés dans cette étude portent sur une idée originale qui a nécessité des études préalables sur l'approche générale contrairement aux travaux plus classiques qui consistent à améliorer une situation existante.

PLAN DU MEMOIRE

Le mémoire est structuré en six chapitres qui peuvent être classés en trois parties: l'état de l'art, le partitionnement en ligne et la validation de l'approche.

L'état de l'art est présenté dans les deux premiers chapitres de ce mémoire: dans le premier chapitre nous exposons les principales méthodes de partitionnement logiciel/matériel développées dans la littérature et nous les comparons d'un point de vue du modèle de l'application, de l'architecture cible, de la fonction coût considérée et de la technique de partitionnement utilisée. Le deuxième chapitre est consacré à la présentation des récentes avancées technologiques des architectures à reconfiguration dynamique.

L'approche de partitionnement logiciel/matériel développée dans notre étude est introduite dans le chapitre trois ainsi que le flot global de notre méthodologie.
Les étapes de ce flot sont détaillées dans les chapitres quatre (module de prédiction) et cinq (module Partitionnement/Ordonnancement).

La validation de l'approche proposée fait l'objet du chapitre six où sont exposés les résultats d'estimation, partitionnement et ordonnancement pour une application test : une caméra intelligente qui effectue la détection de mouvement sur un fond d'image fixe.

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

CHAPITRE 1

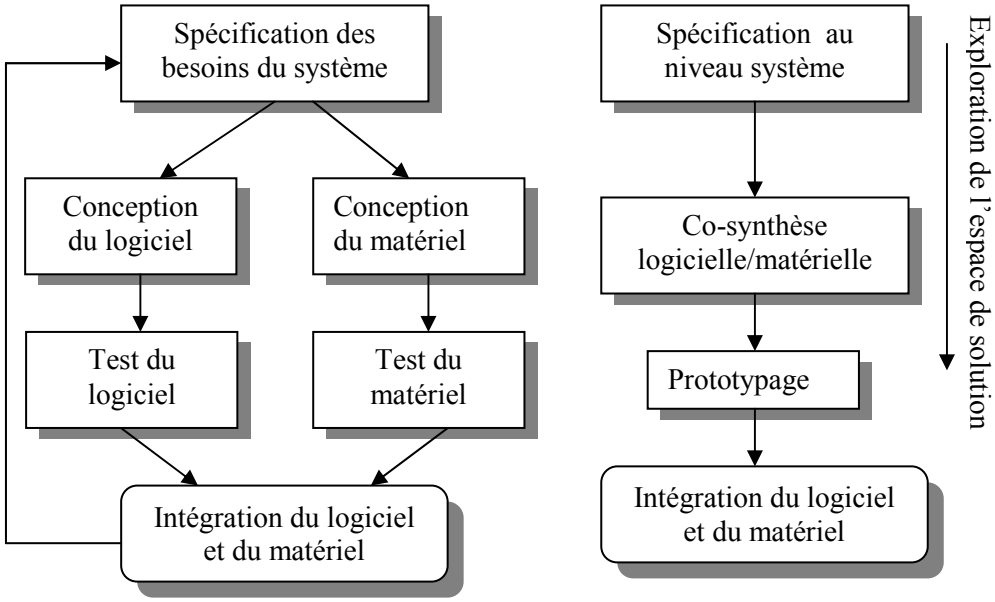
Etat de l'art du partitionnement logiciel/matériel

Introduction

Les techniques et les capacités d'intégration des circuits électroniques n'ont cessé de s'accroître au cours de ces dernières années. L'intégration de systèmes toujours plus complexes nécessite la définition de nouvelles méthodes de conception afin d'exploiter pleinement les évolutions technologiques.

Dans une approche classique, la conception du logiciel et celle du matériel sont séparées et ce n'est qu'à la fin du processus de conception que les différentes parties sont testées ensemble (voir figure 1.1 (a)). Avec cette approche, une réalisation générée risque de ne pas répondre aux contraintes imposées (contraintes temps réel, coût, performances, consommation, fiabilité…). En plus, cette réalisation peut contenir des erreurs de conception aux interfaces entre le logiciel et le matériel. Dans ce cas, il faut reprendre le cycle de conception.

Pour surmonter les limites de l'approche classique et pouvoir aboutir à des réalisations efficaces, l'approche de conception conjointe (*Codesign*) a vu le jour (voir figure 1.1 (b)).



(a) Approche classique (b) Approche de codesign

Figure 1.1 : La stratégie de conception d'un système

La conception conjointe logicielle et matérielle, permet de maîtriser la conception des systèmes complexes. L'analyse des performances avant la fabrication permet une exploration rapide de plusieurs alternatives d'architectures ce qui offre au concepteur une meilleure visibilité et une grande réactivité vis-à-vis des changements technologiques (fiabilité, optimisation, flexibilité, etc.).

Le processus de *codesign* inclut donc plusieurs étapes de raffinement de la conception et des étapes de validation. Partant d'une spécification au niveau système, l'architecture sera raffinée jusqu'à la définition de l'architecture logicielle/matérielle de réalisation. Le processus de conception est donc un chemin progressif, à la fois pour déterminer une solution fonctionnellement correcte et pour exprimer et évaluer les performances à chaque niveau de développement.

L'un des points clés du processus de conception conjointe *Codesign* est le partitionnement logiciel/matériel. Cette étape consiste à obtenir une répartition des tâches entre les unités logicielles (Processeur, DSP…) et les unités matérielles (FPGAs, ASIC, etc.) en fonction des ressources disponibles et des architectures cibles. Ce partitionnement logiciel/matériel doit prendre en considération un certain nombre des contraintes à respecter et des objectifs de performances à atteindre. Dans le cas général le problème du partitionnement est un problème NP-complet composé d'un problème d'ordonnancement et d'un problème d'allocation à plusieurs paramètres.

Depuis plusieurs années, de nombreux travaux ont proposé des approches pour le partitionnement d'une application. Dans la littérature, on peut identifier trois classes d'approches différentes pour aborder le problème du partitionnement:

Les premières sont de nature logicielle [1]. Elles commencent donc à partir d'une spécification entièrement logicielle et cherchent à faire de la migration de code vers le matériel. Les parties critiques d'un système sont identifiées puis affectées à une réalisation matérielle.

Les deuxièmes sont de nature matérielle [2]. Elles partent d'une spécification initiale entièrement en matériel. Par la suite, les parties non critiques sont identifiées afin de les affecter à une réalisation logicielle, ce qui permet de réduire le coût de réalisation.

Les troisièmes sont de nature système (concepteurs de systèmes). Elles ne se limitent pas à un type particulier de spécification en entrée. Dans ce type d'approche, les différentes parties d'une spécification sont affectées à une réalisation (logicielle ou matérielle) qui satisfait les contraintes de conception

(temps de réponse, temps d'exécution, consommation, surface de silicium…) [3].
L'affectation d'une tâche vers une réalisation logicielle ou matérielle se base sur une fonction objective à optimiser.
Dans la suite nous analysons quelques approches de partitionnement en les distinguant en fonction de leurs niveaux de granularité choisis, le modèle de l'application, l'architecture cible, la fonction coût, la technique de partitionnement utilisée, le degré d'automatisation de l'approche et le type d'application envisagé.

1.1 Méthodologies de partitionnement

1.1.1 Niveau de granularité considéré

Dans [4] la granularité est définie comme suit :" Etant donné une description comportementale d'un algorithme, la granularité détermine comment diviser cet algorithme en un ensemble B de n blocs bi dans le but de distribuer ces blocs sur les parties logicielles et matérielles d'un système mixte logiciel/matériel."
B = { b0, b1,…. bn-1}
On parle de granularité fixe quand la taille de ces morceaux reste constante. De nombreux travaux présentent le cas d'une granularité dynamique [5] pour pouvoir adapter les étapes d'optimisation aux spécificités de l'application lors du partitionnement.

Les premières approches de partitionnement logiciel/matériel automatique ont été présentées dans le système VULCAN II [6] et le système COSYMA [1]. Ces approches travaillent à un niveau de granularité fin (bloc de base).

Les approches développées ensuite dans [7,8,9,10,11,12,13,2,1] utilisent également le niveau de granularité grain fin alors que les travaux dans [9,14,15,16,17,18,19] sont basés sur le niveau de granularité gros grain correspondant aux fonctions ou processus de l'application.

Un intérêt d'une granularité fine est que les éléments à partitionner (par exemple des blocs de base) ont des temps d'exécution généralement constants et plus facilement prédictibles que dans le cas d'une granularité de niveau fonction ou processus.

Le partitionnement d'application dont les éléments ont des temps d'exécution dépendants des données peut s'aborder de plusieurs façons.

La première consiste à considérer une granularité plus fine pour réduire la dépendance. On peut atteindre dans ce cas une explosion combinatoire liée au grand nombre d'éléments à partitionner si la granularité considérée est trop fine. Une autre approche consiste à définir des bornes maximales de temps d'exécution mais dans ce cas on est confronté soit à des difficultés pour

déterminer ces bornes (par exemple en traitement d'images) soit à un surdimensionnement de l'architecture obtenue. Un compromis peut être de considérer un niveau de granularité moyen, c'est-à-dire au niveau boucle tel que cela est considéré dans les travaux [19,20,21,22,23].

1.1.2 Architecture Cible

Dans [24] l'architecture cible est un FPGA reconfigurable dynamiquement connecté à un processeur qui contrôle la séquence des contextes de configuration. Les travaux dans [1,2,19,18,25] ciblent un processeur connecté à un ASIC (Application Specific Integrated Circuit). Des architectures multiprocesseurs sont utilisées dans [26,27,28,29,30]. Les approches de partitionnement dans [31,32] utilisent un banc de test composé de FPGA et des interconnexions existantes dans la carte mère d'un ordinateur à usage général.

Pour des raisons de simplification, une architecture de réalisation prédéfinie est utilisée dans [7,33]. Cette architecture est formée par un noyau RISC (Reduced Instruction Set Computer), une mémoire principale, un ensemble de circuits spécifiques et un circuit d'interface pour le contrôle de la communication entre le processeur RISC et les ASIC. Il s'agit dans ce cas d'une architecture du type maître/esclave où le RISC contrôle l'activité de chaque ASIC.
Dans [34], une implémentation basée sur une plate-forme d'architecture paramétrable est discutée. Dans ce cas, une bibliothèque de modules matériels et logiciels paramétrables a été construite.

D'autres travaux de partitionnement logiciel/matériel ne visent aucun modèle d'architecture. Nous citons par exemple la méthodologie de l'université de Cincinnati (le système RAPID) qui n'adopte aucune architecture cible [35]. De même, la méthodologie de l'université de Linköping ne cible aucune architecture précise; les résultats de partitionnement sont transposés sur un ensemble de représentations d'une conception avec une interface bien définie [10].

Nous pouvons citer aussi les travaux qui laissent le choix de l'architecture à un outil automatique effectuant l'étape d'allocation. Cette étape consiste à trouver le meilleur ensemble de composants pour implémenter les fonctionnalités d'un système. L'outil choisit alors ces composants parmi plusieurs composants. On trouve à une extrême 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 tels que les processeurs à usage général. Entre ces deux extrêmes, existent plusieurs composants (exemple, les composants reconfigurables) qui offrent différents compromis en terme de coût, performance, flexibilité, consommation, taille et fiabilité…

Parmi les approches qui ont traité ce problème d'allocation d'une façon automatique, nous pouvons référencer l'étude faite dans [36] qui à partir d'un

ensemble de fonctions, détermine de façon automatique une allocation sur une architecture multiprocesseur qui satisfait des contraintes de temps et de coût. Les outils MOGAC et COSYN présentés dans [37] sont aussi des approches qui cherchent la meilleure solution de surface de silicium respectant des contraintes temporelles.

L'outil de partitionnement CODEF [38], élaboré au laboratoire I3S en collaboration avec Philips Semiconductors, est un environnement d'exploration automatique de différentes allocations de processeurs, de DSPs et d'accélérateurs matériels, qui vérifient des contraintes de temps d'exécution tout en cherchant à optimiser la surface de silicium.

Dans nos travaux nous ciblons une architecture reconfigurable dynamiquement formée par un cœur de processeur connecté à une unité reconfigurable. Nous revenons en détails sur ce type d'architectures et leurs différentes technologies dans le second chapitre de ce manuscrit.

1.1.3 Modèle de l'application et langage de spécification

La modélisation du système spécifie sa fonctionnalité désirée et ses restrictions. La fonctionnalité est décomposée en modules qui, ensemble, forment le modèle conceptuel du système. La spécification peut se baser sur un langage de description au niveau système ou sur une autre description qui fournit un modèle exécutable.
Le modèle exécutable permet la représentation du système par un ensemble de sous-systèmes ou de modules fonctionnels. Les modules devraient offrir les caractéristiques suivantes [39] :

- ✓ Etre formels et non ambigus;
- ✓ Etre complets, de façon à ce qu'ils décrivent la totalité du comportement désiré;
- ✓ Permettre une compréhension facile pour la correction, la maintenance et la réutilisation;
- ✓ Etre modulaires, avec une interface et un comportement bien définis.

De nombreux modèles ont été proposés dans la littérature pour représenter des systèmes électroniques mixtes. Ces modèles peuvent être écrits soit dans un langage de description de matériel (VHDL, Verilog, Hardware C etc…) soit de logiciel (C, assembleur…) soit dans un langage de description au niveau système (SDL, StateCharts, CSP, SpecCharts, Syncharts …).

Dans ce qui suit nous allons présenter les principaux modèles qui ont été utilisés dans le codesign.

1.1.3.1 FSM (Finite State Machines)

Les machines à états finis sont les modèles les plus connus pour la description d'un système qui présente du contrôle. Le modèle consiste en un ensemble d'états Q, un ensemble d'entrées I, un ensemble des sorties O, une fonction I*Q→O qui définit les sorties et une fonction de transition I*Q→Q.

L'un des inconvénients de ce modèle est l'accroissement du nombre des états en fonction de la complexité du système lorsque celui-ci contient du parallélisme.
Un modèle de FSM étendu qui supporte la hiérarchie et la concurrence a été utilisé dans le système SOLAR [40].

Le CFSM (Codesign Finite State Machine) est un modèle formel pour la conception conjointe logicielle/matérielle. Ce modèle proposé par Chiodo et al [41] est une extension du modèle FSM destiné aux systèmes orientés contrôle avec une faible complexité algorithmique. La primitive de communication entre CFSM est appelée "event". Elle représente l'asynchronisme entre les FSM synchrones du modèle.
Dans [42], un modèle appelé Statecharts est utilisé. Il est considéré comme une extension de diagramme d'états traditionnel (Traditional State Diagrams). Ce modèle est principalement destiné aux systèmes réactifs complexes avec un degré de concurrence élevé.

1.1.3.2 Systèmes à événements discrets

Les systèmes à événements discrets peuvent être définis comme des systèmes à événements commandés avec des états discrets. L'évolution de l'état du système dépend entièrement de l'occurrence des événements asynchrones discrets dans le temps [43].

La modélisation par événements discrets est très générale et ne fait que peu d'hypothèse sur les occurrences des événements. De nombreux travaux portent sur ce modèle [44] qui cherchent à appliquer des méthodes pour garantir des propriétés sur le modèle : absence de blocage, accessibilité par exemple.

1.1.3.3 Réseaux de Petri

Dans une approche classique, les réseaux de Petri sont composés de 4 éléments de base: un ensemble de places, un ensemble de transitions, une fonction d'entrée qui relie les transitions aux places, et une fonction de sortie qui relie les places aux transitions [45]. Les deux caractéristiques importantes des réseaux de petri sont la concurrence et l'asynchronisme.

Dans [46], un réseau de Petri hiérarchique (HPN: Hierarchical Petri Net) est utilisé comme modèle de représentation : les transitions représentent les

composants actifs (fonctions) et les places représentent des composants passifs (données ou conditions).

Dans [47], une technique de modélisation pour les systèmes logiciels/matériels est basée sur un réseau de Petri temporisé étendu (ETPN: Extended Timed Petri Net).

1.1.3.4 Graphe de flots de données (DFG)

Les DFG sont souvent utilisés pour modéliser les systèmes suivant une approche fonctionnelle en se focalisant sur les dépendances des données entre fonctions. Ainsi ce type de systèmes peut être facilement représenté par un graphe dirigé dont les nœuds décrivent les traitements et les arcs représentent l'ordre partiel suivi par les traitements.
Dans ce modèle, les calculs sont exécutés seulement lorsque les opérandes exigées sont disponibles. Le modèle DFG est bien adapté pour représenter le parallélisme fonctionnel.
Plusieurs variétés de DFG ont été utilisées pour représenter des systèmes hétérogènes. Par exemple, le DFPN (Data Flow Process Networks) est un modèle hiérarchique utilisé pour représenter les systèmes orientés traitement de signal [48]. Dans [49], une comparaison a été effectuée entre les modèles SDF (Synchronous Data Flow) et CSDF (Cyclo-Static Data Flow). La différence entre ces deux modèles est que dans le modèle SDF les fonctions consomment et produisent un nombre fixe de jetons à chaque activation ; alors que dans le modèle CSDF les tâches ont des règles de production/ consommation de jetons qui évoluent de manière cyclique.

1.1.3.5 Processus communicants

Dans les modèles dérivés de CSP (Communicating Sequential Processes) [50], les systèmes sont représentés sous forme d'un ensemble d'éléments (processus) qui s'exécutent indépendamment. Ces processus communiquent l'un avec l'autre à travers des canaux unidirectionnels utilisant un protocole asynchrone. Les processus peuvent être décrits en tant qu'événements (un événement est une action atomique de durée nulle). Le temps d'exécution des actions doit être calculé en utilisant une paire d'événements.
Dans [32], Thomas et al utilisent un ensemble de processus séquentiels indépendants dérivés de CSP, qui modélise le comportement des systèmes mixtes logiciel/matériel. Ce modèle est utilisé pour la simulation et la synthèse au niveau système.

1.1.3.6 Modèles synchrones/réactifs

Les systèmes réactifs sont ceux qui interagissent continuellement avec leur environnement. Le modèle synchrone est une représentation pour les systèmes temps réel réactifs.

Dans ce modèle de calcul, les réponses en sortie sont produites simultanément avec les stimulations en entrée.

ESTEREL est un langage de programmation pour les systèmes réactifs sous un modèle de calcul synchrone [51]. Dans cette approche de modélisation, les réactions sont instantanées et les sorties sont donc produites de façon synchrone avec les entrées.

Avec ESTEREL les modules synchrones communiquent à travers des signaux; il s'agit d'un langage déterministe bien adapté pour décrire les systèmes réagissant avec des stimuli externes et orientés contrôle.

Cette brève présentation sur les modèles illustre qu'à chaque modèle est associé un ensemble d'hypothèses que doivent vérifier les systèmes ciblés par ce modèle.

En d'autres termes, le choix d'un modèle de description est effectué en fonction de l'adéquation des hypothèses associées par rapport aux caractéristiques du système à modéliser.

1.1.4 Fonction coût

Un algorithme de partitionnement logiciel/matériel vise à distribuer les fonctionnalités de l'application sur les entités de l'architecture suivant un ou plusieurs critères qu'il s'agit d'optimiser. Il faut donc tout d'abord spécifier la nature de ces critères regroupés généralement dans une fonction coût. Celle-ci joue un rôle très important car selon les critères à optimiser, les solutions d'allocation des entités de calcul aux fonctionnalités de l'application peuvent varier fortement.
Dans la littérature, les critères d'optimisation sont assez variés. La plupart des approches considèrent une contrainte de temps réel [1,2,52] et recherchent une accélération des traitements en profitant des temps d'exécution plus faibles sur la partie matérielle tout en minimisant la surface de silicium induite par la partie matérielle.

Dans le système Vulcan [2,6,53,54], les critères à optimiser sont le temps d'exécution avec des contraintes de délai Minimum/Maximum. A l'Université de Californie/Irvine et dans le cadre du système SpecSyn [3,55,56], les algorithmes de partitionnement cherchent à minimiser le temps d'exécution ainsi que la taille du code et celle des données. L'évaluation des performances du système à développer est faite par des techniques de simulation et d'estimation. Un estimateur de logiciel permet de fournir des métriques relatives au logiciel généré en fonction du processeur cible. Cette approche est basée sur l'identification des goulots d'étranglement dans le système. Il est

ainsi possible de repérer des fonctions où l'utilisation d'un circuit spécifique peut améliorer les performances.

A l'université de Californie/Berkeley, le partitionnement dans le système Ptolemy est guidé par des contraintes de surface, de vitesse et de flexibilité. Lors de la synthèse du logiciel pour une architecture multiprocesseur, le partitionnement essaie d'optimiser des fonctions coûts telles que : coût des communications, tailles des mémoires locales et globales [57,58,18,34,59,60,61].

Dans la méthodologie de l'université de Tübingen [7,33], le partitionnement vise à maximiser le parallélisme entre les composants, supporter la réutilisation des structures matérielles, et minimiser les coûts de communication et l'utilisation des ressources.

La fonction de coût est donc adaptée en fonction des objectifs visés et des contraintes du système. Les fonctions sont de plus en plus multi-critères (temps, surface, consommation) ce qui inclut à la fois des compromis et également des temps de partitionnement ayant tendance à augmenter. Dans notre cas de partitionnement en ligne nous aurons à être attentif à ce dernier point.
Dans la suite nous définissons le type d'application ciblé par notre approche de partitionnement en ligne ce qui va naturellement nous orienter vers le modèle le mieux adapté.

1.1.5 Types d'applications envisagées

Comme indiqué ci-dessus, le type d'application envisagé conduit à retenir un modèle adapté et par conséquent le domaine d'application est décisif dans le choix de la méthode de partitionnement. Dans la suite on présente les différents types d'applications considérées dans le cadre de méthodologies de conception de systèmes mixtes logiciels/matériels.

Dans la méthodologie de l'université de Californie/Berkeley (système Ptolemy), les applications sont de type traitement du signal [62,63] ou systèmes communicants. Le logiciel est formé par un ensemble de programmes, assez complexes, s'exécutant sur des composants programmables.

Dans la méthodologie de Siemens (système CODES), les applications sont de type systèmes de communication. Un modèle abstrait pour les machines à accès aléatoire (PRAMs : Parallel Random Access Machines) est adopté. Ce modèle supporte la communication point à point et celle par diffusion. Un système est représenté à travers des unités communicantes, chacune d'elles représente un comportement [64,65,66].

Dans nos travaux nous considérons des applications orientées flots de données qui gèrent des séquences périodiques de données.
De plus, nous considérons des applications ayant une granularité importante, c'est-à-dire que les volumes de calcul sont élevés par rapport aux volumes de données traitées. Ceci implique que les temps de reconfiguration des ressources de calcul dans une architecture reconfigurable peuvent être réduits (ou masqués) par les temps de calcul ou de communication.

1.1.6 Technique de partitionnement et critère de classification

Le partitionnement d'une spécification est un problème NP-difficile du fait du grand nombre de paramètres à considérer.
Il y a plusieurs critères pour classer les approches de partitionnement qui ont été proposées dans le domaine de la conception des systèmes embarqués. Nous pouvons classer les approches suivant le degré d'automatisation de leurs algorithmes:

- Automatique
- Interactif
- Manuel

Par exemple, les travaux dans [2,1,7,12,14,17,67] utilisent une approche automatique pour le partitionnement, par contre [9,68,69,70] sont basés sur des partitionnements manuels.

Un autre critère de classification qui nous intéresse dans notre travail, est de prendre en compte le changement ou pas de résultats de partitionnement en cours d'exécution de l'application. En fait, suivant ce critère, il existe trois types d'approches :

- ❖ Approche Statique : le partitionnement reste inchangé tout au long de la durée de vie du système
- ❖ Approche Semi-statique : le partitionnement change de décisions en ligne en se basant sur des résultats trouvés statiquement hors ligne
- ❖ Approche Dynamique : le partitionnement est fait intégralement en ligne et peut changer dynamiquement suivant les besoins du système.

Dans la suite de notre étude bibliographique nous considérons ce dernier critère de classification pour présenter des travaux de partitionnement des plus traditionnel jusqu'aux plus récents: statique, semi-statique puis dynamique.

1.1.6.1 Approche Statique

C'est l'approche la plus classique de partitionnement. Le concepteur cherche un résultat de partitionnement qui sera valable (optimisé au mieux) pour toutes les exécutions du système. Pour cela il se base en général sur des cas de figure prises au pire cas: WCET (Worse Case Execution Time) afin de garantir la satisfaction des contraintes temps réel. Il s'agit bien alors d'un travail de partitionnement hors ligne (Off-line) nécessitant une étude préliminaire de l'application, de l'architecture et de l'environnement d'exécution.

Dans la suite nous présentons quelques approches statiques qui ont été employées dans le domaine du co-design.

a. Le recuit simulé

Le recuit simulé [1,10,17,71] est inspiré du processus de recuit où la matière est tout d'abord fondue puis ensuite refroidie lentement d'une façon contrôlée pour obtenir un certain arrangement des atomes. Quand la température est élevée, les atomes peuvent se déplacer vers les états d'énergie les plus élevés. Mais, avec la baisse de la température la probabilité de ces déplacements (ou mouvements) est réduite. Dans la procédure d'optimisation, l'énergie d'un état correspond à l'inverse de la valeur de la fonction qualité. La température devient un paramètre de contrôle qui va être réduit durant l'exécution de l'algorithme.

A chaque itération, un voisin de la solution courante obtenue après un mouvement choisi au hasard, est retenu s'il est meilleur que la solution courante. Dans le cas contraire, le mouvement est retenu avec une probabilité fonction de la différence de qualité et en tenant compte de la température courante. Pendant les premières itérations, la probabilité d'accepter des mauvaises solutions est ainsi élevée H; mais avec la baisse de la température, l'acceptation devient de moins au moins possible.

b. L'algorithme génétique

Les algorithmes génétiques [72,73] sont des algorithmes d'optimisation s'appuyant sur des techniques dérivées de l'évolution naturelle : croisements, mutations et sélections basées sur la survie du meilleur. Le but est d'adapter une espèce aux changements de son environnement.

L'algorithme considère une population d'individus qui représente un sous-ensemble de l'espace des solutions. A chaque itération, certains individus seront remplacés par des nouveaux par croisement ou par mutation. Les premiers sont créés en sélectionnant des paires d'individus de la population antérieure dont la probabilité d'être sélectionné est proportionnelle à la qualité

de l'individu. La mutation consiste à effectuer des changements aléatoires dans les gênes des individus de la population antérieure pour favoriser l'exploration de l'espace des solutions.

c. Méthode de recherche Tabou

La recherche tabou [74] est une méthode déterministe qui repose sur une gestion dynamique de deux types de mémoires : l'une à court terme assure une exploration intensive, l'autre à long terme assure une diversification. La mémoire contient des caractéristiques dites tabou i.e. que l'on s'interdit de choisir pour les futures solutions. L'objectif premier est d'éviter une exploration cyclique pour augmenter l'efficacité de la recherche.

Le principe de la recherche tabou est assez simple. À chaque itération on essaie de faire diminuer le coût d'une solution. Dans le cas ou l'amélioration est impossible, on prend la solution qui détériore le moins la solution courante. C'est ce qu'on appelle un mouvement tabou. Une liste taboue (mémoire) mémorise les dernières solutions visitées (celles sur lesquelles un mouvement tabou a été appliqué).

Ceci est utile au moment où l'algorithme n'arrive plus à améliorer la solution courante. En effet on ne peut effectuer un mouvement tabou qui détériore une solution se trouvant sur la liste. Par contre, si le mouvement tabou produit une amélioration par rapport à la meilleure solution trouvée, la liste tabou n'est pas prise en considération.

d. Approche par migration du matériel vers logiciel

Cette approche gloutonne est utilisée à l'université de Californie/Stanford dans le système VULCAN [75]. L'algorithme de partitionnement commence avec une partition initiale d'un code Hardware C où toutes les opérations, exceptées celles à délai non limité, sont affectées au matériel. La partition est raffinée par la migration d'opérations du matériel vers le logiciel afin d'obtenir une partition à moindre coût.

L'approche utilise un ensemble de modèles de graphes de séquencement avec des contraintes temporelles entre les opérations. Un ensemble de modèles de graphes de séquencement est réalisé en logiciel et un autre est réalisé en matériel. Le logiciel est composé par plusieurs threads.

e. *List Scheduling modifié*

L'idée générale du List Scheduling est de traverser en série une liste de nœuds (généralement des nœuds source aux nœuds feuille) et pour chaque nœud on choisit l'implantation qui minimise une fonction objective.
La limite de cette méthode est que le fait de traverser en série la liste des nœuds conduit à des décisions locales qui peuvent conduire à des solutions éloignées de l'optimum.

L'algorithme GCLP (Global Criticality/Local Phase) [18] essaie de surmonter cet inconvénient. A chaque étape de l'algorithme, il choisit d'une façon adaptative l'objectif approprié de l'implémentation. Le choix est fait suivant le critère GC (Global Criticality) qui estime le temps critique à chaque étape de l'algorithme. GC est comparé à un seuil pour déterminer si le temps est critique.

Dans le cas où le temps est critique, une fonction objective qui minimise le temps d'exécution est choisie, sinon une autre fonction qui minimise la surface est prise en compte.

Une autre mesure utilisée dans le choix de la fonction objective est le critère LP (Local Phase). Il s'agit d'une classification des nœuds basée sur leur hétérogénéité et leur propriété intrinsèque. Chaque nœud est classé soit en "extrémité" (LP1), soit en "repeller" (rejeté) (LP2) soit en normal (LP3).

Le GC et le LP sont utilisés pour choisir une implémentation (Mappy Mi). Le processus est répété jusqu'à ce que toutes les tâches soient implémentées.

f. *Min Cut modifié*

Supposons un graphe dirigé avec des valeurs de débit sur chaque arc. On a un noeud source et un noeud destination. Lorsque le débit maximal est atteint, le long de chaque chemin entre la source et la destination il y a au moins un goulot d'étranglement qui est soit un arc qui fonctionne à capacité dans la bonne direction soit un arc en sens contraire qui ne transporte rien. Autrement, il est possible d'augmenter le débit. Ceci est équivalent à prendre des coupes et voir celle qui permet le plus petit débit, ce qui sera le facteur limitant pour le réseau (min-cut).

Partir avec un réseau dont les valeurs de flot sont à 0.
Partir de la source et ajouter les arcs (et noeuds) selon les critères suivants, si l'autre extrémité n'est pas encore dans l'ensemble cherché.
* L'arc sort de l'ensemble et son flot n'atteint pas la capacité de l'arc. La capacité additionnelle pouvant rejoindre le noeud au bout de l'arc étant le minimum entre celle du noeud courant et celle de l'arc (capacité moins flot).

* L'arc entre dans l'ensemble et son flot n'est pas nul. La capacité additionnelle est alors le minimum entre la capacité additionnelle du noeud courant et le flot (négatif) de l'arc.

Les noeuds ainsi rejoints sont alors cherchés à leur tour selon l'ordre dans lequel ils sont rencontrés (breadth-first).

Lorsque la destination est rejointe, on a trouvé un chemin le long duquel la capacité n'est pas toute exploitée. On change alors le flot en conséquence et on recommence.

Si on ne peut pas aller plus loin, on a trouvé le min-cut composé d'arcs remplis à capacité vers l'avant et d'arcs à flot nul vers l'arrière.

g. Exploration récursive SW-HW

Le partitionnement logiciel/matériel par programmation dynamique proposé par Madsen et al dans la méthodologie PACE [76] permet de résoudre le problème de la minimisation du temps d'exécution total en tenant compte de la contrainte de surface allouée au matériel. Il permet aussi de résoudre le problème dual c'est à dire minimiser la surface en tenant compte de la contrainte de temps d'exécution total.

Pour autoriser un passage d'une tâche du logiciel vers le matériel, l'algorithme calcule l'accélération et la pénalité en surface liées à ce passage. Ensuite il évalue l'accélération induite par les nouvelles communications (après le passage). Le passage qui apporte la plus forte accélération comparativement à l'augmentation de surface, est retenu.

Une approche de partitionnement logiciel/matériel basée sur une programmation dynamique est développée dans [77]. Dans ce travail, chaque fonction nécessite une configuration totale d'un circuit reconfigurable type FPGA. A chaque événement un ensemble de procédures est répété :

- Chercher la fonction qui termine au plus tôt
- Effacer la configuration de la fonction qui a fini son exécution
- Mettre les successeurs de la fonction finie dans la liste d'attente si leurs prédécesseurs se sont déjà exécutés.
- Essayer toutes les solutions d'implantations des fonctions sur les différentes ressources (en incluant la possibilité de laisser une ou plusieurs ressources libre(s))
- Choisir la solution qui donne le temps d'exécution minimum.

Un exemple d'exécution de ce dernier algorithme sur un graphe de fonctions contenant 40 nœuds et 600 arcs a mis moins d'une seconde pour trouver le partitionnement optimal.

h. Formulation en ILP

Une autre méthode de résolution du problème du partitionnement logiciel/matériel est de le formuler sous forme d'un programme linéaire en nombres entiers (en anglais ILP : Integer Linear Programming).

Les études de partitionnement et d'exploration architecturale par Kaul et al. en 1998 [78] à l'université de Cincinnati aux Etats-Unis ciblent des architectures reconfigurables dynamiquement et proposent une linéarisation de ce problème pour le mettre sous forme ILP et puis de le résoudre par une méthode de Branch and Bound.

Dans [79], I. Ouaiss et al. ont proposé un flot de partitionnement (appelé SPARCS) avec une étape de partitionnement 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.

Une autre technique basée sur la formulation ILP est proposée par Marwedel [80] pour résoudre à la fois les problèmes de partitionnement et d'ordonnancement.

Nous aurons l'occasion de revenir sur la méthode ILP dans le chapitre 6 : résultats expérimentaux de ce manuscrit pour une étude comparative avec la méthode de partitionnement proposée dans le cadre de nos travaux.

1.1.6.2 Approches Semi-Statiques

Ces approches de partitionnement sont en général plus récentes que les précédentes. Elles se basent à la fois sur une étude statique hors ligne et une analyse complémentaire en ligne. Ces études de partitionnement ciblent généralement des applications à contraintes temps réel mou à l'inverse des approches statiques qui couvrent toutes les applications à contraintes temps réel strict et mou.

Les approches semi-statiques se fondent sur l'objectif de minimisation du pessimisme engendré par les approches statiques précédentes. En effet dans le cas des applications à contraintes temps réel strict, le partitionnement opère généralement sur des tâches à temps d'exécution correspondant aux pires des cas (WCET : Worse Case Execution Time).

L'affectation des WCET aux tâches d'une application à contraintes temps réel mou (devant tendre à respecter des contraintes dont le non-respect temporaire engendre au plus un dysfonctionnement partiel qui ne remet pas en cause la mission) engendre souvent un surdimensionnement de l'architecture, et par la

suite une mauvaise gestion des ressources due à un pessimisme le plus souvent excessif.

L'exemple sur la figure 1.2 montre bien que le WCET est un cas très peu fréquent et que souvent les temps d'exécution sont très inférieurs au WCET.

Figure 1.2 : densité de probabilité du temps d'exécution d'une tâche

Partant de là des approches de partitionnement semi-statiques ont vu le jour [82,83]. De telles approches cherchent à adapter les ressources de traitement aux besoins de la tâche, en d'autres termes : suivant la nature des données à traiter par la tâche, il s'agit de considérer le temps d'exécution effectif et l'allocation de ressources en liaison avec les autres tâches de l'application et en tenant compte des contraintes de temps et de l'ensemble des ressources disponibles.

Les travaux proposés dans [81] nous semblent parmi les premiers à identifier une approche de partitionnement logiciel/matériel semi-statique. L'architecture proposée dans ces travaux est formée par un processeur à usage général connecté à une unité reconfigurable dynamiquement.

L'étape de partitionnement hors ligne consiste à:

- chercher tous les chemins d'exécutions possibles du DFG conditionné [85].
- représenter sous forme de segments la courbe du temps d'exécution de la tâche en fonction d'un paramètre de corrélation et d'associer le temps d'exécution maximum du segment à l'ensemble du segment (voir figure 1.3)
- de transformer le graphe de tâches en un DFG conditionné par duplication de chaque tâche autant de fois qu'il y a des segments identifiés sur la courbe associée à cette tâche.
- appliquer un algorithme génétique sur chacun des ces chemins pour construire des configurations (ou bien contextes) et les charger dans la mémoire de l'architecture. Le travail en ligne consiste, à l'aide d'une

logique de contrôle de charger le bon contexte dans l'architecture lorsque les variations des temps d'exécution des tâches permettent de passer d'un chemin à un autre (changement de segment pour au moins une tâche).

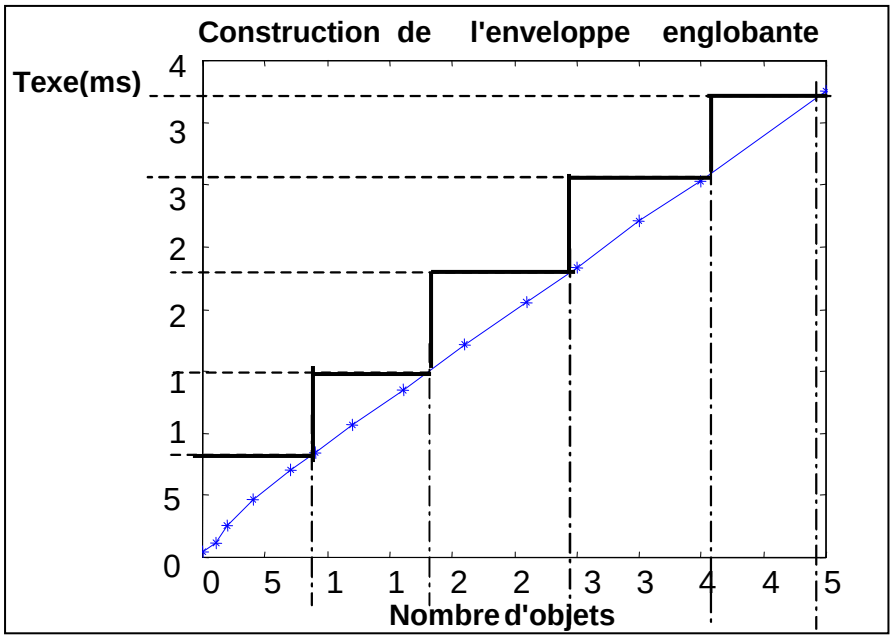


Figure1.3 : Exemple d'une tâche à temps d'exécution dépendant du nombre d'objets dans une image

Une autre étude plus récente (janvier 2006) élaborée par Yu Kwang [84] propose une méthodologie de partitionnement semi-statique appelée *On-Off methodology*. Le principe est d'utiliser en ligne des dérivées des implémentations préparées hors ligne. La méthode commence alors par des implémentations initiales puis re-implémente dynamiquement entre les itérations de l'application les fonctionnalités de l'application en se basant sur les effets de changement de caractéristiques des données appelées paramètres dynamiques.

La phase hors ligne est basée sur un algorithme génétique pour générer des implémentations pour un intervalle de valeurs de paramètres dynamiques. L'ordonnancement des tâches est effectué à l'aide d'une heuristique appelée ECL (*Earliest Completion Time)* [86].

L'architecture cible est multiprocesseur et la fonction objective consiste à minimiser le temps d'exécution total de l'application. Il s'agit d'un problème connu sous le nom équilibrage de charges (*Load Balancing*). Dans ce type de problèmes l'architecture (homogène ou hétérogène) est figée et le partitionnement cherche à redistribuer les charges de calcul sur les machines d'une façon à minimiser le temps d'exécution global de l'application.

Dans [87] une stratégie gloutonne (greedy) est proposée pour la re-implémentation dynamique des applications parallèles travaillant sur des données itératives. Un exemple est étudié avec un problème de dynamique de fluides sur des architectures homogènes.

1.1.6.3 Approches Dynamiques

Abordant la même problématique que celle des approches semi-statiques, ces approches de partitionnement dynamiques permettent au système de s'auto-adapter à l'environnement d'exécution de l'application. Les applications visées sont celles qui présentent des caractéristiques variables en fonction des données à traiter. La différence avec les approches précédentes est l'absence de tout ou partie du traitement effectué hors ligne. Ainsi le système est capable de résoudre le problème du partitionnement logiciel/matériel en ligne, en concurrence avec l'exécution de l'application, à chaque fois que la situation l'impose.

Des approches dans ce sens sont en cours d'études dans quelques laboratoires de recherche. A l'Université de California Riverside, F.Vahid et al proposent pour la première fois une approche de partitionnement logiciel/matériel dynamique dans [88]. Dans ce travail, les auteurs font un profiling sur l'application afin de détecter les boucles logicielles les plus critiques en temps d'exécution. Les boucles sont implémentées en reconfigurable pour pouvoir passer le processeur en mode repos et donc diminuer la consommation d'énergie.

Plus tard, dans [89] les mêmes auteurs présentent un travail plus détaillé et les outils nécessaires pour leur méthode de partitionnement dynamique à savoir les outils de profiling, de décompilation, de synthèse et de placement/routage en ligne. La méthode de partitionnement suivie est composée des étapes suivantes :

1. chercher les parties critiques par profiling
2. décompilation du code logiciel
3. synthèse comportementale vers une cible reconfigurable
4. synthèse logique
5. placement et routage
6. mettre à jour le logiciel pour communiquer avec le matériel.

Une architecture dédiée à cette méthode de partitionnement a été proposée dans [90]. Malgré les efforts fournis pour réaliser une telle approche adaptative, elle reste limitée à des applications séquentielles vu que le partitionnement opère au niveau des boucles d'un code C.

Remarquons aussi que dans ce travail il s'agit d'un bi-partitioning, c'est à dire que le partitionnement fait le choix entre uniquement deux solutions d'implémentation, l'une logicielle et l'autre matérielle. Ceci est une limitation

car dans les architectures systèmes on trouve plusieurs processeurs et des entités de plus en plus souvent reconfigurables par parties.

Parmi les autres travaux qui étudient le problème du partitionnement logiciel/matériel dynamique, nous pouvons citer ceux de [91]. Il s'agit d'une stratégie de partitionnement logiciel/matériel basée sur l'équilibrage de charges basée sur une heuristique évolutionnaire. En fonction d'un changement de demande de puissance de calcul, le système doit réagir par une distribution dynamique des charges sur les ressources disponibles. Un inconvénient majeur de cette approche est que les changements de distribution ne sont pas prédits, donc il y a des pertes de données importantes pendant le régime transitoire. De plus s'il y a de fréquentes variations, le système peut facilement rentrer dans un état instable.

Le travail présenté dans [92] considère aussi le problème de partitionnement logiciel/matériel dynamique. Les auteurs proposent une architecture adaptée pour les applications temps réel embarquées avec la basse consommation comme contrainte. Il s'agit d'une architecture mixte logiciel/matériel où le choix de l'implémentation est fait dynamiquement. Le fait de changer l'implémentation des tâches du matériel vers le logiciel fait diminuer la consommation du système. Mais l'inconvénient de ce travail est qu'il ne présente pas de technique particulière de partitionnement ni d'ordonnancement logiciel/matériel.

Une méthodologie de partitionnement auto-adaptatif est en cours d'étude dans le projet RAAR : Reconfiguration Algorithmique Architecturale Régulée au laboratoire LESTER de l'Université de Bretagne sud [93]. Le système dans ce cas est modélisé par un asservissement en boucle fermée disposant d'un observateur et d'un régulateur. Dans un souci de compromis complexité/efficacité, le système observe et construit son propre modèle dynamiquement. Lorsque c'est nécessaire, il observe son état à l'aide des capteurs disponibles (mesures des temps d'exécution, du niveau de batterie, de la qualité de résultat). Sinon il fait appel au modèle pour estimer son état. Le système est conçu de manière à se stabiliser dans la configuration qui sera la plus favorable par rapport aux contraintes imposées par l'utilisateur. Des résultats de partitionnement dynamique sont encore attendus de ce projet qui intègre des aspects automatiques dans les systèmes à base d'architectures reconfigurables dynamiquement.

Conclusion

Comme nous pouvons le remarquer dans le tableau de synthèse ci-dessous, l'état de l'art du partitionnement logiciel/ matériel est très riche par plusieurs approches avec des techniques de partitionnement différentes.

Auteurs	Modèle	Granularité	Architecture	objectif	Technique
Cosyma [Ernst 93]	C étendu	Instruction	1 proc ; 1 ASIC	Temps	Récuit simulé SW->HW
Vulcan [De Micheli 94]	CDFG	Instruction	1 proc ; 1 ASIC	Temps	Glouton HW->SW
GCLP [Kalavade94]	DFG	Tâche	1 proc ; 1 ASIC	Temps/surface	List Scheduling Modifié
Lycos [Madsen 97]	CDFG	Instruction	1 proc ; 1 ASIC	Temps	Prog. Dynamique SW->HW
SpecSyn [Vahid 97]	CDFG	Fonction	1 proc ; 1 ASIC	Temps/surface	Min Cut modifié
[Teich 97]	CDFG	Fonction	SOC	Temps	Algo. Génétique
Cosyn [Dave 99]	GT	Tâche	Multi proc	Temps/conso	Clustering
Crusade [Dave 99]	GT	Tâche	Multi proc +FPGA	Temps/conso	Clustering
Move [Karkawski 98]	CDFG	Boucles	SOC multi ASIP	Temps/nbre proc	Enumération
Class [Kuchcinski 99]	DFG	Tâche	SOC	Temps/surface	Prog. Par contraintes et B&B
[LI 00]	CDFG	Boucles	1 proc ; 1 FPGA	Temps	Clustering
Syndex [sorel 94]	DFG	Tâche	Multi proc	Temps	List Scheduling modifié
[Wolf 95]	GT	Tâche	Multi proc	Temps/surface	Glouton + Exclusion mutuelles
Codef [Auguin 00]	CDFG	Instruction	SOC	Temps/Surface	Glouton + Exclusion mut.

Vu leur complexité d'implémentation et leurs temps de calcul, toutes ces techniques peuvent s'appliquer facilement dans une étape de conception hors ligne. Mais lorsqu'il s'agit d'une implémentation dans un contexte d'exécution en ligne ces approches ne sont pas réalisables (une approche de partitionnement basée sur les algorithmes génétiques nécessite quelques heures de calcul voire même des jours pour trouver une solution optimale). Le concepteur doit alors chercher d'autres approches de partitionnement basée sur des techniques optimisant leurs temps de calcul, leurs consommations d'énergie, leurs surfaces de silicium, leur flexibilité…

Nous revenons sur ces aspects dans le chapitre 3 de ce rapport qui présente une nouvelle approche de partitionnement logiciel/ matériel en ligne. Mais tout d'abord nous continuons notre étude bibliographique dans le chapitre 2 qui met en exergue les différentes technologies des architectures reconfigurables dynamiquement.

CHAPITRE 2

Architectures à reconfiguration dynamique

Introduction

Une architecture est qualifiée de reconfigurable dès lors qu'elle dispose d'un support lui permettant de s'adapter aux traitements qui lui sont assignés. Plusieurs travaux présentés dans la littérature ont proposé des classifications des architectures reconfigurables. Dans [73], les architectures sont classées selon la granularité, la reconfigurabilité, le couplage et enfin l'organisation mémoire. Dans [94] les architectures sont présentées selon trois critères :

- ✓ le type d'unité de traitement,
- ✓ le type de connexions/ communications
- ✓ les possibilités de reconfiguration.

D'autres travaux classent les architectures reconfigurables suivant les fabricants ou les groupes de recherche travaillant dessus [95]. Nous proposons dans ce chapitre une classification des architectures reconfigurables basée sur les granularités de traitement, de la communication et de la reconfiguration.

Dans ce chapitre nous commencerons par une caractérisation des architectures reconfigurables; une attention particulière sera donnée à la reconfigurabilité. Nous expliquons à la fin de ce chapitre notre choix d'architecture.

2.1 Caractéristiques des architectures reconfigurables

A partir de la littérature traitant des systèmes reconfigurables, nous pouvons retenir quatre caractéristiques essentielles: la granularité de traitement, la granularité de la communication, le couplage avec un éventuel processeur (qui peut être sur la même puce) et le type de reconfiguration. C'est cette dernière caractéristique qui nous attire le plus vue qu'elle confère à l'architecture une possibilité de s'adapter aux besoins de traitements lorsqu'ils sont évolutifs.

2.1.1 Granularité de traitement

Les architectures reconfigurables considèrent généralement deux types de granularités : grain fin et gros grain. Le premier correspond à des ressources logiques souvent de type LUT (Look Up Table). Les LUT sont de petite mémoire que l'on retrouve dans le FPGA. Elles sont capables de réaliser n'importe quelle fonction booléenne simple sur ses entrées. Le deuxième concerne plutôt des architectures construites autour des matrices d'ALU, voir des cœurs de processeurs

élémentaires. D'autres architectures gros grain sont composées d'opérateurs arithmétiques de type additionneur, multiplieur ou MAC.

Souvent les architectures reconfigurables sont hétérogènes, c'est-à-dire qu'elles ont des ressources de calcul de granularités différentes. Nous pouvons avoir par exemple dans la même architecture une partie grain fin pour traiter les manipulations binaires et des parties avec une granularité plus importante pour les calculs arithmétiques.

2.1.2 Granularité de la communication

Certaines architectures, vu le nombre de connexions, utilisent des réseaux de connexion locaux. Les connexions se font alors point à point. Ce type de réseaux facilite les communications locales en limitant le nombre de connexions possibles au niveau global.

La connectivité des modules est assurée par les matrices de connexions configurables. Ces dernières diminuent les performances pour des communications éloignées (en fréquence de fonctionnement et consommation de puissance) et nécessitent des outils de placement routage efficaces [96].

Dans les technologies actuelles, les délais des commutations sont bien inférieurs à ceux de la propagation des signaux le long des fils. Les connexions longues distances sont donc lentes et consommatrices d'énergie. La figure 2.1 montre une architecture grain fin utilisant un mode de connexion point à point.

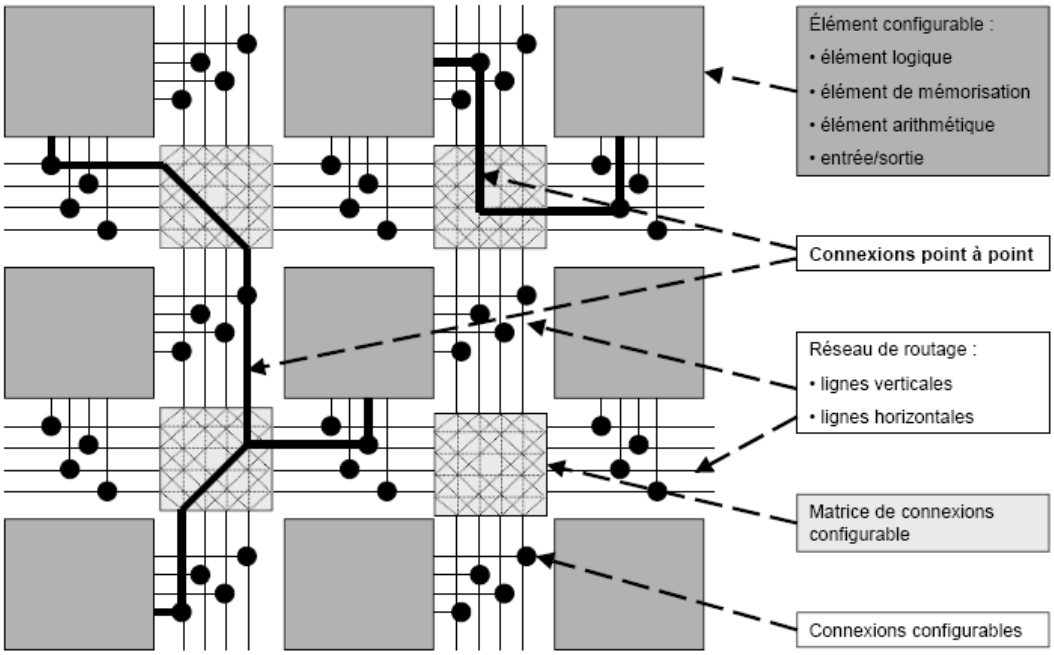


Figure 2.1: Exemple d'architecture utilisant une connexion point à point

2.1.3 Le couplage avec le processeur

Dans le cadre de la conception mixte, l'architecture peut contenir plusieurs cœurs basés sur différents modèles de traitement et qui participent à l'exécution de l'application. Les parties critiques d'une application sont identifiées puis affectées à un cœur de traitement dédié ou reconfigurable, afin d'en accélérer le traitement.

En tirant partie des propriétés respectives des systèmes programmables et des systèmes reconfigurables il est possible d'améliorer l'adéquation du système global avec l'application développée. C'est l'un des principes de l'approche "plateforme" qui privilégie la généricité et la flexibilité. Dans une approche mixte, le partitionnement de l'application influence le niveau de couplage entre le processeur et le co-processeur [97].

La finesse du partitionnement fixe le degré de communication nécessaire entre chaque partie.

Il existe trois modes de couplage d'une ressource reconfigurable avec un processeur. Le premier mode consiste à coupler la partie reconfigurable avec le processeur comme un périphérique. Les échanges s'effectuent par l'intermédiaire du bus d'entrées/sorties à l'aide, par exemple, d'accès DMA (Direct Memory Access). Le temps de latence est élevé, mais la bande passante disponible assure une alimentation efficace de la ressource reconfigurable. Les traitements peuvent s'effectuer en recouvrement avec ceux du processeur.

Il est possible aussi de coupler directement le processeur et l'accélérateur matériel reconfigurable via un bus local, c'est le second mode. Après reconfiguration, le reconfigurable manipule les données stockées dans la mémoire du processeur, sans intervention du processeur.

Le troisième mode de couplage est direct : le processeur et l'accélérateur sont intégrés sur la même puce de silicium. La ressource reconfigurable est placée dans les chemins de données internes du processeur ce qui minimise le temps de latence [98].

Ce dernier mode est issu des progrès technologiques en matière d'intégration. La famille Virtex proposant des FPGA à 8 millions de portes équivalentes, il est possible d'y placer de gros systèmes composés de plusieurs modules distincts.

L'intégration de systèmes numériques entiers est alors possible sur des structures entièrement reconfigurables.

Contrairement aux modes précédents, la ressource reconfigurable tend à inclure toutes les entités d'un système, la plupart basées sur des cœurs IP (Intellectual Property), tels que des microprocesseurs RISC par exemple. A titre d'exemple nous pouvons citer les microprocesseurs *MicroBlaze* développé par Xilinx [99] et *NIOS* développé par Altera [100], tous deux capables de délivrer une puissance de calcul de 125 MIPS.

2.1.4 La reconfigurabilité

Lorsque la reconfiguration de l'architecture ne peut intervenir qu'une seule fois sans remise en cause durant l'exécution de l'application, il s'agit de **reconfiguration statique**. La reconfiguration et l'exécution de l'application dans ce cas sont séparées dans le temps.

L'autre alternative de reconfiguration est **la reconfiguration dynamique**. Elle permet de modifier dans le temps la totalité ou une partie des éléments reconfigurables. Le but de la reconfiguration dynamique est alors d'optimiser la surface configurée dans le temps.

L'aspect reconfigurable d'une architecture offre plusieurs opportunités. En terme de performance, la reconfiguration permet d'adapter plus étroitement le composant matériel à la nature du traitement. En terme d'adaptabilité (flexibilité), elle permet de réagir à des événements en changeant de mode (mode basse consommation, mode dégradé, mode haute résolution…). Les événements peuvent venir de l'environnement (IMEC [101]), ou du système lui-même (des résultats intermédiaires de calcul, des valeurs des capteurs de température…).

En revanche, plus un circuit offre de possibilités de se reconfigurer, plus il contient de ressources pour gérer cette reconfiguration ce qui prend de la place sur le silicium et par conséquent plus il est lent.

Dans ce contexte les travaux cités dans [102] présentent un nouveau critère de classification des architectures reconfigurables qui s'appelle : La rémanence des architectures (R).

La rémanence R est définie comme le nombre de cycles d'exécution nécessaires à la reconfiguration complète de l'architecture.

Pour une rémanence donnée, les auteurs définissent l'ordre de grandeur du volume minimum de données qui doit être traité entre deux configurations pour garantir l'efficacité de l'architecture.

Le rapport entre le volume de données à traiter entre deux configurations et R guide le choix entre la mise en œuvre d'un parallélisme de données ou d'un parallélisme de traitement. Une faible valeur de ce rapport encouragera l'utilisation d'un parallélisme de traitement.

On remarque aussi que l'inverse de R mesure le caractère "dynamique" de l'architecture. Plus R est faible, plus l'architecture est reconfigurable dynamiquement et donc plus elle peut être considérée comme architecture à reconfiguration dynamique.

Pour la majorité des FPGA, la configuration ne peut se faire que globalement. Seuls les *Virtex* de Xilinx et les *FPSLIC* d'Atmel offrent des possibilités de reconfiguration partielle [103], où certaines parties du circuit peuvent être modifiées sans influer sur le fonctionnement du reste du circuit.

Dès lors, un nouvel axe de recherche a vu le jour concernant des systèmes d'ordonnancement de tâches matérielles [104] afin de pouvoir implémenter une application tout en minimisant l'espace nécessaire sur le FPGA. Un ordonnancement spécial permet de ne charger les contextes qu'au moment de leur utilisation, grâce à la reconfiguration partielle. Nous revenons sur ces aspects dans le chapitre 5.

Une approche souvent citée avec la reconfiguration partielle est l'utilisation de la reconfiguration multi-contextes [105]. Chaque contexte correspond à une configuration. La mémoire de contextes permet d'anticiper le prochain état de configuration de l'architecture. Le changement d'une configuration à une autre est alors accéléré.

Mais l'utilisation de cette technique exige le développement de méthode de gestion sophistiquée de contextes et de gestion de la reconfiguration [106].

L'architecture modulaire Ardoise [107], basée sur l'utilisation de circuits FPGA Atmel AT40K40, met en œuvre la reconfiguration dynamique par pré-positionnement de configurations. Cette architecture est principalement dédiée au domaine du traitement temps réel des images.

Le temps de passage d'une reconfiguration à une autre doit être très rapide pour permettre une utilisation efficace de la reconfiguration dynamique. Des projets de recherche industriels, comme ceux de Xilinx [108] ou de NEC [109], basés sur le principe du multiplexage temporel affichent des temps de l'ordre de 5 à 30 nanosecondes.

Dans le même sens, des architectures comme Garp [110], MorphoSys [111] ou Chameleon [112] ont opté pour l'utilisation de la notion de cache de reconfiguration. Elles consistent à stocker un nombre de configurations dans des buffers (généralement en mémoire locale du composant) et de permettre 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.

Nous assistons de nos jours à des nouvelles technologies qui minimisent considérablement les temps de reconfiguration qui peuvent devenir plus faible que les temps d'exécution des tâches.
Aujourd'hui, la plupart des architectures reconfigurables disposent d'un certains degré de reconfigurabilité. La possibilité de changer la configuration d'un circuit peut exister à divers niveaux : système, fonctionnel, opérateur et porte.
Pour chaque niveau nous donnons la définition et quelques exemples d'architectures.

2.1.4.1 Reconfiguration au niveau système

Les architectures reconfigurables au niveau système sont appelées aussi processeurs configurables. La figure 2.2 montre une architecture générique reconfigurable au niveau système.

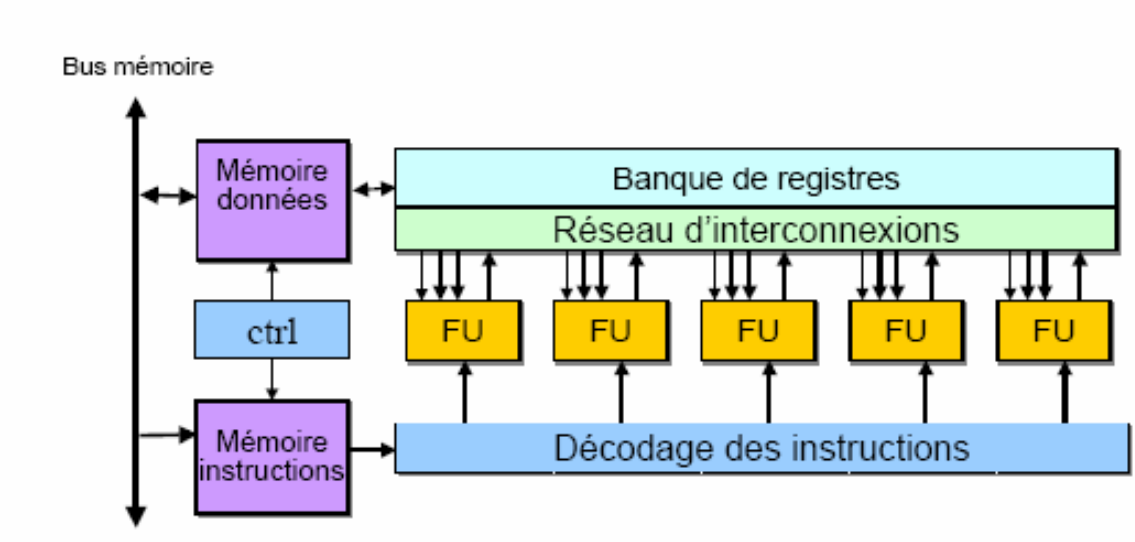


Figure 2.2 : Architecture générique reconfigurable au niveau système

Elle est constituée d'un ensemble de ressources travaillant sur une banque centrale de registres. Les opérations sont ordonnancées par un contrôleur unique.
Parmi ces architectures on trouve celles qui offrent la possibilité aux utilisateurs de spécialiser leur architecture en intégrant des unités de calcul spécialisées [113]. Les processeurs CARMEL [114] et ReAL [115,116], proposés respectivement par Infineon technologies et Philips, sont des exemples d'architecture supportant l'intégration de fonctionnalité dédiées.

Un autre exemple d'architecture reconfigurable au niveau système est le *S5000* proposé par Stretch Inc [117] au début de l'année 2004.

2.1.4.2 Reconfiguration au niveau fonctionnel

L'architecture générique de ce type est donnée dans la figure 2.3. Elle consiste en un ensemble de ressources de calcul dédiées ou programmables, reliées par un réseau d'interconnexion reconfigurable.

Dans la suite nous détaillerons quelques exemples d'architectures reconfigurables au niveau fonctionnel à partir de deux exemples académique et industriel.

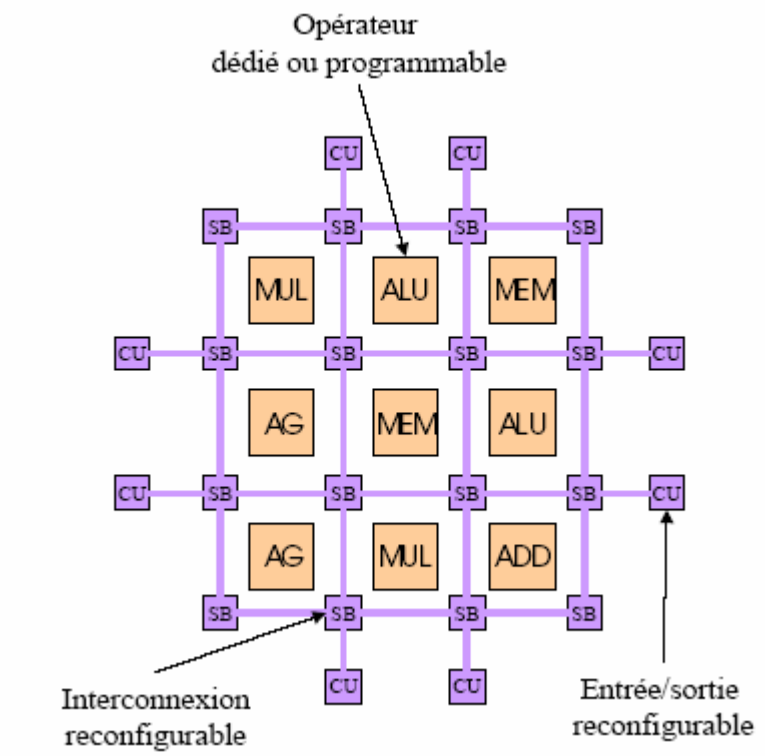


Figure 2.3 : L'architecture générique du reconfigurable au niveau fonctionnel

MorphoSys :

Développée à l'université de Californie, Irvine, USA, cette architecture [118] est à la pointe de la recherche dans le domaine des systèmes reconfigurables à grain large. Un processeur RISC, un réseau d'unités de traitement reconfigurables et une interface mémoire très élaborée sont regroupés sur un seul circuit VLSI. L'architecture est représentée sur la figure 2.4.

Chaque unité de traitement est constituée de deux multiplexeurs d'entrée, une ALU 16 bits avec multiplieur, un opérateur de décalage et quatre registres. Les unités sont configurées par un registre de contexte global de 32 bits.

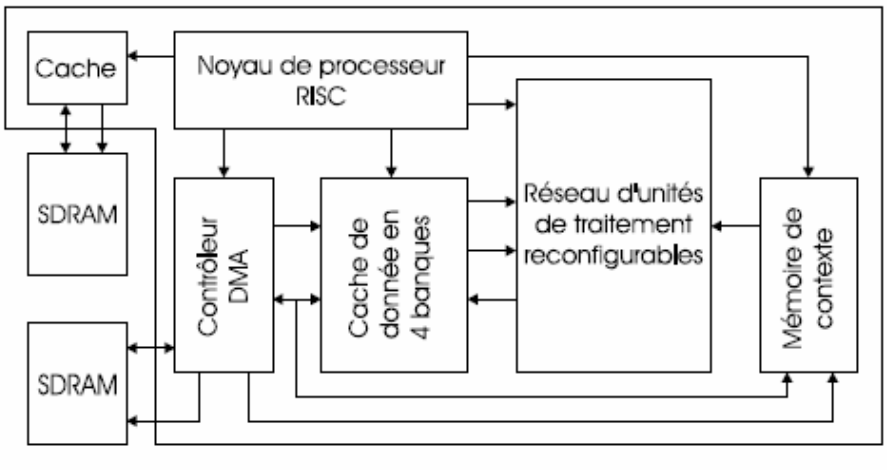


Figure 2.4 : L'architecture Morphosys

Chameleon

L'idée novatrice de Chameleon [119] est de regrouper sur un seul circuit un processeur ARC, un contrôleur PCI, un autre

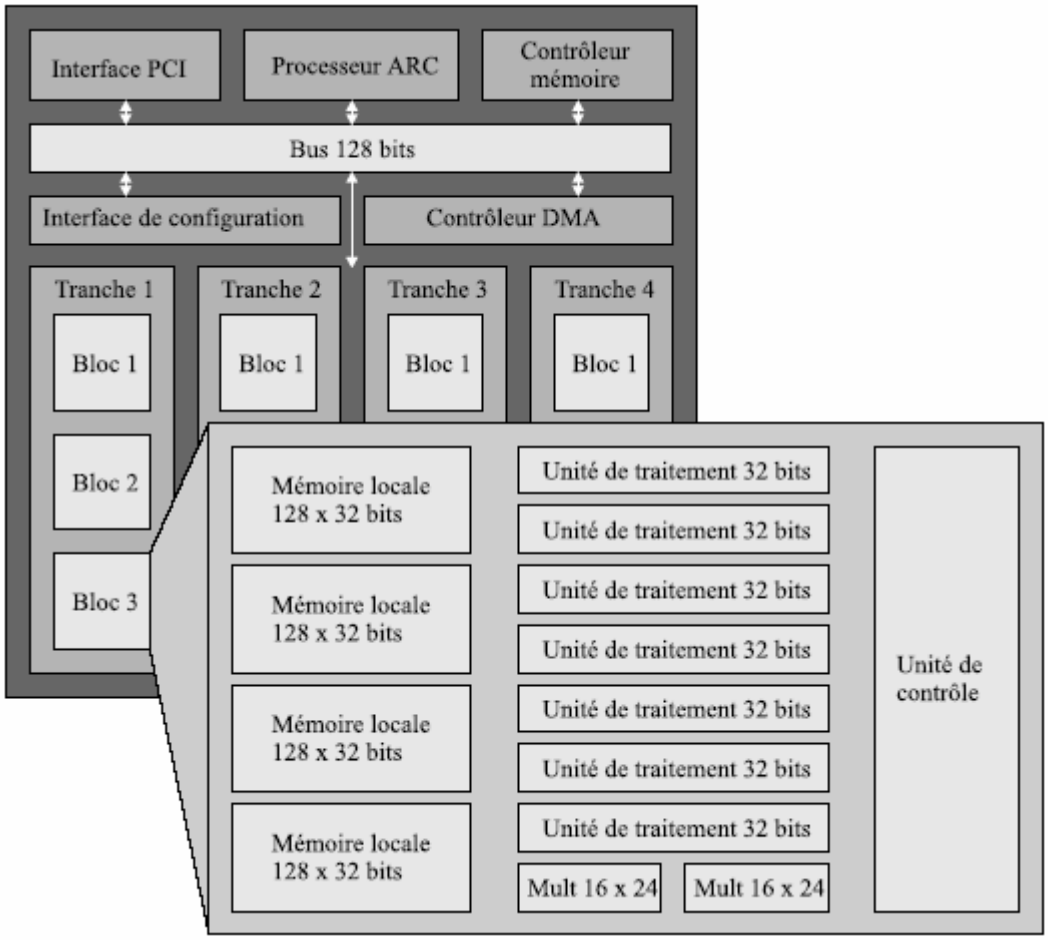


Figure 2.5 : l'architecture Chameleon

contrôleur de mémoire et un réseau d'unités de traitement reconfigurables (voir figure 2.5).

Plusieurs tailles de circuits sont disponibles. Par exemple, le CS2112 dispose de 84 unités de traitement, 24 multiplieurs et 48 mémoires locales. Un taux de transfert sur ses entrées-sorties de 2 Gbytes/s peut être atteint et la puissance de calcul maximale est de 24 Gop/s sur 16 bits.

Systolic Ring

Développée par le laboratoire LIRMM de l'université de Montpellier particulièrement pour le domaine de traitement du signal et des images, le Systolic Ring [120] adopte au niveau le plus haut de hiérarchie une topologie de communication linéaire en anneau. Cette structure favorise de part sa nature le

déroulement linéaire des calculs propres aux applications flot de données.
La figure 2.6 présente l'architecture en anneau de la couche opérative du Systolic Ring composée de Dnodes.

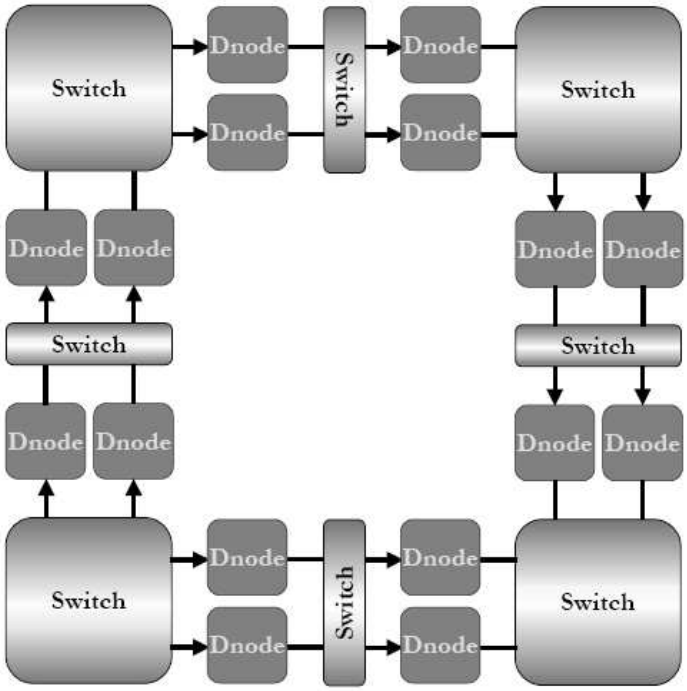


Figure 2.6: Architecture de la couche opérative de Systolic Ring

L'architecture complète du Systolic Ring a été décrite en VHDL. Une version à 8 Dnodes traitant des données sur 16 bits a été complètement simulée, prototypée et synthétisée. Le prototypage est fait sur la carte Altera APEX [121].

aSoC

L'aSoC (adaptive System on Chip) [122] est une architecture reconfigurable développée par le laboratoire VLSI Signal Processing Group (VLSI-SPG) de l'Université du Massachusetts au USA.

L'architecture est constituée de tuiles qui peuvent être de granularité différente. L'aSoC est très flexible puisque le concepteur peut définir les cœurs (IP) qu'il souhaite intégrer à son architecture.

Certaines tuiles peuvent être des tuiles reconfigurables de gros grain ou de grain fin. Les interfaces de communication de l'aSoC relient point à point les tuiles entre elles.

Les instructions de connexions contrôlent un crossbar qui peut établir des liaisons entre les quatre tuiles du voisinage (au nord, au sud, à l'est et à l'ouest) direct de la tuile dont il fait partie.

Le crossbar peut aussi créer une liaison entre une de ces tuiles de voisinage et le cœur avec lequel il peut communiquer. La figure 2.7 montre l'interface de communication d'une tuile avec le crossbar et la mémoire d'instructions. Sur cette figure nous trouvons aussi un dispositif permettant la gestion dynamique de la consommation de puissance de la tuile.

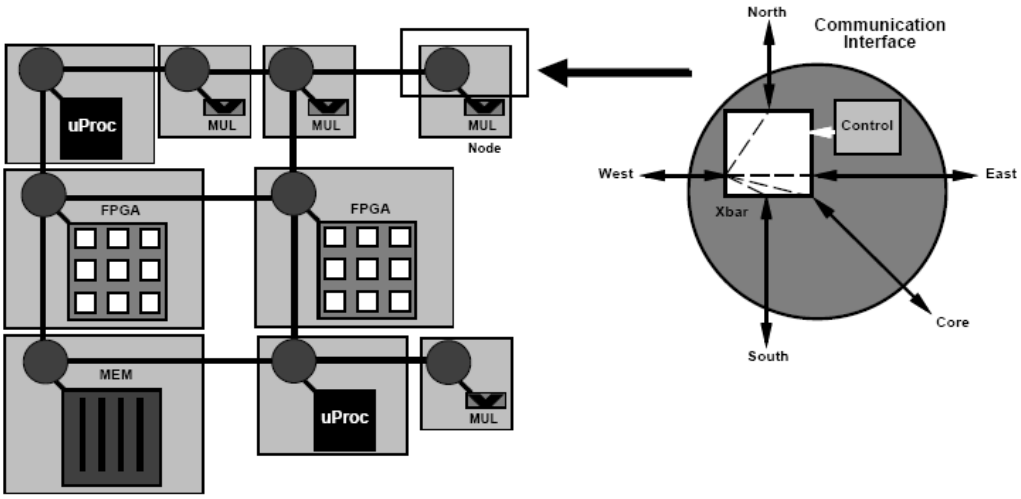


Figure 2.7: Architecture multi-tuiles de l'aSoC

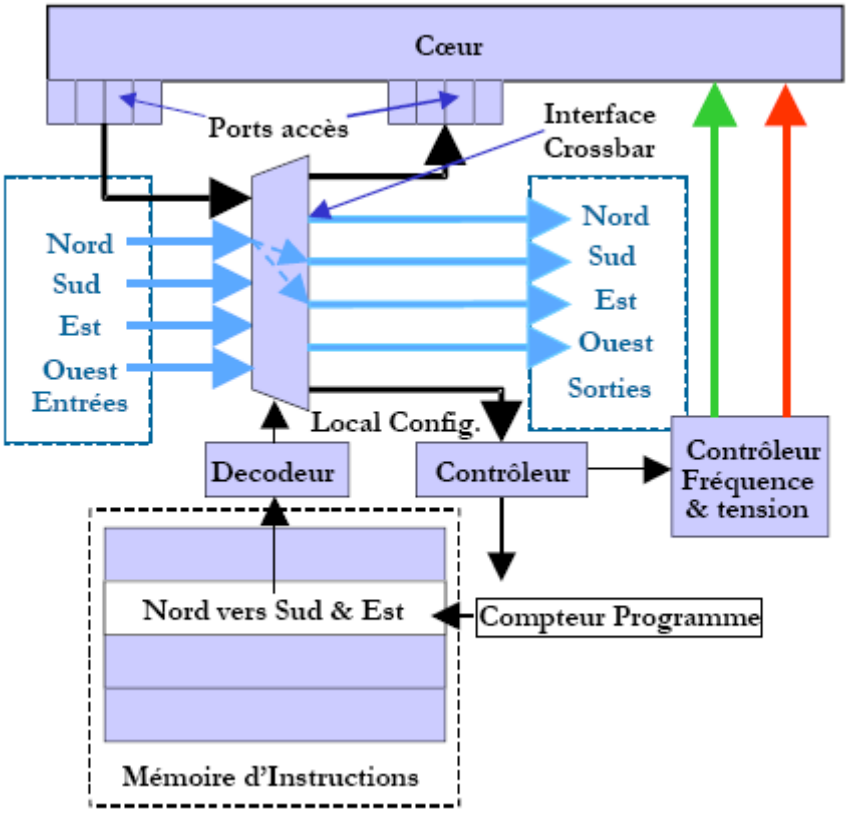


Figure 2.8: L'interface de communication d'une tuile dans un aSoC

L'architecture aSoC est adaptée aux applications de traitement du signal, et des images [123,124]. Dans [125] les auteurs proposent une interface de communication de l'aSoC en technologie 0.18 µm avec une fréquence de fonctionnement de 400 Mhz.

DART

L'architecture DART [126] est une architecture reconfigurable dynamiquement développée par l'IRISA de Rennes. Elle est conçue pour le traitement des applications mobiles de troisième génération.

Au niveau système, l'architecture DART (comme indiqué sur la figure 2.9) est composée d'un contrôleur de tâches qui est chargé d'affecter aux clusters hiérarchiques les différents traitements à exécuter. Une fois configurés par ce contrôleur, les clusters sont autonomes et gèrent leurs propres accès à la mémoire de données.

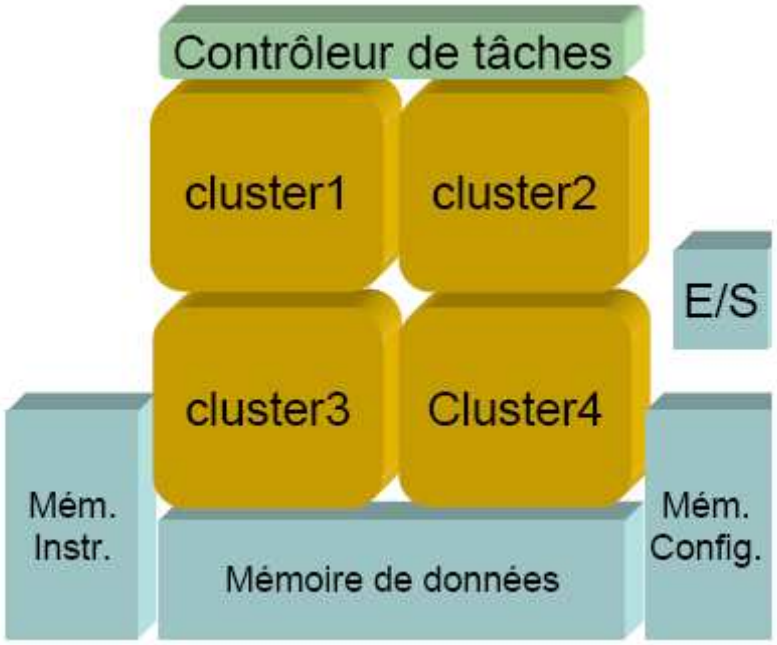


Figure 2.9 : Vue système de l'architecture DART

Un cluster de l'architecture DART se compose de 6 DPR (data Path Reconfigurable) reliés par un réseau local de communication, et un cœur de FPGA à grain fin.
L'architecture de CLUSTER de DART est schématisée sur la figure 2.10.
Les transmissions de données entre le FPGA (en tant que IP) et les DPRs se font via la mémoire des données du cluster.

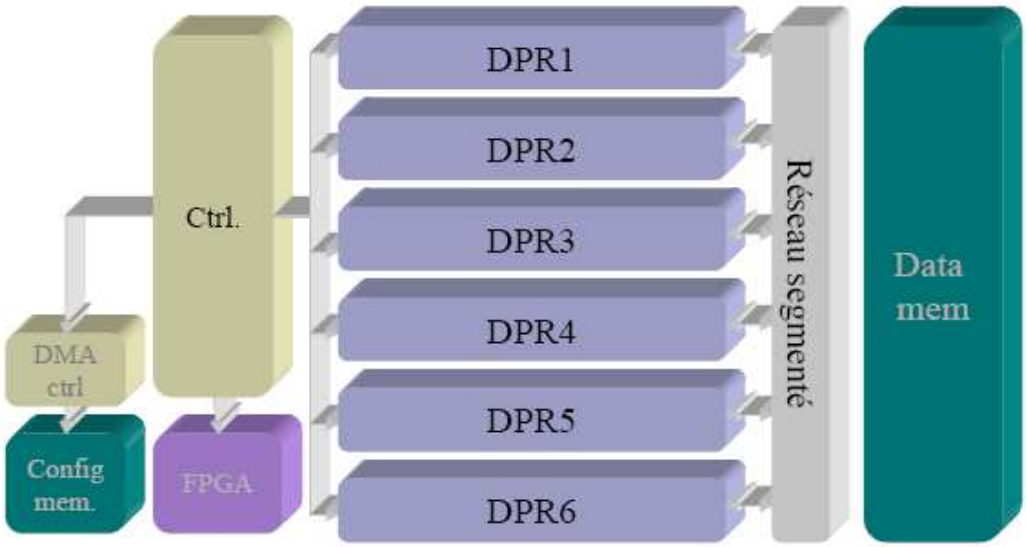


Figure 2.10: Architecture d'un Cluster de DART

2.1.4.3 Reconfiguration au niveau logique

Les architectures reconfigurables au niveau logique sont qualifiées de grain fin en raison de la faible largeur de leur chemin de données et du faible nombre d'opérations exécutées par chaque opérateur sur ses entrées. La reconfiguration opère sur des LUTs (Look-Up Table) et sur leurs interconnexions. La reconfiguration au niveau logique nécessite un très grand nombre de données de configuration.

Ces architectures sont principalement utilisées dans les domaines du prototypage et de l'accélération de traitement. La première application des architectures reconfigurables au niveau logique tels les FPGA était la validation des systèmes en vue d'intégration dans des circuits de type ASIC [127]. Pour ce but le reconfigurable est configuré une seule fois d'une façon statique durant toute la phase de la validation.

Aujourd'hui, ces architectures peuvent jouer le rôle d'accélérateur matériel qui permet de décharger le système hôte de traitements "jugés" critiques. De nombreuses applications en traitement du signal, d'image, de la vidéo, en cryptographie, … nécessitent des architectures massivement parallèles. L'utilisation des FPGA en tant qu'accélérateur matériel est un choix intéressant pour ces systèmes.

Plusieurs recherches ont été menées pour l'étude de plateformes constituées par un processeur hôte connecté à une ressource reconfigurable.

Le projet GARP proposé par le groupe de recherche BRASS (Berkeley Reconfigurable Architecture, Systems and Software) à Berkeley [128], le projet EPICURE (dont I3S fait partie) [106] et le projet NAPA proposé par National Semiconductor [129] ont étudié des solutions d'interfaçage du processeur hôte à la ressource reconfigurable. Le processeur peut être spécifique (le cas de NAPA et GARP) ou peut être un composant commercial (le cas d'EPICURE).

Le point faible de ces types d'architectures est le temps mis par la reconfiguration. Pour cela, plusieurs mécanismes de réduction de ce temps ont été envisagés. Les deux principales solutions retenues dans la littérature sont l'utilisation des circuits reconfigurables multi-contextes ou l'utilisation de la reconfiguration partielle.

La première solution a été adoptée dans le projet EPICURE. Par contre la reconfiguration partielle a été employé dans les projets GARP, NAPA et ARDOISE.

Nous nous intéressons dans notre travail aux applications de traitement d'images dont la charge de calcul des tâches est variable.

Nous ciblons donc une architecture reconfigurable dynamiquement composée par un processeur de type RISC connectée à une composante reconfigurable de type FPGA. Nous optons pour des outils de partitionnement logiciel/matériel dynamique en ligne pour pouvoir adapter l'architecture aux besoins de traitement. Nous proposons dans la section suivante quelques travaux effectués dans ce contexte.

2.2 Exemples de Travaux

Très peu de travaux dans la littérature ont abordé le problème des architectures logiciel/matériel permettant des migrations de tâches dynamiques en ligne. Nous exposons dans la suite les résultats de deux équipes de recherche qui ont proposé des solutions architecturales à ce problème.

2.2.1 Travaux à l'IMEC

Dans [130] les auteurs présentent une architecture qui permet la réallocation des tâches entre les cibles logicielles et matérielles. Le principe de la communication après avoir migrer d'une implémentation à une autre est présenté. Un système d'exploitation embarqué a été utilisé pour gérer les différentes communications, les sauvegardes de contextes, le placement/routage et la gestion de la mémoire.

L'architecture proposée est basée sur le principe de Network-on-chip logiciel. L'ICN (InterConnections Network) divise le reconfigurable sur des tuiles de même taille et même forme. Chaque tuile peut exécuter une seule tâche à la fois. Pour rendre la communication transparente par rapport aux implémentations des tâches, un système d'exploitation spécifique appelé OS4RS a été conçu.

La figure 2.11 montre le principe de la communication entre les tâches sur l'architecture proposée par l'IMEC.

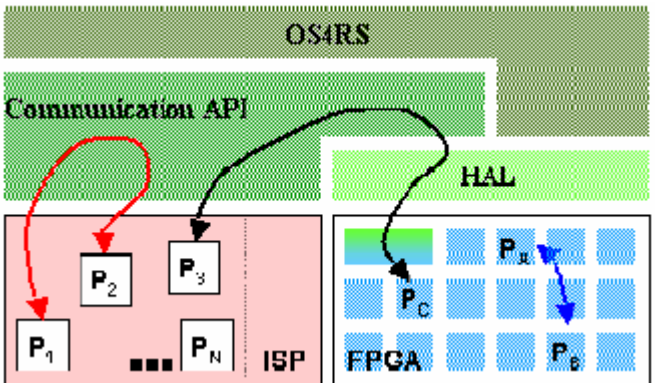


Figure 2.11 : Principe de la communication entre tâches

Le placement/routage entre les tuiles est contraint par les emplacements des tâches dans le reconfigurable. C'est le concepteur lui-même qui peut fixer ces contraintes. Le fait que l'OS4RS ait l'information sur l'utilisation des tuiles à chaque instant réduit le temps de placement. Le remplacement du contenu d'une tuile par une nouvelle tâche correspond à une reconfiguration partielle à laquelle on associe un *bitstream* partiel lié à la tâche. Le routage d'une nouvelle tâche revient à mettre à jour un tableau de routage à chaque fois qu'une tâche est ajoutée/supprimée au/du reconfigurable.

La librairie OCAPI-xl [130] permet une représentation unifiée des tâches logicielles et matérielles.

2.2.2 Travaux au CECS (Université de Californie Irvine)

Les travaux présentés dans [131] proposent une architecture formée par de la logique reconfigurable destinée pour le partitionnement logiciel/matériel dynamique (voir paragraphe 1.1.6.3 du chapitre1). Le but principal lors de la conception de cette architecture est de minimiser au mieux le temps de placement routage effectué en ligne sur le reconfigurable.

L'idée est de concevoir une logique reconfigurable sur mesure (dédiée). Cette logique reconfigurable permet:

- l'accélération de l'exécution des boucles en ajoutant au reconfigurable un module qui calcule les adresses des données pour faciliter les accès mémoire (Data Adresses Generators : DADG) et un module de contrôle de boucle (Loop Control Hardware : LCH)
- la réduction du temps d'exécution de placement et routage de multiplieurs en ajoutant un accumulateur/multiplieur 32 bits
- de contenir une logique configurable simple : SCLF (Simple Configurable Logic Fabric) constituée par une série de CLB (Combinational Logic Blocks) entourés par des matrices de commutation (SM : Switch matrices) pour le routage entre CLB. Chaque CLB est connecté à une seule matrice avec laquelle il échange ses entrées et ses sorties. Chaque matrice peut router les signaux vers ses 4 matrices voisines ou vers des matrices plus éloignées.

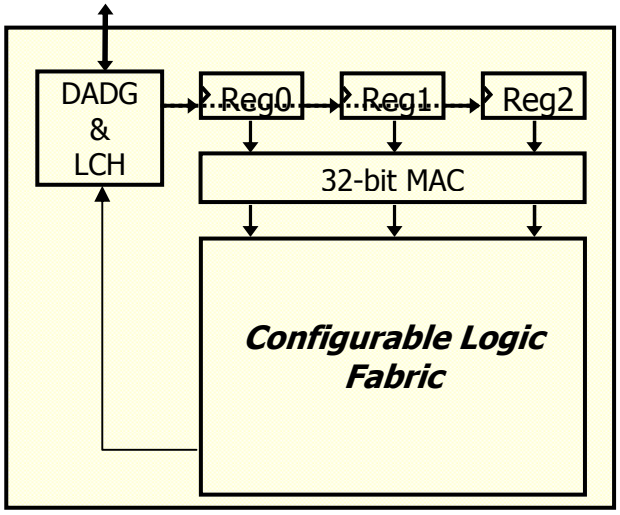


Figure 2.12: L'architecture WCLA (Warp-processor Configurable Logic Architecture)

Conclusion

Nous avons présenté dans ce chapitre un état de l'art des architectures reconfigurables dynamiquement ainsi que leurs caractéristiques permettant de les classer.

Il faut noter que l'aspect architecture n'est pas réellement abordée dans la thèse mais que les travaux relatives à l'IMEC, CECS, et EPICURE nous a convaincu que des études d'architectures devraient prochainement déboucher sur des systèmes reconfigurables permettant/ facilitant la migration de tâches entre les entités reconfigurables dont fait partie les processeurs. De telles architectures n'ont un intérêt que si des méthodes efficaces sont développées pour décider des migrations (partitionnement) en ligne.

On peut aussi souligner que le couplage Architecture/ OS/ Méthode de migration/ Classe d'application devra être considéré de façon précise pour arriver à des solutions efficaces.

Le chapitre suivant donne une présentation générale de notre méthodologie de partitionnement dynamique en ligne.

CHAPITRE 3

Présentation générale de partitionnement logiciel/matériel en ligne

Introduction

De nombreuses applications, en particulier en traitement des images, ont des temps d'exécution fonction de la nature des données à traiter. Les techniques classiques de partitionnement statique [18,71,72,74,75,76,78,132], qui déduisent une solution à partir d'une analyse des caractéristiques (temps d'exécution, coûts) des fonctions de l'application exprimées sous forme de constantes ne sont pas adaptées. En effet, elles sont basées sur une approche conservative de l'évaluation des temps d'exécution avec souvent pour conséquence un pessimisme excessif qui conduit à sur-dimensionner les architectures produites.

Nous présentons dans ce chapitre une nouvelle méthode de partitionnement logiciel/matériel dynamique. Avant de décrire globalement notre flot de partitionnement, nous commençons par définir le modèle des applications ciblées, la fonction coût et l'architecture cible. En effet une méthode de partitionnement s'applique sur des modèles, ceux de l'application et de l'architecture cible, en vue d'optimiser une fonction de coût.

3.1 Modèle d'application choisi

Des nombreuses applications, en particulier en télécommunication et multimédia, ne nécessitent pas des réalisations temps réel strict. Pour ces domaines d'application, des implémentations visant à obtenir une qualité de service adaptée aux besoins sont donc suffisantes dans de nombreux cas (temps réel souple ou mou).

Les applications de traitement du signal et des images font apparaître des traitements intensifs sur des structures de données régulières au niveau des données brutes et plus irrégulières lorsque les informations deviennent plus abstraites. Souvent ces applications peuvent être représentées par des graphes de flots de données (*DFG: Data Flow Graph*), formalisme bien adapté, au moins dans les niveaux bas et moyen de traitement pour décrire leur comportement, indépendamment d'une implémentation logicielle ou matérielle. La description initiale du système est donc un graphe dirigé sans cycle. Les noeuds du graphe représentent les traitements (calculs) et les arcs du graphe représentent les dépendances de données (voir figure 3.1).

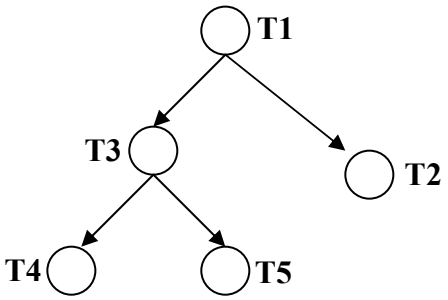


Figure 3.1: Graphe de Flots de Données (DFG: Data Flow Graph)

A chaque nœud terminal on associe une contrainte de temps maximum (identique pour tous les nœuds terminaux) qui correspond à la période d'activation du graphe.
A partir de ce modèle, le partitionnement consiste à déterminer pour chaque nœud son type d'implantation (Processeur, ASIC, FPGA …) son allocation (numéro de la ressource du type choisi, s'il y en a plusieurs) et son ordonnancement. Ce partitionnement doit vérifier les contraintes sur le temps de calcul entre nœuds initiaux et terminaux du graphe. De même, les communications entre les nœuds doivent être prises en compte dans le partitionnement aussi bien dans le calcul du temps d'exécution total que dans celui des ressources nécessaires à leur réalisation.

Il est à noter ici que le niveau de granularité intervient comme paramètre primordial dans le choix de l'architecture cible et aussi dans les résultats de partitionnement. En effet, une granularité fine permet d'expliciter plus de parallélisme et plus d'opérations, et à l'inverse, une granularité élevée diminue la complexité du problème.

Dans le cadre de nos travaux, nous choisissons une granularité au niveau fonction. La corrélation entre les temps d'exécution des fonctions et les caractéristiques des données traitées peut être facilement caractérisée à ce niveau. A un niveau de granularité plus élevé, cette corrélation devient plus complexe à mettre en évidence vue l'accroissement du nombre de paramètres qui rentrent en jeu dans l'évaluation des temps d'exécution. A un niveau de granularité très fin la corrélation devient de plus en plus absente du fait du faible nombre de données et d'opérations considéré mais dans ce cas la complexité du problème du partitionnement est telle qu'il est impossible d'effectuer un partitionnement en ligne.
La méthode de partitionnement vise à optimiser une fonction de coût, celle retenue dans notre étude est présentée dans le paragraphe suivant.

3.2 Fonction coût

La fonction coût permet de mesurer la qualité d'une solution fournie par le partitionnement et de guider l'algorithme de partitionnement vers une meilleure solution. La fonction coût peut être une combinaison de plusieurs métriques (temps d'exécution, consommation, surface…).

Dans le problème de partitionnement étudié dans notre travail l'objectif est le respect de la contrainte temps réel souple. Il s'agit donc de partitionner une application temps réel mou en respectant la contrainte temporelle globale tout en exploitant un nombre de ressources matérielles minimal.
La fonction coût est donc de garder au maximum le caractère auto-adaptatif de l'architecture: rendre les ressources disponibles autant que faire se peut pour pouvoir réagir à des augmentations des temps d'exécutions.

3.3 Modèle d'architecture cible

L'architecture générique utilisée dans notre approche est la même que celle considérée dans [73]. Elle est composée d'un processeur connecté à une unité de calcul reconfigurable (UCR) à travers une interface intelligente conçue par le CEA[1] et appelée *ICURE*[2] [106]. L'architecture de base est donnée par la figure 3.2.

L'interface ICURE contient plusieurs contextes qui sont autant des possibilités de reconfigurations partielles de l'UCR.
Chaque contexte est automatiquement chargé sur l'UCR dès que le processeur demande son exécution. Un automate dans ICURE assure cette gestion transparente pour le processeur.

L'interface contient également une matrice de connexion, commandée par cet automate, qui permet de mettre en relation les entrées/sorties de la fonction placée sur l'UCR et les entrées/sorties du processeur.
Cette interface intelligente est un élément important dans l'implémentation d'un partitionnement en ligne dans lequel il s'agit de migrer des tâches d'une implémentation à une autre.

Le processeur embarqué considéré dans [73] peut être d'un type quelconque à condition de permettre un couplage efficace avec l'UCR par l'intermédiaire d'*ICURE*. Dans l'approche de partitionnement basée sur les WCET[3] [73] le processeur doit avoir un comportement déterministe afin de permettre une optimisation d'un point de vue ressources et consommation d'énergie. Ceci n'est plus une obligation dans notre cas du fait que le partitionnement opère en ligne sur des temps d'exécution variables.

[1] Commissariat de l'Energie Atomique
[2] Intelligent Control Unit for Reconfigurable Element
[3] Worst Case Execution Time

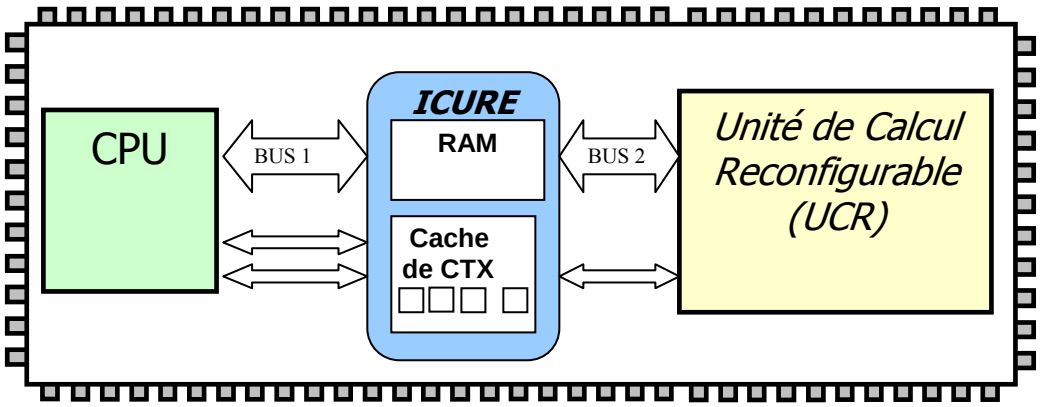


Figure 3.2 : l'architecture générique cible

Le reconfigurable doit supporter la réallocation dynamique des tâches. Il doit assurer un mécanisme de communication entre les tâches qui peuvent changer leurs implémentations au cours de l'exécution de l'application. Pour garantir cette flexibilité dans la gestion du reconfigurable, plusieurs travaux tels que ceux cités dans le paragraphe 3.1 de ce rapport [134,135] ont proposé d'ajouter une couche d'abstraction au dessus du FPGA. Cette approche consiste à effectuer un pré-partitionnement de la logique reconfigurable en un nombre fixe de tuiles de même taille et à implémenter une infrastructure de communication entre ces dernières.

Cette couche d'abstraction facilite la gestion des ressources matérielles reconfigurables en offrant la possibilité d'effectuer le placement des tâches en ligne (c'est à dire pendant l'exécution de l'application) et de contrôler la communication entre les tâches.

L'inconvénient majeur de l'utilisation de surfaces reconfigurables pré-partitionnées est qu'il s'agit d'un nombre constant des tuiles de taille fixe. En effet, lorsque la taille de la tuile ne correspond pas à la taille des ressources matérielles nécessaires pour l'implémentation de la tâche, un certain nombre de ressources non utilisées est perdu (ceci est appelé fragmentation interne [136]). De plus, le temps de reconfiguration ainsi que la taille du bitstream partiel sont liés à la taille de la tuile et non pas à celle de la tâche matérielle.

Dans le cas où la taille de la tuile est inférieure à la taille de la tâche, le concepteur est obligé de diviser les tâches matérielles sur plusieurs sous-tâches de tailles plus petites de telle façon que chaque sous-tâche puisse occuper une tuile.

Cette dernière solution complexifie l'ordonnancement et le partitionnement en ligne de l'application à cause du coût associé au placement/routage à effectuer en ligne pour prendre en compte une migration de tâche.

Une solution adoptée par le groupe de travail à l'IMEC[1] est d'utiliser un système d'exploitation temps réel adapté au système reconfigurable OS4RS (Operating Systems for Reconfigurable Systems) [130]. Lorsque la taille de la tuile est supérieure à la taille d'une tâche, l'OS4RS, associé à une couche d'abstraction pour le multiplexage, permet de placer plusieurs tâches dans la même tuile reconfigurable. La couche de multiplexage sert à aiguiller les messages de données vers une tâche ou vers une autre au sein de la même tuile (voir figure 3.3).

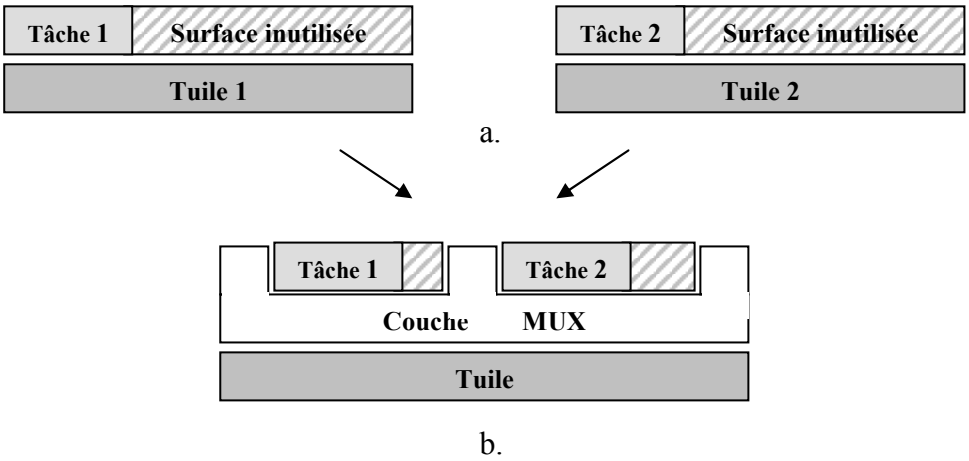


Figure 3.3 : L'utilisation d'une couche d'abstraction pour réduire les surfaces du reconfigurable inutilisée

3.4 Flot global de Partitionnement/Ordonnancement en ligne

La motivation principale pour ces travaux a pour origine la constatation de la dépendance de la charge de calcul de certaines applications par rapport aux données traitées. Par exemple, le temps d'exécution de la fonction : extraction de contours des objets dans une image dépend du nombre et de la taille des objets dans l'image.

En implémentant de telles fonctions sur une architecture hétérogène et en considérant des temps d'exécution pire cas, le concepteur risque fort de surdimensionner son architecture et donc de ne pas l'exploiter efficacement. Nous proposons ici une nouvelle méthodologie de partitionnement/

[1] Institut of Micro-technologie

ordonnancement en ligne qui permet d'adapter l'allocation des ressources en fonction des évolutions des temps d'exécution des tâches.

3.4.1 Présentation du problème

Traditionnellement, dans le cas des applications à contraintes temps réel strict le partitionnement opère sur des temps d'exécution des tâches correspondant aux pires des cas (WCET).

L'utilisation des WCET dans le cas d'une application à contraintes temps réel mou (devant tendre à respecter des contraintes dont le non-respect temporaire engendre au plus un dysfonctionnement partiel qui ne remet pas en cause la mission) engendre souvent un surdimensionnement en ressources de l'architecture lié à un pessimisme excessif. En effet, les calculs de WCET ont tendance à être par nature pessimiste du fait de décisions à caractère conservatif mais le pessimisme est également dû au fait que pour toutes les tâches on considère leur WCET alors qu'il peut exister des corrélations entre les temps d'exécution des tâches.
Comme le montre la figure 1.2, le temps d'exécution d'une tâche est souvent très loin de la valeur de WCET. Les données statistiques prises sur plusieurs tâches à temps d'exécutions variables et pour des séquences de test différentes illustrent sur ces exemples que les temps d'exécution WCET sont très rarement atteints.
Ceci nous montre la nécessité d'une approche de partitionnement capable de considérer des temps d'exécution variables. En considérant que des architectures dynamiquement et partiellement reconfigurables vont émerger (voir état de l'art des architectures dans le chapitre 2) il apparaît envisageable d'allouer dynamiquement des ressources aux tâches en fonction des évolutions des temps d'exécution des tâches.
Dans le cadre du partitionnement logiciel/matériel d'une application en ciblant une architecture hétérogène, le problème du temps d'exécution variable se présente au niveau modèles d'implantation (temps d'exécution en fonction du nombre de ressources). En effet les résultats des estimations des temps d'exécution des tâches d'une application conduit généralement à affecter un temps d'exécution correspondant au pire des cas pour un nombre de ressources constant (voir figure 3.4)

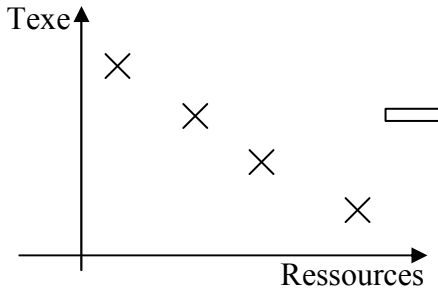


Figure 3.4 : courbe d'implantations avec les WCETs

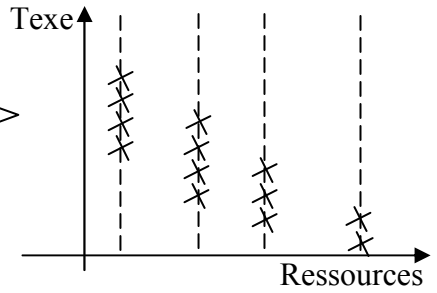


Figure 3.5 : courbe d'implantations avec les estimations des temps d'exécutions

Ceci n'est pas forcement la représentation la plus fidèle de la réalité, car à l'exécution nous obtenons des temps d'exécution généralement inférieurs à l'ensemble des WCET (voir figure 3.5).

3.4.2 Flot global d'une approche de partitionnement dynamique

Pour illustrer le principe de la méthode proposée, nous prenons l'exemple d'un DFG constitué de trois tâches, présenté sur la figure 3.6.

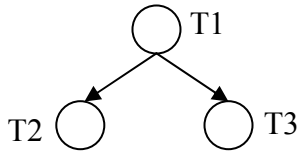


Figure 3.6 : Exemple simple d'un DFG de trois tâches

Dans un premier temps nous négligeons les temps de communication et les temps de reconfiguration des tâches. Les temps de communication peuvent être ajoutés aux temps d'exécution des tâches sans changer considérablement la méthode de partitionnement. Le temps de reconfiguration dépend de la technologie du reconfigurable.

Nous supposons que chaque tâche de l'exemple peut avoir quatre implémentations différentes : une implémentation logicielle et trois implémentations matérielles (sur le reconfigurable).

Les données d'implémentations temps d'exécution logiciel: (Tsw) et matériel (Thw) ainsi que le nombre des ressources matérielles utilisées: (Ress en CLB[1]) pour chaque implémentation sont regroupées dans le tableau suivant :

	Tsw	**Thw1**	**Ress1**	**Thw2**	**Ress2**	**Thw3**	**Ress3**
T1	**21**	**17**	**100**	**10**	**150**	**8**	**200**
T2	**28**	**12**	**150**	**10**	**200**	**9**	**250**
T3	**29**	**20**	**290**	**15**	**300**	**10**	**350**

Toutes les valeurs du temps d'exécution indiqués dans le tableau ci-dessus sont en ms, les nombres de ressources sont en CLB).

La contrainte temporelle sur le temps d'exécution total est par exemple de 40 unités de temps.
Nous prenons l'hypothèse que le temps d'exécution de la tâche T1 est constant et que les temps d'exécution des autres tâches sont variables (les valeurs données pour les tâches dans le tableau 1 sont des WCET).

[1] Configurable Logic Bloc

Nous supposons aussi que la tâche T1 garde toujours une implémentation logicielle.

Le temps d'exécution total pour cet exemple se calcule par la formule suivante :

$$Texe_{total} = Texe\ (T1) + Max\ (Texe\ (T2)\ ,\ Texe\ (T3))$$

La surface totale du reconfigurable utilisée est la somme des ressources utilisées par les tâches.

$$Surface_{totale} = Ress\ (T1) + Ress\ (T2) + Ress\ (T3)$$

Un partitionnement optimal en surface basé sur les WCET et respectant la contrainte temporelle est donné par le tableau suivant. Ce partitionnement est calculé hors ligne et donne un temps d'exécution de 36 unités de temps et une utilisation de 450 ressources

Tâche	Implémentation	Temps d'exécution	Ressource
T1	**SW**	**21**	**0**
T2	**HW1**	**12**	**150**
T3	**HW2**	**15**	**300**

Admettons que les distributions des temps d'exécution des tâches T2 et T3 respectivement sur les unités HW1 et HW2 soient celles de la figure 3.7 et que ces temps ne soient pas corrélés entre eux.

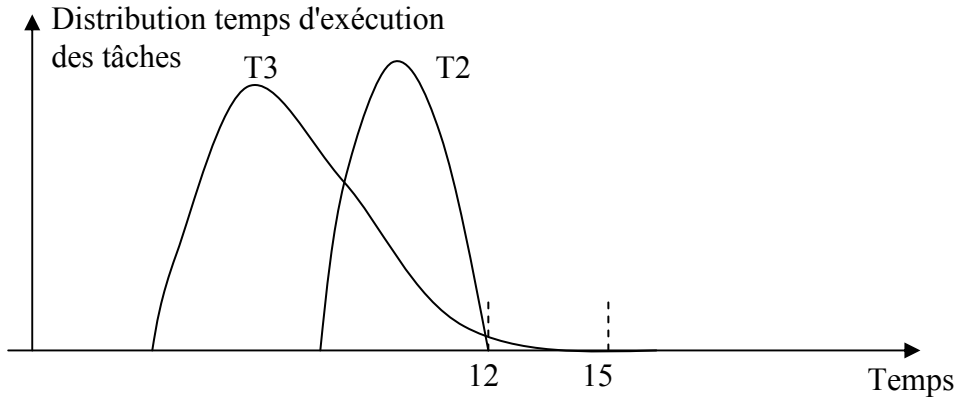


Figure 3.7: Exemple de distribution de temps d'exécution des tâches

Avec une telle distribution, c'est majoritairement la tâche T2 qui impose les temps maximum d'exécution du DFG (voir figure 3.8).

Par exemple à l'itération i (une itération est une instance d'exécution, par exemple le traitement d'une image) les temps d'exécution mesurés des tâches sont les suivants :

Tâche	Temps d'exécution	Implémentation
T1	**21**	**SW**
T2	**6**	**HW1**
T3	**4**	**HW2**

Nous représentons sur la figure 3.8 le schéma d'exécution sur l'architecture cible composée du processeur et du reconfigurable:

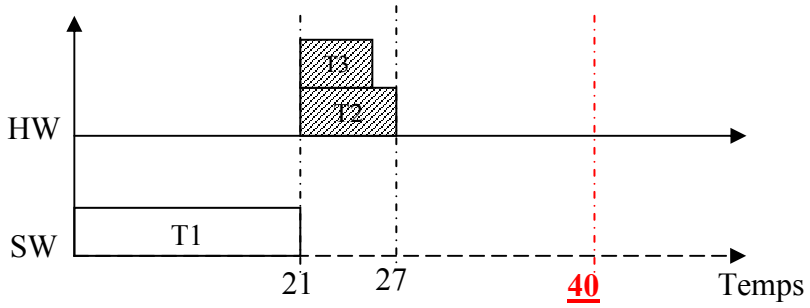


Figure 3.8 : Schéma d'exécution de l'itération i

Les temps d'exécution des deux tâches T2 et T3 varient d'une itération à une autre mais le temps d'exécution total reste majoritairement très inférieur à la contrainte temporelle.

Par conséquent, dans un grand nombre d'itérations les ressources matérielles sont utilisées d'une façon inefficace. Un surdimensionnement de l'architecture est causé par le pessimisme excessif des hypothèses de partitionnement basées sur les valeurs des WCET.

La solution que nous proposons consiste à changer l'implémentation d'une tâche dynamiquement c'est-à-dire en cours d'exécution de l'application pour allouer au mieux les ressources du reconfigurable en fonction des évolutions des temps d'exécution.

Une première approche pourrait consister à utiliser au mieux les ressources du reconfigurable pour allouer dynamiquement l'ensemble des tâches de l'application sur le processeur et le reconfigurable. En d'autres termes, à chaque fois que nécessaire, par exemple quand un dépassement des contraintes est observé ou prédit, un partitionnement complet des tâches est réalisé. Ce partitionnement aurait pour objectif d'effectuer l'allocation et l'ordonnancement sous contraintes de temps et de ressources. La complexité d'un tel

partitionnement ne peut être considérée dans un système qui vise à s'auto-adapter dynamiquement, pendant l'exécution de l'application.

La deuxième approche consiste à effectuer des ajustements de l'allocation en fonction de l'observation des évolutions des paramètres de l'application qui influent sur le temps d'exécution. Cette approche, beaucoup plus simple, présente l'avantage de n'effectuer que des migrations des tâches à partir d'un état courant afin d'éviter des dépassements de contraintes pour les prochains traitements. L'inconvénient est que ces migrations constituent autant de choix locaux qui peuvent conduire à un état peu cohérent en terme d'adéquation de l'utilisation des ressources par rapport aux traitements. C'est cependant cette deuxième approche qui est considérée dans la suite. Pour éviter le problème soulevé par ces migrations successives des tâches, nous proposons de gérer le reconfigurable de telle sorte qu'autant que faire se peut un volant de ressources libres soit constitué afin de disposer d'un budget ressources quand l'augmentation du temps d'exécution impose d'accélérer des traitements.
Pour réaliser cet ensemble de ressources libres nous considérons un seuil (appelé seuil bas) sur le temps d'exécution qui, lorsqu'on est en deçà, indique qu'il n'y a pas d'urgence par rapport à la contrainte. Dans ce cas, on cherche à libérer des ressources, sachant qu'on dispose justement de temps pour effectuer cette migration du reconfigurable vers le processeur.

Inversement, nous considérons également un seuil haut sur le temps d'exécution qui, lorsqu'il est atteint ou dépassé conduit à migrer une ou plusieurs tâches du processeur sur le reconfigurable pour ramener le temps d'exécution en dessous de la valeur du seuil. Cette opération de migration est normalement facilitée si on dispose des ressources qui ont été préalablement libérées lorsque le temps d'exécution était en dessous du seuil bas.

En appliquant cette méthode sur l'exemple précédent, nous fixons par exemple un seuil bas du temps d'exécution à 30 unités de temps, soit 75% de la valeur de la contrainte temporelle.

A la fin de l'itération i nous avons un temps d'exécution total égal à 27 unités de temps, donc un partitionnement intervient pour alléger le reconfigurable et migrer des tâches qui ne sont plus critiques vers le processeur.

La tâche T3 est implémentée sur HW2, donc nous avons le choix entre une réallocation vers HW1 ou vers le processeur. Le premier choix n'est pas intéressant car il ne libère que 10 ressources. Par ailleurs, le deuxième choix risque de faire augmenter le temps d'exécution total au dessus de la contrainte temporelle.

La tâche T2 est implémentée sur HW1. En la mettant sur le processeur, le nombre de ressources matérielles libérées est égal à 150, tout en respectant la contrainte temporelle.

L'exécution de l'application avec ce nouveau résultat de partitionnement suit le schéma d'exécution ci-après, avec les temps d'exécution de l'itération i.

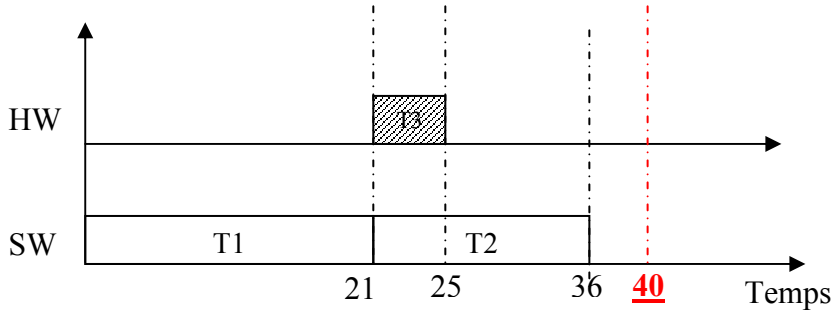


Figure 3.9 : Schéma d'exécution après réallocation dynamique de la tâche T2

Avec cette configuration le système n'utilise que 300 ressources et respecte la contrainte temps réel. Après k itérations les temps d'exécutions des tâches T2 et T3 sont égaux à 17 et 12 respectivement. La valeur du seuil haut est fixée à 35 unités de temps. La valeur du seuil étant dépassée il devient urgent de migrer des tâches vers le reconfigurable pour éviter un dépassement des contraintes.

D'après cet exemple simple, nous pouvons tracer les grandes lignes de la méthodologie de partitionnement adaptatif en ligne.

1. Le partitionnement opère en se basant sur les mesures des temps d'exécution des tâches de l'application. L'objectif est que le système ait la capacité de prédire les interventions du partitionnement pour laisser le temps à ce dernier de chercher la meilleure solution d'allocation et d'ordonnancement. Nous introduisons donc une étape de prédiction de performances.
   La méthode de prédiction/ estimation fera l'objet de notre prochain chapitre dans ce rapport.

2. L'algorithme de partitionnement s'exécute en ligne aussi. Il doit résulter d'un compromis entre efficacité et rapidité pour éviter de pénaliser les performances tout en fournissant des solutions satisfaisantes. Nous présentons cette approche de partitionnement dans le chapitre 5.

3. Pour évaluer en ligne une solution de partitionnement, nous avons besoin d'un algorithme d'ordonnancement qui retourne le temps d'exécution total estimé de la solution proposée par le

partitionnement. L'heuristique d'ordonnancement en ligne est développée dans la deuxième partie du chapitre 5 de ce rapport.

La figure 3.9 ci-dessous rappelle la comparaison entre un flot de conception classique basé sur une méthode de partitionnement statique hors ligne et notre approche de partitionnement dynamique en ligne. On peut remarquer que nous avons déplacé dans le flot la position de l'étape de partitionnement/allocation/ordonnancement pour la placer au plus proche de la structure d'exécution.

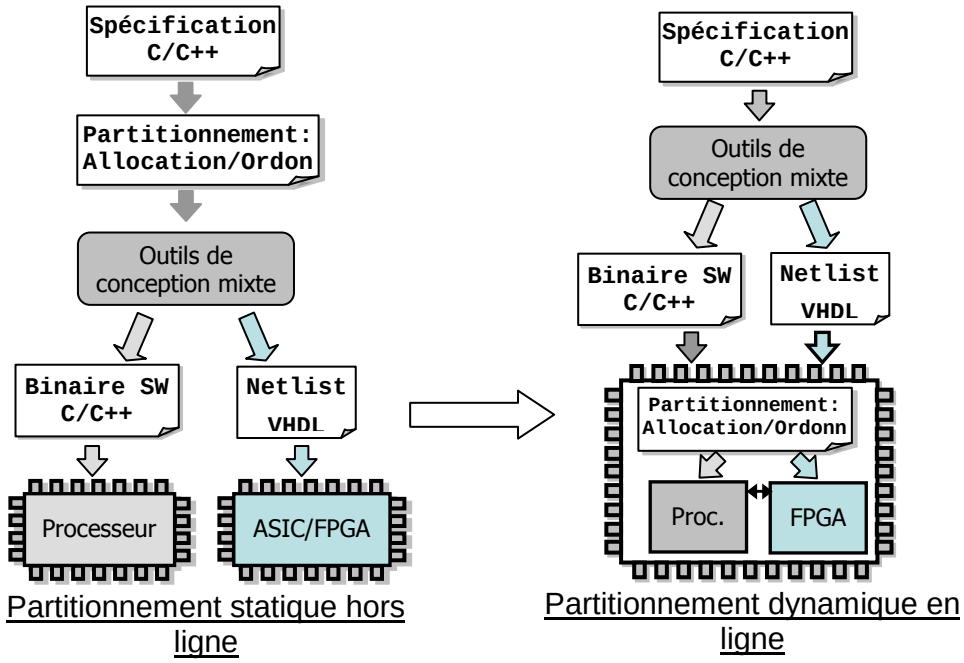


Figure 3.10 : Flot de partitionnement classique et approche de partitionnement en ligne

La partie à intégrer sur la même puce est détaillée dans le schéma de la figure 3.11. Ce schéma sera utilisé tout au long des trois prochains chapitres afin de détailler chaque bloc constituant la méthodologie de partitionnement en ligne.

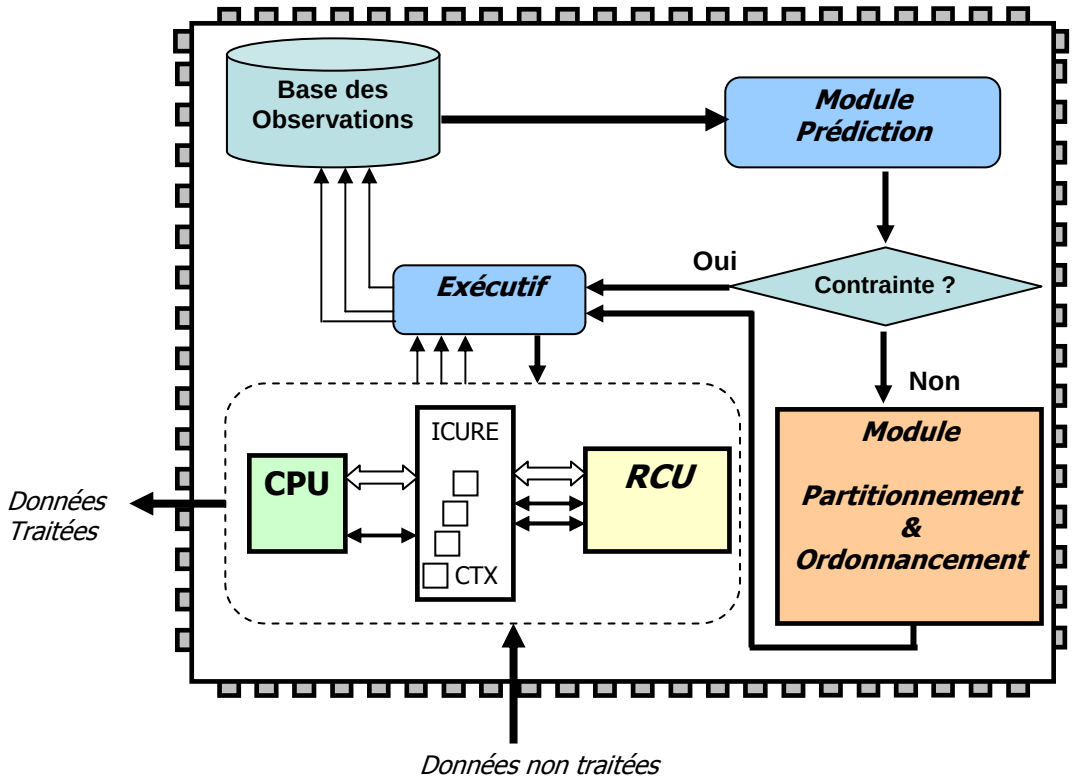


Figure 3.11 : Méthodologie de partitionnement dynamique en ligne

Notre méthodologie se compose donc de trois modules principaux: prédiction, partitionnement et ordonnancement. Ces modules s'ajoutent à l'architecture générique présentée dans la figure 3.2 de ce chapitre. Des solutions d'implémentations de ces modules sont proposées dans les trois prochains chapitres. Mais tout d'abord décrivons succinctement la méthode avec toutes ses étapes.

Le système commence l'exécution des premières données (non traitées) avec une solution du partitionnement trouvée par exemple hors ligne. A l'issue de chaque exécution, l'exécutif récolte les mesures des temps d'exécution des tâches ainsi que les valeurs des paramètres de corrélation.

Le module de prédiction fournit à chaque itération des estimations des temps d'exécution des tâches pour la prochaine itération basées sur les mesures des temps d'exécution précédents. Si le temps d'exécution prédit est tel que la contrainte temporelle est respectée, l'exécutif garde le schéma d'exécution de l'itération en cours. Sinon le module de Partitionnement/ Ordonnancement fournit un nouveau schéma d'exécution qui a pour objectif de respecter la contrainte temporelle durant la ou les prochaines

itérations. Ce nouveau schéma d'exécution est constitué par une nouvelle allocation des tâches aux contextes sur le reconfigurable et au processeur. Il décrit également l'ordonnancement à effectuer pour exécuter l'ensemble des tâches conformément à la contrainte temporelle. L'ensemble des informations d'allocation et d'ordonnancement sont rassemblées dans une table qui est communiquée à l'exécutif ayant pour charge de contrôler l'architecture de traitement pour mettre en œuvre le schéma d'exécution ainsi défini.

Si les valeurs des paramètres de corrélation varient lentement d'une itération à une autre, le module Partitionnement/Ordonnancement intervient peu pendant l'exécution de l'application.

Conclusion

Il est donc important de disposer dans l'architecture d'implémentations (les contextes) judicieusement choisies pour permettre à l'architecture de s'auto-configurer efficacement de manière stable pour des longues séquences des valeurs d'entrée. Cette étude n'a pas été réalisée dans nos travaux.

Nous avons introduit dans ce chapitre le flot global de notre méthodologie de partitionnement logiciel/matériel en ligne. Chaque module de ce flot est détaillé dans les chapitres suivants.
Le chapitre 4 détaille le module de prédiction et le principe d'estimation des performances.

CHAPITRE 4

Estimation des performances

Introduction

Ce chapitre étudie le module de prédiction des temps d'exécution.

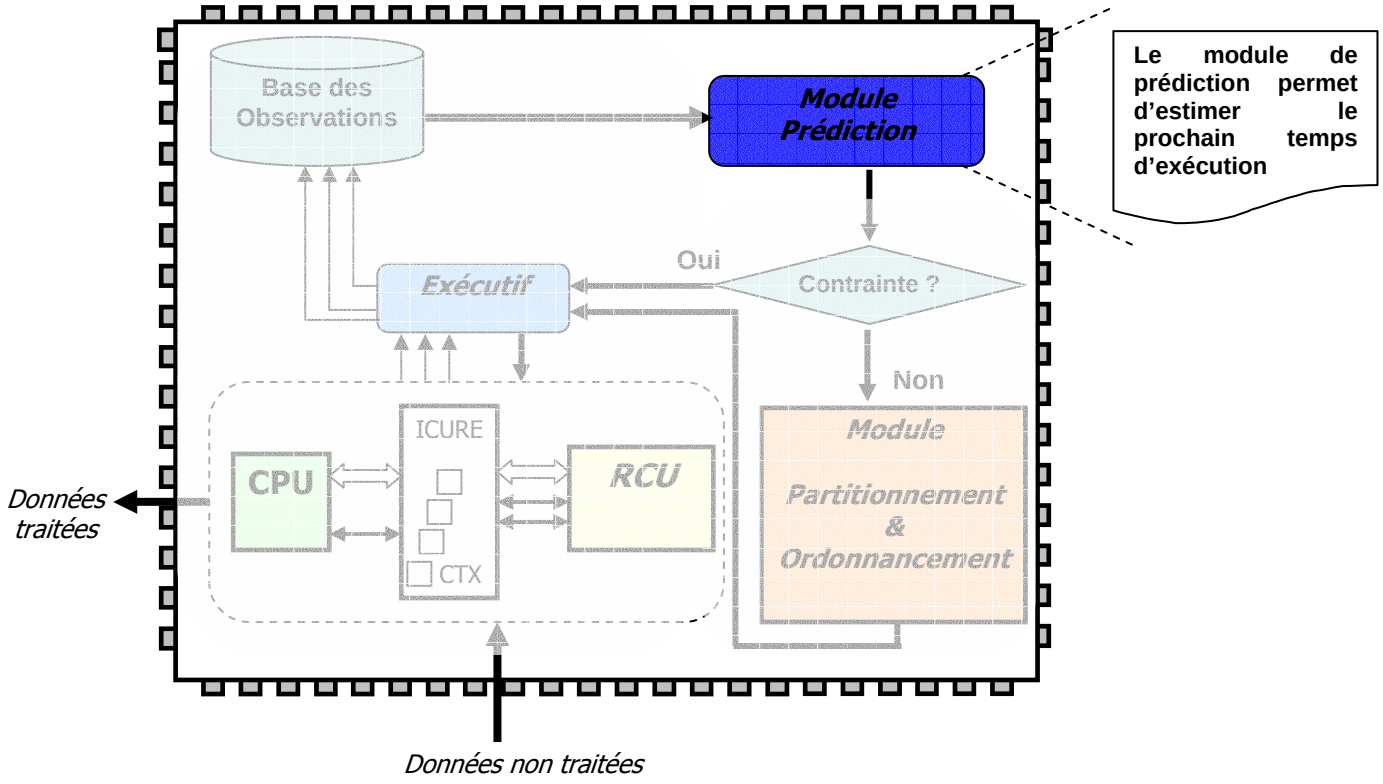


Figure 4.1 : Module de prédiction dans la méthodologie de partitionnement dynamique

Il se compose d'une étape de mesure et d'une étape d'estimation.
L'étape de mesure collecte les temps d'exécution des tâches ainsi que les valeurs de paramètres de corrélation associés. Ensuite l'étape d'estimation a pour but de prédire les temps d'exécution correspondant aux paramètres de corrélation estimés.

Nous présenterons deux méthodes d'estimation des temps d'exécution : la première consiste à effectuer des estimations par équation d'approximation (régression polynomiale) ; la deuxième méthode se base sur l'approche des

K Plus Proche Voisins (KPPV). Cette dernière méthode implique une gestion mémoire des observations.

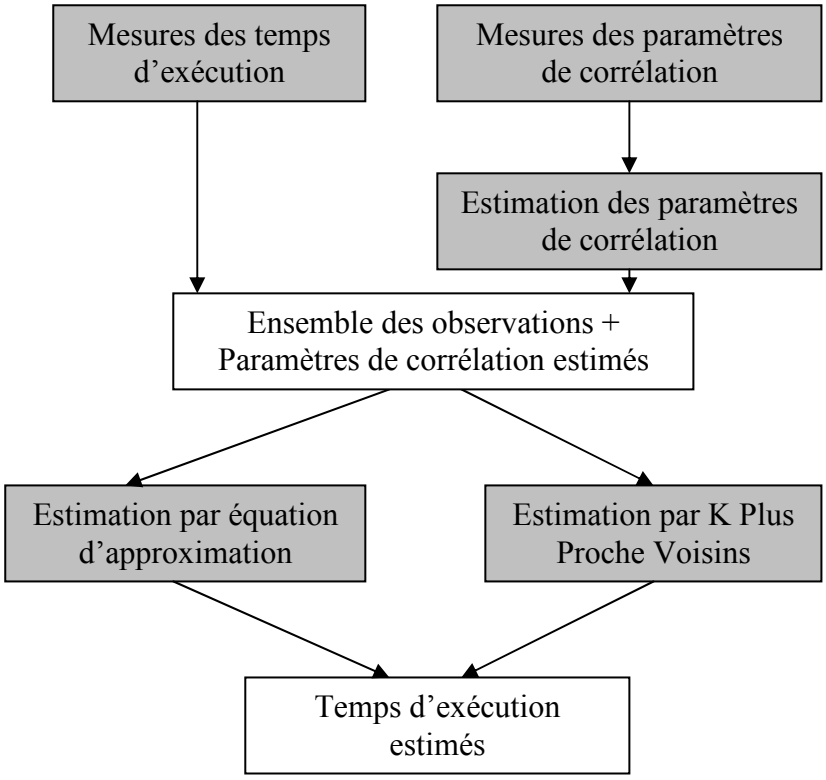


Figure 4.2 : Processus d'estimation des temps d'exécution des tâches

4.1 Mesures des temps d'exécution

Une mesure des temps d'exécution se base sur l'exécution réelle de l'application (ou de la tâche) sur son support d'exécution. Pour un programme donné, la machine exécutera une séquence d'instructions qui peut évoluer d'une exécution à l'autre et dont il s'agit de mesurer le temps d'exécution. L'exécution d'une instruction est contrôlée par l'horloge du processeur qui est régulée par un oscillateur de précision [137].

Vu que l'architecture cible dans notre approche est hétérogène : composée par une unité logicielle (le processeur) et une unité matérielle (le FPGA), nous distinguons deux types de méthodes de mesures du temps d'exécution : l'une matérielle (pour le support d'exécution matériel) et l'autre logicielle (pour le processeur).

Chaque support d'exécution qu'il soit matériel ou logiciel, a ses propres méthodes d'évaluation du temps d'exécution, mais cela n'empêche pas qu'il y ait des méthodes utilisées pour le logiciel qui ont leurs analogues en matériel.

En outre, nous devons distinguer aussi les méthodes de mesures hors ligne, c'est-à-dire sur des séquences de test prédéfinies, de celles effectuées en ligne (sur les données réelles). Les premières sont utilisées dans deux contextes :

- ✓ Pour l'estimateur basé sur la méthode de KPPV, ces mesures servent à initialiser la base des observations.

- ✓ Pour le deuxième estimateur (basé sur les équations d'approximation) ces mesures hors ligne servent à définir les équations d'approximation.

Les deuxièmes méthodes interviennent dans le régime permanent.

Une application est un ensemble de tâches qui s'exécutent suivant un ordonnancement calculé en ligne ou hors ligne. L'objectif consiste à quantifier le temps d'exécution individuel de chaque tâche. Ceci peut présenter quelques difficultés dès lors que les tâches subissent des préemptions.

4.1.1 Critères des méthodes d'évaluation

Ils existent différentes méthodes de mesure du temps d'exécution. Chaque méthode de mesure présente un compromis entre plusieurs attributs comme la précision, la résolution et la difficulté de manipulation.

- ➢ La résolution est limitée par le matériel d'évaluation. Elle correspond à la durée minimale mesurable par le matériel.

- ➢ La précision est un intervalle de fluctuation de la mesure. En effet, pour une même application exécutée plusieurs fois, la mesure de sa durée d'exécution peut fluctuer d'une mesure à l'autre.

- ➢ La difficulté de manipulation est définie comme l'effort nécessaire pour obtenir les mesures. Une méthode qui nécessite une simple exécution du code et qui donne des réponses instantanées est considérée « simple ». Une méthode qui exige une instrumentation évoluée est considérée « difficile ».

4.1.2 Exemples de méthodes de mesure du temps d'exécution

Dans le domaine de l'évaluation des performances, la notion de mesure du temps d'exécution est nécessaire, et par la suite nous trouvons une variété de méthodes permettant de connaître la durée d'exécution d'un programme ou d'une séquence d'instructions. Dans la suite nous présentons un survol de quelques méthodes de mesures de temps d'exécution.

La méthode « ***Stopwatch*** » [138] consiste à chronométrer le temps d'exécution. Elle utilise une référence de temps externe. Ainsi elle est seulement appropriée aux programmes s'exécutant sans interruption et dont le temps d'exécution est de l'ordre de plusieurs minutes pour minimiser les approximations des mesures.

Sous Unix figurent plusieurs commandes de mesures du temps d'exécution : Date, Time, Prof/Gprof [138,139].

La commande ***Date*** utilise l'horloge interne de l'ordinateur au lieu d'une horloge externe. Elle est plus précise que le « ***Stopwatch*** », elle ne tient pas compte des préemptions ou interruptions.

La commande ***Time*** est similaire à la fonction Date sauf qu'elle prend en compte les préemptions, les interruption. On obtient une mesure plus précise du temps d'exécution dans ce cas.

Les commandes ***Prof/Gprof*** sont plutôt des outils de « Profiling » qui permettent de sélectionner les parties du programme qui contribuent le plus au temps d'exécution global.

La méthode utilisant la fonction « ***clock*** » se base sur l'ajout de quelques lignes de code à insérer dans le programme pour mesurer le temps d'exécution. Elle est simple à utiliser. Cependant, elle inclut le temps de mesure dans la durée d'exécution qui peut dans certains cas ne pas être négligeable.

Le temps d'exécution d'une tâche peut être très faible de façon qu'on ne puisse pas le mesurer, il est alors indispensable de répéter l'exécution plusieurs fois et diviser le temps total par le nombre d'itérations.

La majorité des systèmes embarqués ont des circuits ***timer***/compteur qui sont programmables par l'utilisateur. Ces circuits peuvent être utilisés pour les mesures du temps d'exécution. La valeur du compteur est prélevée par appel de fonction aux endroits significatifs du code des tâches ou de l'application. La différence entre deux valeurs mesurées successivement représente la durée d'exécution du code exécuté entre les deux appels.

Cependant, si au cours de la mesure du temps d'exécution du programme il y a des préemptions ou des interruptions celles-ci sont comptabilisées par le timer

L'analyseur logiciel est un terme général pour les outils qui permettent en particulier de mesurer des temps d'exécution. Plus généralement, un analyseur logiciel est un outil qui donne, non seulement, des informations sur les processus ou les fonctions, mais aussi, contient des moyens de mesures du temps d'exécution de petits segments comme les boucles, les blocs de code, et même les instructions singulières.

Les outils de Profiling développés par « QNX software systems » ou l'outil « Scope Profiler » [140], permettent de localiser les parties du code qui à l'exécution prennent le plus de temps.

Les outils d'analyse de performances comme « VENIX-EDS » et ZAP Cross Debugger » [140], vont plus loin et fournissent des informations temporelles sur le système comme les temps de changement de contexte entre tâches.

L'environnement CoDeNIOS [141], donne à l'utilisateur la possibilité d'effectuer des mesures du temps d'exécution de fonctions matérielles et logicielles. En effet, des compteurs sont placés automatiquement dans le système lors de la conception. Un compteur est attaché à chaque module matériel ou logiciel pour évaluer le nombre total de cycles d'horloge puis par l'exécution du module.

Une méthode de mesure de temps d'exécution non invasive est l'utilisation d'un ***analyseur logique***. Cet outil de mesure précis nécessite de disposer de signaux physiques ad hoc pour déclencher la mesure. Cette approche n'est pas utilisable pour un système embarqué directement intégré dans son environnement.

En conclusion, nous pouvons dire que la précision donnée par chaque méthode de mesure diffère de l'une à l'autre. Certaines sont aisées à réaliser et moins précises, cependant d'autres sont plus précises mais plus difficiles à manipuler et coûteuses.

Dans le cas de la plateforme utilisée (1 processeur connecté à une unité reconfigurable dynamiquement) nous considérons que nous disposons d'un timer permettant de mesurer les temps d'exécution des tâches allouées au processeur

En ce qui concerne les réalisations matérielles des tâches sur le reconfigurable nous considérons qu'au moment de la conception, un compteur et les signaux de contrôle correspondants sont automatiquement

intégrés de manière à initialiser et activer le compteur au moment de l'exécution de la tâche puis d'arrêter le compteur et lire son contenu lorsque la tâche termine son exécution. Ces valeurs sont ensuite utilisées pour mettre à jour la base de données.

4.2 Prédiction du paramètre de corrélation des tâches

Le paramètre de corrélation d'une tâche peut être une caractéristique des données traitées par la tâche. S'il s'agit du traitement d'image par exemple, le temps d'exécution de la fonction extraction de contour est à priori fortement corrélé au nombre d'objets présents dans une image.

Généralement, la mesure de la valeur du paramètre de corrélation ne pose pas de problème dans des applications orientées flots de données comme le traitement d'images mais estimer sa prochaine valeur n'est pas toujours aisé.

La variation de la valeur du paramètre de corrélation peut être dans certains cas aléatoire (il est difficile de prédire le nombre d'objets qui seront présents dans la prochaine image). L'objectif serait alors d'estimer la prochaine valeur du paramètre de corrélation en minimisant l'erreur d'estimation tout en évitant de sous-évaluer cette valeur. En effet, l'inverse aurait pour effet de sous-estimer le prochain temps d'exécution avec pour conséquence un risque de dépassement d'échéance.
Nous n'avons pas approfondi ce point car il est dans un premier abord dépendant de l'application. Des travaux comme ceux développés dans [156] utilisent un historique pour estimer le temps d'exécution et les ressources relatives à un "job" dans un contexte de calcul sur grille.
Ce type de techniques basée sur un historique est d'une trop grande complexité par rapport à l'objectif de prédire en temps réel l'évolution de paramètres de corrélation.

De manière très simplifiée, nous considérons l'hypothèse selon laquelle la valeur du paramètre de corrélation ne peut qu'augmenter d'une itération à la suivante. Cette hypothèse est relativement pessimiste mais induit une erreur à priori positive sur l'estimation du temps d'exécution prédit.

Pour éviter de surcontraindre le système il est nécessaire d'ajuster au mieux cette prédiction de variation du temps d'exécution. L'expérimentation présentée dans le dernier chapitre illustre cette approche pragmatique.

4.3 Estimation du temps d'exécution

A partir de la valeur prédite d'un paramètre de corrélation, il est nécessaire d'en déduire un temps d'exécution.
Pour ce faire, nous considérons deux méthodes d'estimation du temps d'exécution des tâches. L'une est basée sur une technique de régression paramétrique et l'autre non-paramètrique. Notons que nous pouvons avoir dans un seul système la combinaison de plusieurs estimateurs.

4.3.1 Etat de l'art des estimateurs de temps d'exécution

On distingue les approches d'estimation hors ligne du temps d'exécution d'une tâche : par analyse du code [142,143], par simulation (ou profiling) [144,145,146,147,148,149,150,151] de celles qui opèrent en ligne: par exemple par prédiction statistique [152,153,154,155].

L'approche hors ligne par analyse du code se base essentiellement sur une répartition du programme en blocs de base. Un bloc de base est une suite d'instructions séquentielle ne contenant qu'un seul point d'entrée et un seul point de sortie.

La structure du programme est représentée comme un graphe de contrôle dont les nœuds sont les blocs de base. Ce graphe décrit tous les enchaînements possibles entre blocs de base.

Pour une architecture donnée, une analyse du plus long chemin dans le graphe est utilisée pour identifier le temps d'exécution correspondant au pire cas.

La technique hors ligne d'estimation par simulation consiste à exécuter un programme sur l'architecture cible avec un ou plusieurs jeux de test. Le nombre de cycles mesurés donne une estimation du temps d'exécution. Cette méthode est sensible à la qualité des tests considérés.

Ces techniques d'estimation hors-ligne ne permettent pas de tenir compte des évolutions des données traitées à chaque activation de la tâche et donc des variations des temps d'exécution que nous essayons d'exploiter pour obtenir une architecture plus compacte que celle issue d'une conception basée sur un ensemble de WCET associé aux tâches.

Les approches en ligne utilisent les observations antérieures pour prédire le temps d'exécution futur.

Par exemple dans le cas des méthodes de prédictions statistiques, au moment de l'exécution d'une tâche, la valeur du temps d'exécution mesuré

peut être ajoutée à l'ensemble des observations précédentes afin d'améliorer les estimations. La qualité des estimations produites par ces méthodes statistiques est donc liée à la gestion de cet ensemble d'observations.

L'avantage de cette méthode réside dans ses aptitudes à être indépendante des caractéristiques des données (taille, type de données), de la structure du code et de l'architecture considérée. Cependant, l'ensemble des observations construit à l'exécution est dépendant de l'architecture considérée. Dans notre cas, une tâche peut s'exécuter sur le processeur et sur un ou plusieurs contextes du reconfigurable.

Parmi les approches capables d'opérer en ligne, nous nous sommes intéressé aux techniques de régression.

Dans la littérature, il existe deux classes (ou deux catégories) de techniques de régression :

- Techniques de régression paramétrique
- Techniques de régression non-paramétrique

Dans la technique de régression paramétrique, on cherche à définir une forme de la fonction qui lie la valeur Y au paramètre X. Cette forme sera décrite par un ensemble fini de paramètres, correspondant par exemple à une équation de régression polynomiale. Cette approche par régression paramétrique est utilisée dans le paragraphe 4.3.2.

Dans la technique de régression non paramétrique (Smoothing technique), les estimations ne dépendent que de l'ensemble des observations passées sans avoir besoin d'approximer la forme de la fonction.

Les principales méthodes de Smoothing sont :

- Estimateur Kernel
- Estimateur Spline
- Estimateur KNN (K Nearest Neighbours) ou KPPV
- Estimateur par série orthogonale

Nous avons étudié la méthode KPPV qui pourrait permettre d'intégrer dans une architecture auto-adaptative un estimateur générique ne nécessitant pas d'effectuer une étude hors-ligne pour définir un modèle (paramétrique) représentant le temps d'exécution en fonction du paramètre de corrélation.

4.3.2 Méthode d'estimation du temps d'exécution basée sur KPPV

La technique d'estimation par K Plus Proches Voisins (KPPV) est une méthode de régression non-paramètrique basée sur une courbe de régression qui décrit un rapport général entre une variable explicative X et une variable de réponse Y.

Ayant observé X, la valeur moyenne de Y est donnée par la fonction de régression. La forme de la fonction de régression peut donner une idée sur la position des maximum des observations (mesures) ou bien sur un type spécial de dépendance entre les deux variables X et Y.

Si n points $\{(x_i , y_i)\}_{i=1..n}$ sont collectés, la relation de régression peut être modélisée par

$$\boldsymbol{y_i = m(x_i) + e_i \qquad i=1..n} \tag{4.1}$$

Avec une fonction de régression m et une erreur d'observation $\boldsymbol{e_i}$

La fonction de régression la plus adéquate à notre contexte est celle employée par M. Iverson dans [155].
La fonction m(x) peut être calculée par la formule générique suivante :

$$m(x) = \frac{1}{n}\sum_{i=1}^{n} W_i(x) t_i \tag{4.2}$$

Où n est le nombre de mesures disponibles, les coefficients W_i sont des poids affectés aux valeurs des mesures t_i. Les méthodes de calcul des poids peuvent varier d'une approche de régression à l'autre.

Comme le montre la figure 4.3, en fixant une valeur au paramètre de corrélation (à A par exemple sur la figure), la valeur de l'estimation m(x) est obtenue par la somme pondérée des voisins. La pondération est fonction de la position de ce paramètre de corrélation. Des poids forts sont affectés aux voisins les plus proches et des poids de plus en plus faibles aux voisins les plus éloignés.

Pour des raisons de simplicité, nous calculons uniquement la variance de l'erreur $\sigma^2(e)$ par la formule suivante:

$$\sigma^2(e) = \frac{1}{n}\sum_{i=1}^{n} W_i(x)(t_i - m(x))^2 \tag{4.3}$$

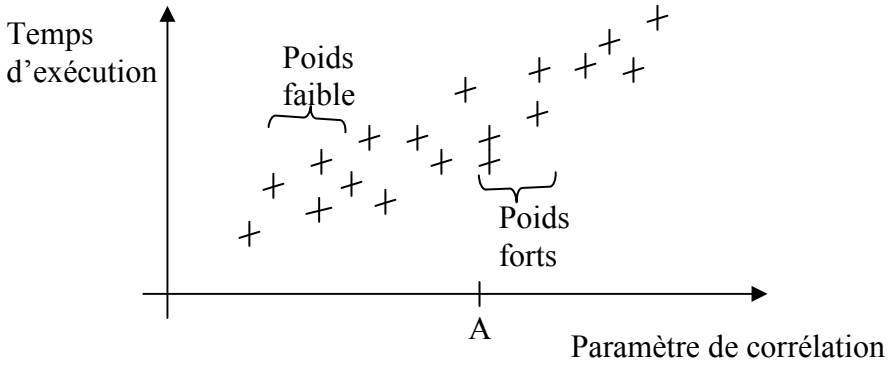


Figure 4.3 : Méthode des K plus proches voisins (KPPV)

L'expression des poids dans notre approche d'estimation utilise la fonction ***Epanechnikov*** [155].
L'expression qui donne la séquence des poids sera alors :

$$K(u) = 3/4(1-u^2), |u| = 1 \tag{4.4}$$

$$W_i(x) = \frac{K_r(x - x_i)}{f_r(x)} \tag{4.5}$$

Avec :

$$f_r(x) = 1/k \sum_{i=1}^{k} K_r(X - X_i) \tag{4.6}$$

Et :

$$K_r(u) = \frac{1}{R} K(\frac{u}{R}) \tag{4.7}$$

La valeur R est la distance maximale qui sépare le paramètre de corrélation estimé et le voisin le plus éloigné parmi les k plus proches voisins.

Les résultats d'expérimentation de cet estimateur seront donnés en détails dans le dernier chapitre (chapitre expérimentations et résultats) de ce rapport.

La Gestion des observations (mesures) est nécessaire dans la méthode d'estimation du temps d'exécution par KPPV. En effet, à chaque exécution d'une tâche une mesure est effectuée dont il s'agit de décider si cette valeur doit être intégrée dans les observations.

Ceci pose plusieurs questions relatives à la taille et au contenu de cette base des observations et également sur son contenu initial. Le contenu initial de la base des observations influe beaucoup sur la rapidité de convergence de l'estimateur dans le régime permanent. Dans notre cas, ce dernier est

préparé hors ligne avec des mesures réelles effectuées sur des séquences types de données.
Nous n'avons pas fait d'études approfondies sur la meilleure gestion de la base des observations (taille, mise à jour, complexité induite). Plusieurs autres expérimentations ont été cependant menées qui sont présentées dans le chapitre résultats.

Un autre problème relatif à l'estimation proprement dite concerne le choix de **la valeur de k** (le nombre de voisins à considérer). En effet, si le nombre d'observations utilisées augmente trop, la qualité de l'estimation m(x) se détériore. En outre un nombre d'observations réduit influe sur la variance de m(x) et par la suite sur l'erreur e_i. Dans notre cas la valeur de k est déterminée expérimentalement.
Il est clair que tous ces paramètres ont un impact important sur la qualité des estimations. La méthode de KPPV nécessite une attention particulière et peut conduire à une complexité de réalisation élevée compte tenu de la précision et de la généricité obtenue.

Pour cela nous avons également considérée une méthode d'estimation basée sur une régression paramétrique plus simple à mettre en œuvre mais impliquant une paramétrisation hors ligne.

4.3.3 Méthode d'estimation du temps d'exécution basée sur une fonction polynomiale

Dans cette méthode d'estimation basée sur une technique de régression paramétrique, nous cherchons à définir la forme de la fonction qui lie la valeur du temps d'exécution au paramètre de corrélation.

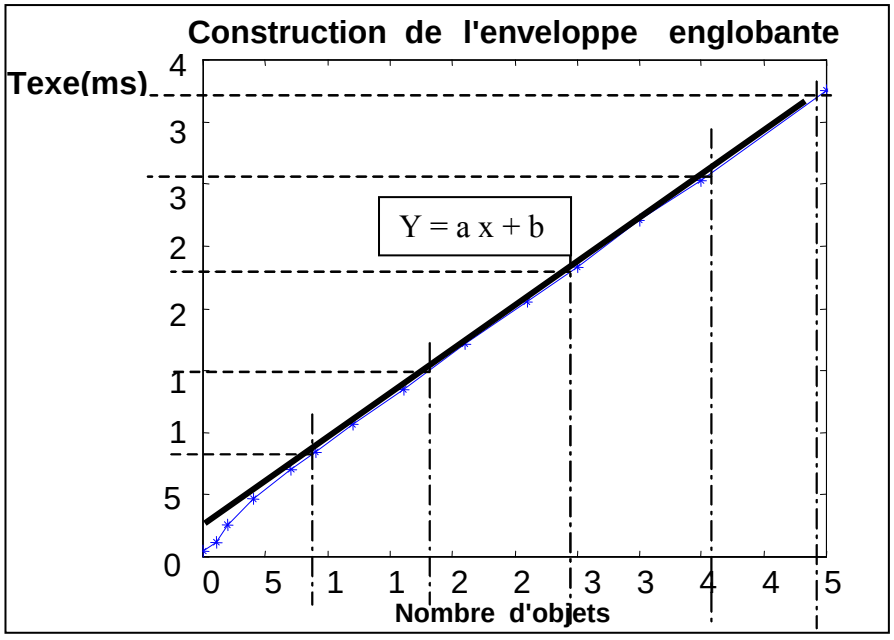


Figure 4.4 : Exemple d'une estimation par équation d'approximation

Cette forme est décrite par un ensemble fini de paramètres correspondant par exemple aux coefficients d'un polynôme. Dans le cas de la figure 4.4, la courbe qui relie le temps d'exécution de la fonction "Construction de l'enveloppe englobante" dans une application de détection de mouvement au paramètre de corrélation (le nombre d'objets) peut être représentée par le polynôme de degré 1 suivant:

$$\mathbf{Texe = a * Nombre\_objets + b} \quad (4.8)$$

Le degré du polynôme et les coefficients sont calculés après un travail effectué hors ligne. Lorsque la corrélation entre le temps d'exécution et le paramètre de corrélation est faible (c'est à dire qu'il existe d'autres paramètres qui influent sur le temps d'exécution), l'identification des paramètres du modèle conduisant à une erreur faible n'est pas facile à trouver. Pour ce type de tâche, nous choisirons l'utilisation du premier estimateur basé sur l'approche KPPV. Ce choix est validé par les expérimentations présentées dans le chapitre résultats.

4.3.4 Avantages et inconvénients de ces méthodes d'estimation

Comme nous venons de l'indiquer, chaque méthode d'estimation a ses avantages et ses inconvénients. La méthode KPPV est plus complexe à implémenter (avec des divisions dans les formules de calcul des poids), mais elle est plus générique car elle s'adapte à toute forme de corrélation entre le temps d'exécution et le paramètre de corrélation.
Au contraire, l'approche d'estimation par fonction polynomiale ne nécessite pas un effort d'implémentation en ligne car la majeure partie du travail est effectuée hors ligne (calcul des coefficients). Mais l'inconvénient est la qualité des estimations pour certaines tâches qui peut être faible si plusieurs paramètres de corrélation contribuent au calcul du temps d'exécution.
Le tableau suivant récapitule les inconvénients et les avantages de chaque méthode d'estimation.

Méthode	**Avantages**	**Inconvénients**
KPPV	• Générique • Travail hors ligne faible • efficace	• complexe • implémentation difficile dans un système embarqué
Fonction polynomiale	• Simple à implémenter en ligne	• Travail hors ligne nécessaire • Efficace pour des fonctions simples (un seul paramètre de corrélation)

Conclusion

Dans ce chapitre nous avons présenté le principe de la méthode de prédiction du temps d'exécution. Nous avons proposé deux approches d'estimation : la première est basée sur le principe des K plus proche voisins, la deuxième est basée sur des équations d'approximation.
La première est plus générique, donc mieux adaptée à une large classe d'applications mais elle est plus complexe à réaliser que la seconde qui est dédiée à l'application.
Le chapitre suivant détaille le module de partitionnement/ordonnancement en ligne.

CHAPITRE 5

Heuristique de Partitionnement/Ordonnancement logiciel/matériel en ligne

Introduction

Nous étudions dans une première partie de ce chapitre une nouvelle heuristique de partitionnement basée sur le principe de la réallocation dynamique des tâches (migration d'une cible à une autre). L'évaluation d'un résultat de partitionnement logiciel/matériel nécessite un ordonnancement en ligne. L'approche d'ordonnancement mise en oeuvre fait l'objet de la deuxième partie de ce chapitre.

5.1 Partitionnement logiciel/Matériel

Le module de partitionnement est associé au module d'ordonnancement car ce dernier valide les résultats du premier (voir figure 5.1).

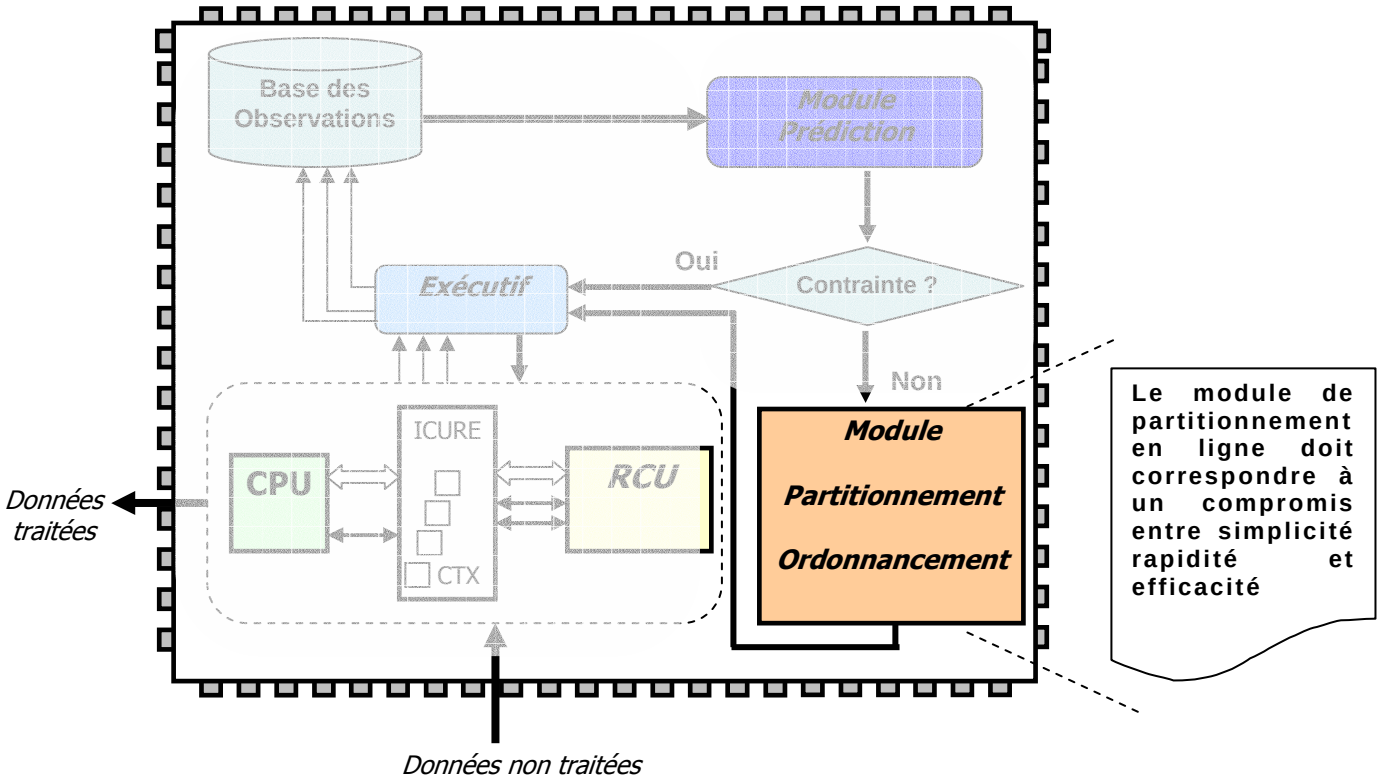


Figure 5.1 : Module de Partitionnement/Ordonnancement dans la méthodologie de partitionnement dynamique

Le module de partitionnement opère pendant l'exécution de l'application. Il est sollicité à chaque fois que l'estimation du temps d'exécution de la configuration en cours ne satisfait plus la contrainte temps réel.

Vu la complexité du problème, il est impossible d'effectuer une recherche en ligne de la solution optimale de partitionnement logiciel/matériel tout en continuant à traiter en temps réel les données de l'application.
Il faudrait soit suspendre les traitements de l'application soit considérer des heuristiques rapides, et donc sous-optimales, de partitionnement.
C'est ce deuxième cas que nous considérons.
L'objectif est de déterminer une heuristique de partitionnement qui soit à la fois rapide, simple et efficace.

Notre idée principale consiste ainsi à considérer la solution de partitionnement actuelle pour laquelle l'estimation prédit un dépassement de la contrainte de temps et à l'améliorer de telle sorte qu'elle respecte la contrainte temps réel pour les futures exécutions de l'application.

5.1.1 Concept de migration des tâches

De par sa définition, un résultat de partitionnement logiciel/matériel consiste à affecter des tâches sur les unités logicielles et d'autres sur les unités de calcul matérielles. Modifier ce résultat revient alors à changer ces choix d'affectation par d'autres choix qui doivent par exemple répondre aux exigences de temps réel du système.

Ainsi le principe retenu est assez simple : une tâche logicielle dont le temps d'exécution devient critique est une candidate privilégiée pour être affectée à une implémentation matérielle, éventuellement à la place d'une autre tâche qui est devenue non critique.
Reste alors à décider quelles tâches doivent migrer du logiciel vers le matériel parmi plusieurs tâches critiques, et éventuellement à la place de quelles tâches matérielles parmi celles qui ne sont plus critiques.

Notre concept de partitionnement logiciel/matériel est fondé sur le principe de migration "directe" de tâches des unités logicielles vers les unités matérielles et de migration "inverse" des tâches des unités matérielles vers les unités logicielles.

5.1.2 Migrations directes: Accélération du traitement

Le but d'une migration directe est d'accélérer le traitement en vu de satisfaire la contrainte temps réel de l'application. Ceci implique que la migration directe est déclenchée lorsque l'estimateur prévoit un dépassement du temps d'exécution total de l'application par rapport à la contrainte.

Il se pose alors le problème de déterminer les tâches à migrer et avec quels objectifs. Le premier objectif est que les contraintes de temps de l'application doivent être vérifiées au mieux. Le deuxième objectif vise à se donner les capacités, autant que faire se peut, de permettre ces migrations.
En effet, les migrations directes ont pour effet d'occuper les ressources matérielles pour les tâches identifiées comme critiques à un instant donné.
Si des migrations inverses ne sont pas effectuées, les ressources matérielles peuvent devenir insuffisantes si une nouvelle tâche venait à devenir critique. Il faut donc organiser également les migrations inverses.

Pour illustrer la méthode proposée, nous prenons un exemple de DFG qui comporte cinq tâches (voir figure 5.2) et une architecture mixte logicielle/ matérielle. Chacune des tâches du DFG peut être implémentée sur le logiciel (le processeur) ou sur le matériel (FPGA) avec trois possibilités pour ce dernier qui correspondent à la courbe d'implémentation donnée dans la figure 3.5 du chapitre 3.

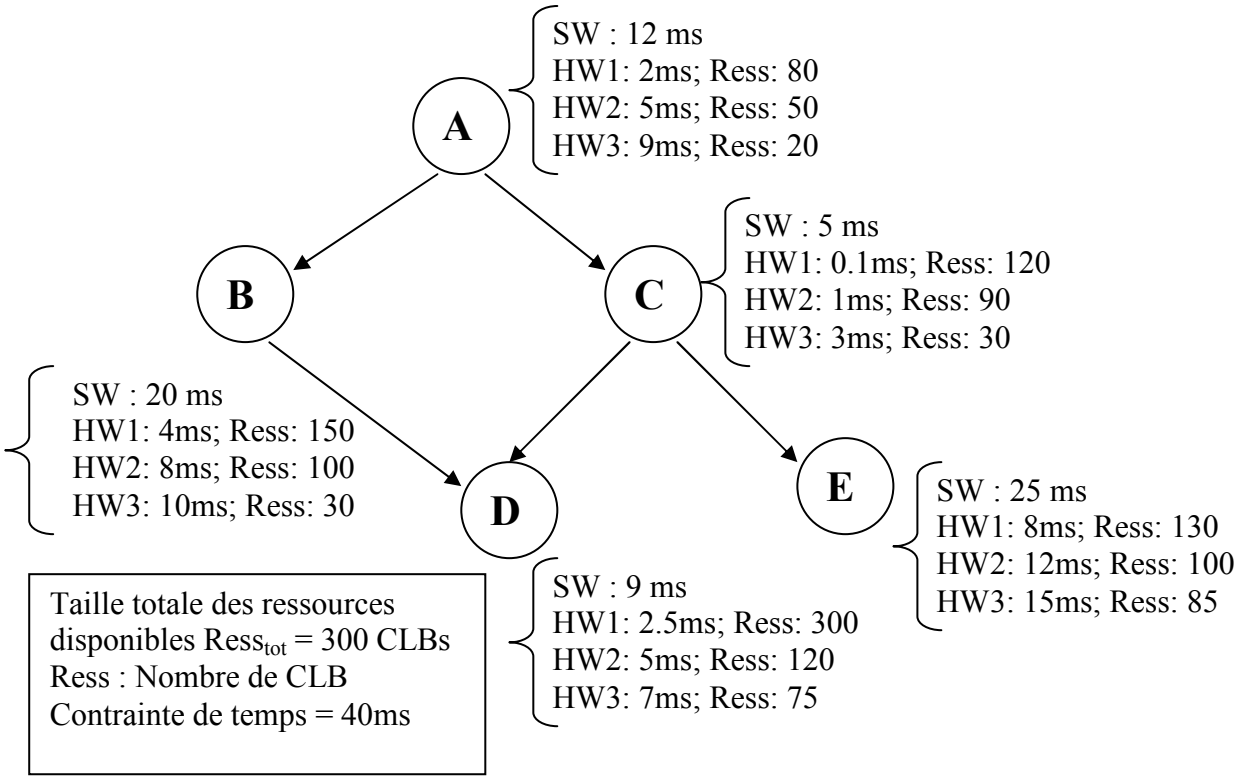


Figure 5.2 : Exemple d'un DFG test

En partant par exemple lors de l'itération 1 d'une implémentation entièrement logicielle de l'application, nous obtenons après ordonnancement de toutes les tâches sur le processeur le résultat donné par le tableau et le schéma d'exécution suivants (figure 5.3):

Itération 1 :

Tâche	A	B	C	D	E
$Texe_i$	12	20	5	9	25
Ress	0	0	0	0	0

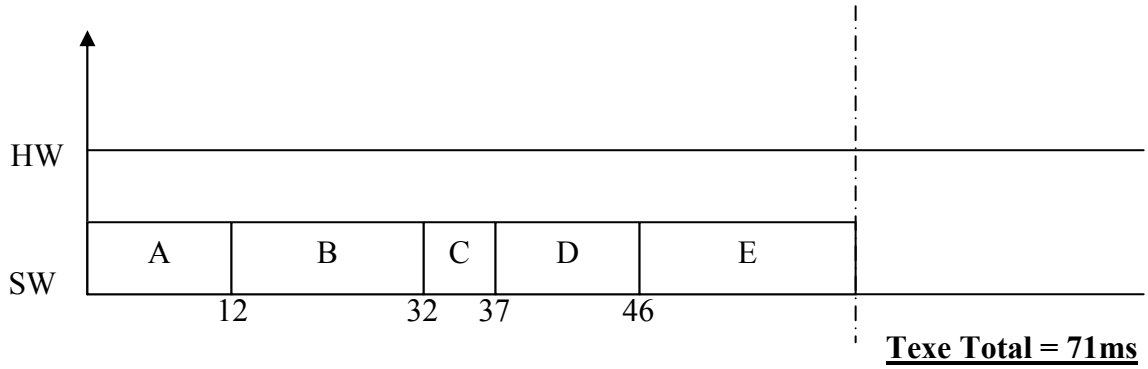


Figure 5.3 : Exemple d'ordonnancement de l'application

La solution purement logicielle possède un temps d'exécution très supérieur à la contrainte temps réel de 40 ms.
Pour l'itération suivante il faut absolument faire migrer des tâches considérées comme critiques en temps d'exécution vers l'unité matérielle. Afin d'effectuer les choix parmi ces tâches, nous dressons dans le tableau qui suit toutes les valeurs possibles des gains potentiels en temps (différences des temps d'exécutions estimés après migration par rapport aux valeurs des temps d'exécution avant migration pour toutes les tâches) et les différences des nombres de ressources utilisées (avant et après une migration).

Gains (temporel et ressources) obtenus par tâche après migration

Tâche	A		B		C		D		E	
Gain	Gt	Gr	Gt	Gr	Gt	Gr	Gt	Gr	Gt	Gr
G1	10	80	16	150	4,9	120	6,5	300	17	130
G2	7	50	12	100	4	90	4	120	13	100
G3	3	20	10	30	2	30	2	75	10	85

Les gains Gt et Gr sont respectivement le gain temporel et le gain en ressources matérielles obtenus après migration et calculés respectivement par les équations 5.1 et 5.2 :

$$\mathbf{Gt = Texe_{apm} - Texe_{avm}} \qquad \mathbf{(5.1)}$$

$Texe_{apm}$ est le temps d'exécution de la tâche après migration
$Texe_{avm}$ est le temps d'exécution de la tâche avant migration

$$\mathbf{Gr = Ress_{apm} - Ress_{avm}} \qquad \mathbf{(5.2)}$$

$Ress_{apm}$ est le nombre de ressources matérielles utilisées par la tâche après migration
$Ress_{avm}$ est le nombre de ressources matérielles utilisées par la tâche avant migration

G1, G2 et G3 correspondent aux trois couples de gain (Gt et Gr) possibles pour une tâche implémentée sur le processeur :

1) Du SW vers HW1
2) Du SW vers HW2
3) Du SW vers HW3

Notre objectif est de satisfaire la contrainte temporelle de l'application en minimisant le nombre de ressources à utiliser.
Identifier l'ensemble des tâches à migrer correspondant à cet objectif est un problème d'allocation et d'ordonnancement sous contraintes d'optimisation. Ce problème est de type NP-difficile.
L'approche proposée consiste à opérer itérativement en considérant à chaque itération la tâche qui, lorsqu'elle est migrée du logiciel vers le matériel, maximise le gain temporel global du DFG avec un minimum de ressources matérielles utilisées. Pour déterminer cette tâche, nous considérons le critère suivant:

$$\mathbf{C_j = Srest_j * Gt_j} \qquad \mathbf{(5.3)}$$

Avec $$\mathbf{Srestj = Ress_{tot} - \sum_{j=1}^{J=5} Ress_{apm}(j)} \qquad \mathbf{(5.4)}$$

$Ress_{tot}$ est le nombre total de ressources matérielles disponibles dans le reconfigurable.

Gt_j est le gain temporel local induit par la migration de la tâche j.

Le gain temporel calculé pour chaque tâche par la formule 5.1 est un gain local. C'est-à-dire qu'il correspond à la différence des temps d'exécution de la tâche sur les différentes cibles (dans notre exemple une réalisation logicielle et trois réalisations matérielles).

Le gain temporel global de l'application est la différence entre le temps d'exécution total de l'application après migration et celui avant migration.

Le gain temporel local d'une tâche n'est pas forcement égal au gain temporel global de l'application sauf pour les applications entièrement séquentielles.

Pour les applications qui présentent un parallélisme de traitements, un gain temporel local d'une tâche peut être recouvert par le temps d'exécution d'une autre tâche parallèle. Dans ce dernier cas, trois scénarios sont possibles. Les figures 5.4,5.5 et 5.6 illustrent ces différents scénarios en prenant deux tâches T1 et T2 qui s'exécutent en parallèle.

Scénario A :

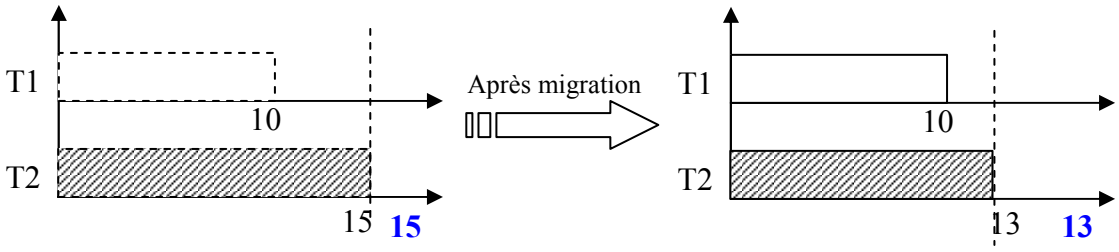


Figure 5.4 : Différence entre gain temporel global et local (scénario A)

Dans ce cas le gain temporel local de la tâche T2 est égale à 2 (15 – 13) et est égal au gain temporel global de deux tâches.

Scénario B :

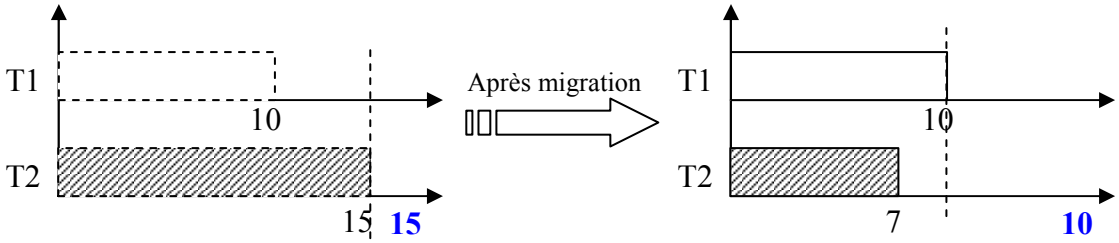


Figure 5.5 : Différence entre gain temporel global et local (scénario B)

Dans ce scénario, le gain temporel local de la tâche T2 après sa migration est égal à 8. Par contre le gain temporel global de deux tâches n'est que 5 (15 – 10)

Scénario C :

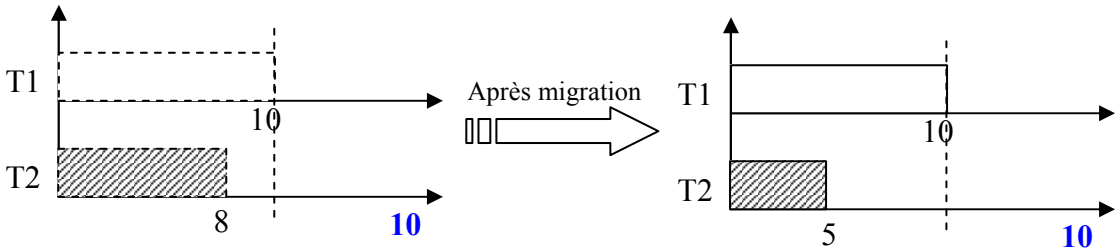


Figure 5.6 : Différence entre gain temporel global et local (scénario C)

Dans ce dernier cas, le gain temporel local de la tâche T2 est de 3 mais le gain temporel global est nul (le temps global d'exécution des deux tâches reste invariant).
Ce problème de différence entre gain temporel local d'une tâche et gain global de l'application se complexifie lorsqu'il s'agit de migrer plusieurs tâches parallèles et/ou séquentielles en même temps.
Connaître l'évolution du gain temporel global induit par la migration d'une ou plusieurs tâches nécessite de calculer un ordonnancement. On ne peut se permettre de tester itérativement la migration de chaque tâche, et ce pour chacune de ses implémentations, jusqu'à trouver une solution satisfaisant la contrainte de temps et qui soit admissible vis-à-vis des ressources libres du reconfigurable. Pour simplifier, on fait l'hypothèse que les tâches avec les gains locaux les plus grands sont celles qui apportent en général les gains globaux les plus importants. Par conséquent, on considère itérativement les tâches suivant leur gain temporel local par rapport à l'accroissement de ressources nécessaires pour réaliser la migration.

Par exemple, le tableau ci-dessous présente toutes les tâches du DFG, triées par ***ordre décroissant*** des valeurs de C_j données par la formule 5.3 pour les trois migrations possibles de chaque tâche vers le matériel.

Tâche j	**Gt_j**	**Gr_j**	**C_j**
E	**17**	130	2890
B	**10**	30	2700
E	13	100	2600
B	12	100	2400
B	16	150	2400
A	**10**	80	2200
E	10	85	2150
A	7	50	1750
C	4.9	120	882
C	4	90	840
A	3	20	840
D	4	120	720
C	**2**	30	540
D	2	75	450
D	6.5	300	0

Selon notre critère, et pour satisfaire la contrainte temps réel à l'itération i+1, il faut migrer dans l'ordre les tâches suivantes (on considère dans ce cas la migration du logiciel vers le matériel) : La tâche E puis B ensuite A et enfin C.

Pour chaque tâche candidate à la migration, l'évaluation du partitionnement obtenu est donnée par un algorithme d'ordonnancement. Notre technique d'ordonnancement mise en œuvre est détaillée dans la deuxième partie de ce chapitre.

Le résultat d'ordonnancement est comparé à la contrainte temps réel après chaque décision de migration. Lorsque l'estimation du temps d'exécution total de l'itération i+1 respecte la contrainte temps réel, le module exécutif garde le schéma d'allocation du dernier partitionnement pour les futures exécutions de l'application.
Le schéma suivant montre les différentes interactions entre les modules Partitionnement, Ordonnancement et Exécutif pour déterminer un partitionnement qui répond au mieux aux contraintes temps réel.

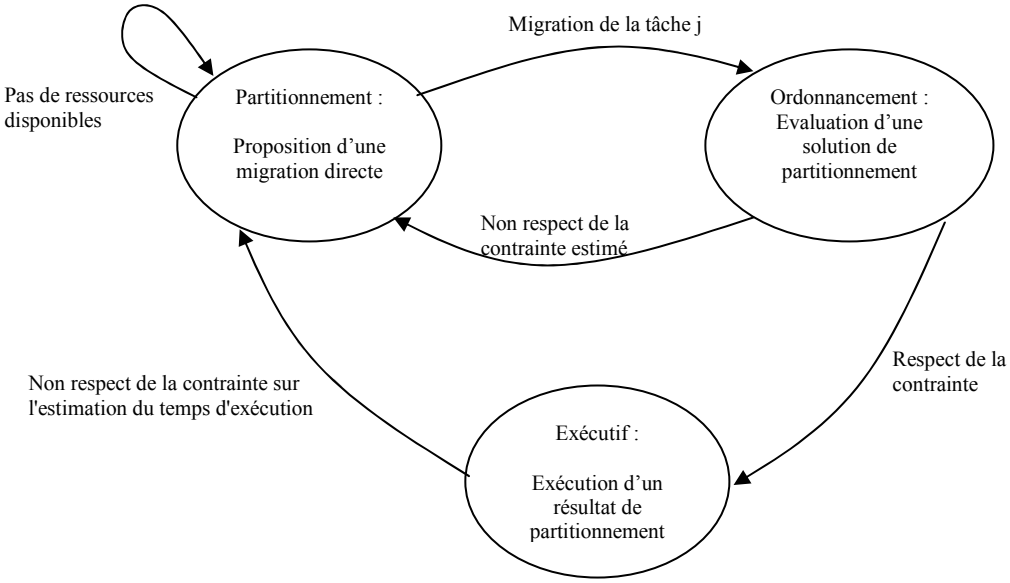


Figure 5.7 : Recherche en ligne d'une solution de partitionnement

5.1.3 Migrations inverses : libération des ressources

En effectuant plusieurs migrations des tâches du logiciel vers le matériel, le reconfigurable peut saturer, particulièrement si toutes les tâches ne peuvent être allouées au matériel. Ainsi le système ne peut plus effectuer de migrations directes alors que les contraintes de temps sont dépassées. Pour éviter de mettre le système dans une telle situation, nous avons proposé une migration des tâches qui vise à libérer des ressources matérielles lorsque le temps d'exécution de l'application est faible par rapport à la contrainte.

Deux approches ont été envisagées : la première effectue un remplacement de tâches lors d'une migration directe. C'est-à-dire on associe une migration

inverse à une migration directe. Mais cette solution est complexe à réaliser, voire même impossible vu que la migration directe est requise lorsque la contrainte de temps est dépassée, c'est-à-dire qu'on dispose de peu de temps pour trouver une solution satisfaisante. C'est pour cela que nous avons opté pour une deuxième approche qui consiste à effectuer les deux types de migration en temps différé. Ceci veut dire que les migrations inverses précédent les migrations directes pour une préparation du reconfigurable à recevoir des tâches devenues critiques sur le processeur.

Pour réaliser la migration inverse, nous avons défini deux seuils du temps d'exécution : un seuil bas (LL) et un seuil haut (HL). Le seuil HL correspond à la contrainte temps réel : il faut que le temps d'exécution de l'application soit toujours inférieur à ce seuil ou tout du moins le plus souvent possible inférieur à ce seuil.

Lorsque l'ordonnancement retourne un temps d'exécution total, estimé pour l'itération suivante, supérieur à HL, l'algorithme de partitionnement effectue des migrations directes des tâches critiques (du processeur vers le reconfigurable). Ceci dans la limite du nombre de ressources matérielles libres du reconfigurable.

Dans le cas où l'ordonnancement retourne un temps d'exécution total estimé inférieur au seuil bas (LL), l'algorithme de partitionnement migre les tâches qui occupent le plus de ressources matérielles vers le processeur dans le but de libérer une partie du reconfigurable et augmenter ainsi le budget de ressources libres pour effectuer des migrations directes lorsque ceci est nécessaire.

Par conséquent, on distingue trois cas en fonction de la valeur du temps estimé par l'ordonnanceur:

- Temps d'exécution estimé supérieur à HL :

  S'il reste de ressources matérielles disponibles alors le partitionnement effectue des migrations directes. Sinon le système perd une image et passe à l'image suivante en gardant la même configuration.
- Temps d'exécution estimé inférieur à LL :

  Le partitionnement calcule des migrations inverses
- Temps d'exécution estimé se situe entre les deux seuils :

  L'application s'exécute en gardant la même configuration d'architecture.

Définitions des seuils :

Dans l'absolu le seuil HL est égal à la contrainte temporelle qui peut être dans le cas de traitement d'images égal à 40ms de traitement d'une séquence vidéo à 25 images par seconde.
Pour effectuer l'ordonnancement et le partitionnement en ligne, il faut tenir compte de leurs temps d'exécution respectifs. Ainsi, le seuil haut est égal à la

valeur de la contrainte temps réel diminuée de la somme des temps d'exécution des algorithmes d'estimation, d'ordonnancement et de partitionnement.

$$HL = Ctr - (Texe\_sch + Texe\_Part + Texe\_Estim) \quad (5.5)$$

Où Texe_sch, Texe_Part et Texe_Estim sont respectivement les temps d'exécution des algorithmes d'ordonnancement, de partitionnement et d'estimation.
La valeur du seuil bas est plus compliquée à déterminer. En effet, un LL trop bas ne permet pas d'effectuer des migrations inverses et ainsi le système sera saturé dès que les migrations directes remplissent totalement le reconfigurable. Par contre, un seuil LL trop élevé (très proche du seuil HL) met le système dans un état instable à cause des migrations continuelles, alternativement directes puis inverses. Une migration inverse sera suivie nécessairement par une autre migration directe pour corriger le temps d'exécution total du système. Il est à noter que le seuil LL ne peut dépasser en aucun cas le seuil HL.

On pourrait considérer que la valeur du seuil LL est déterminée dynamiquement de manière à maximiser le nombre de ressources libres (LL plutôt haut) sur le reconfigurable et en diminuant le nombre de migrations (LL plutôt bas).
Cette évaluation dynamique permettrait de rendre le système auto-adaptable en fonction de l'environnement dans lequel il est placé. Dans l'étude menée dans la thèse nous n'avons pas considéré cette approche intéressante. De façon pragmatique, la valeur de LL a été déterminée par des simulations sur des séquences de test afin d'obtenir le résultat ci-dessus.

Les migrations inverses ne concernent pas uniquement celles du reconfigurable vers le processeur. Elles peuvent être réalisées d'une implémentation sur le reconfigurable vers une autre implémentation sur le reconfigurable qui utilise moins des ressources.
Pour le cas de l'exemple donné par la figure 5.2, nous distinguons six possibilités de migrations inverses :

1) Du HW1 vers HW2
2) Du HW1 vers HW3
3) Du HW1 vers SW
4) Du HW2 vers HW3
5) Du HW2 vers SW
6) Du HW3 vers SW

Notre approche pour la migration inverse consiste à trouver la migration (ou plusieurs migrations) d'une tâche j du matériel vers le logiciel qui libère le maximum de ressources du reconfigurable avec un minimum d'impact sur le temps d'exécution. Pour réaliser une migration inverse, nous trions toutes les possibilités de migration selon le critère donné par l'équation 5.6 :

$$\mathbf{Cinv_j = Sinv_j * Pt_j} \qquad \mathbf{(5.6)}$$

Pt_j est la perte temporelle locale induite par la migration de la tâche j.
$Sinv_j$ est le gain en ressources: $\mathbf{Sinv_j = Ress_{tot} - Ress\ (j)}$ **(5.7)**
Ress (j) est le nombre de ressources matérielles utilisées par la tâche j (ou bien la différence des nombres de ressources lorsqu'il s'agit d'une migration du reconfigurable vers le reconfigurable).

Nous calculons toutes les valeurs de ce critère (5.6) pour toutes les tâches matérielles et nous sélectionnons la tâche matérielle qui minimise ce critère.

Nous revenons à notre exemple de la figure 5.2 et nous supposons que toutes les tâches choisies précédemment pour des migrations directes ont été réallouées effectivement sur le reconfigurable. L'ordonnancement nous retourne un temps d'exécution total inférieur au seuil LL. Le module de partitionnement est alors déclenché pour une ou plusieurs migrations de tâches du reconfigurable de telle sorte que le temps d'exécution total estimé soit entre les deux seuils HL et LL.

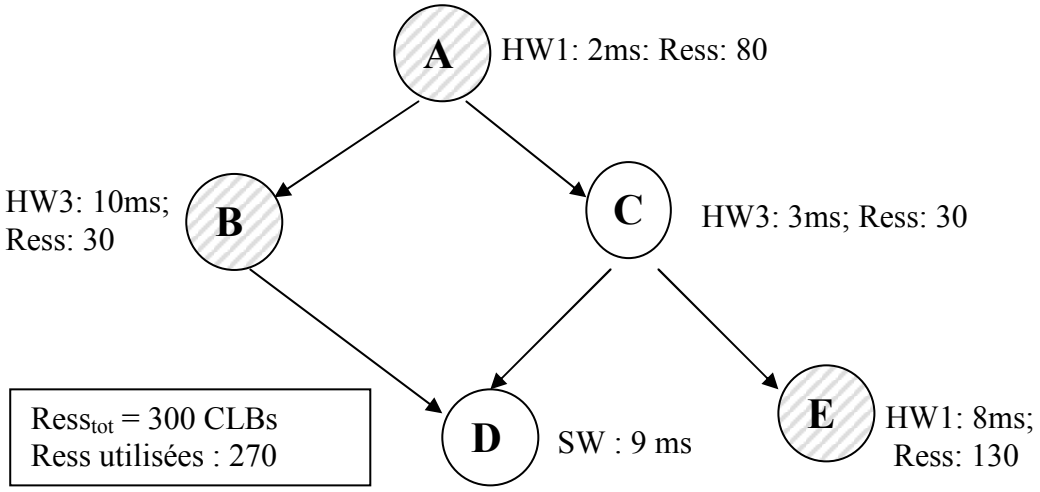


Figure 5.8 : Implémentations des tâches après partitionnement (migrations directes)

Itération 2 :

Tâche	A	B	C	D	E
$Texe_i$	2	10	3	9	8
Ress	80	30	30	0	130

L'ordonnancement de l'application sur l'architecture avec le partitionnement de la figure 5.8 est donné par la figure 5.9.

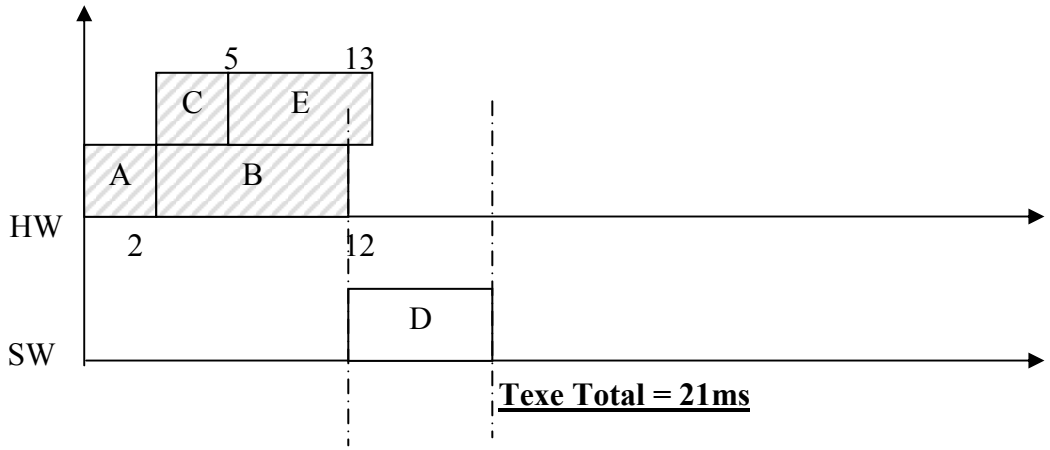


Figure 5.9 : Ordonnancement de l'application après partitionnement (migrations directes)

Pour effectuer une étape de migration inverse, considérons les valeurs données dans le tableau suivant pour l'ensemble des tâches. Notre méthode de partitionnement commence par migrer la tâche C du HW3 vers le SW. Cette migration de la tâche C est évaluée par l'ordonnanceur. Si la valeur du seuil LL est dépassée, et celle de HL n'est pas dépassée, la migration est acceptée et cette solution est retenue pour l'itération suivante. Si la valeur du seuil LL n'est pas dépassée, le processus de migration peut continuer tant qu'il reste du temps disponible à l'itération présente pour effectuer ces migrations, c'est-à-dire la valeur du seuil HL. Dans notre cas, la migration de la tâche suivante dans le tableau est celle de la tâche A du HW1 vers le HW2. Et ainsi de suite jusqu'à ce que le temps d'exécution total se situe entre les deux seuils HL et LL.

Tâche j	**Pt_j**	**$Sinv_j$**	**$Cinv_j$**
C	**2**	270	540
A	**3**	270	810
E	4	270	1080
A	7	240	1680
A	10	220	2200
B	**10**	270	2700
E	17	170	2890

5.1.4 Coût d'une migration

Pour résumer, nous avons trois types de migrations possibles des tâches :
1) Du SW vers HW
2) Du HW vers SW
3) Du HW vers HW

Une migration directe du SW vers HW coûte en surface du reconfigurable, en temps de reconfiguration, en temps de partitionnement et en temps d'ordonnancement. En fait, une migration de ce type se déclenche dans le contexte suivant :

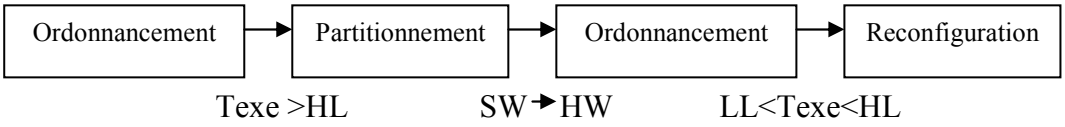


Figure 5.10 : contexte d'une migration directe

Le coût temporel est alors la somme des temps de partitionnement, d'ordonnancement et de reconfiguration.
Sachant que le temps de reconfiguration dépend généralement du nombre de ressources à reconfigurer, une migration du SW vers HW1 coûte en général moins en temps de reconfiguration que la migration de la même tâche du SW vers HW2 et cette dernière coûte moins que la migration du SW vers HW3.
Une migration inverse du HW vers SW coûte un temps de partitionnement et un temps d'ordonnancement. Evidemment ce type de migration engendre généralement une augmentation du temps d'exécution total de l'application. Dans le cas où l'augmentation engendrée dépasse le seuil HL, la migration inverse est annulée et le partitionnement choisit une autre tâche candidate à la migration.
Une migration du HW vers HW peut être directe (du HW1 vers HW3 par exemple) ou inverse (du HW2 vers HW1 par exemple). Ce type de migration coûte un temps de partitionnement, un temps d'ordonnancement et un temps de reconfiguration.
Il est clair que la migration inverse de type HW vers SW est moins coûteuse que la migration inverse HW2 vers HW1 car cette dernière exige un temps de reconfiguration en plus.

Avec les avancées actuelles de la technologie du reconfigurable, le temps de reconfiguration est de plus en plus faible ce qui peut autoriser ce type de reconfiguration dynamique, partielle, à chaud.

5.1.5 Temps de communication après une migration

Il existe trois types de communications : SW-HW, SW-SW ou HW-HW. Généralement les deux derniers temps de communications peuvent être négligeables en comparaison du premier. Ainsi si nous considérons deux tâches qui communiquent et qui sont toutes les deux sur le processeur, alors une migration de l'une de ces tâches vers le HW engendre un temps de communication qui vient s'ajouter au temps d'exécution global de l'application.
De même si nous considérons deux tâches dépendantes dont l'une est sur le matériel et l'autre sur le processeur, alors la migration de l'une des implémentations vers l'autre enlève un temps de communication et peut ainsi diminuer le temps d'exécution global.

Nous n'avons pas tenu compte explicitement des temps de communication dans notre étude. Leur influence porte essentiellement sur le calcul du temps d'exécution par l'ordonnancement et dans la gestion matérielle de la redirection de la communication. La prise en compte du temps de communication peut être aisément réalisée afin d'obtenir un comportement plus précis. L'autre impact important concerne l'utilisation des ressources de communications (par exemple mémoire entre le logiciel et le matériel). Une façon d'en tenir compte serait d'étendre le critère de migration avec un paramètre qui reflète l'utilisation des ressources avant et après migration.

5.2 Ordonnancement logiciel/matériel

Le problème d'ordonnancement des tâches sur plusieurs processeurs (>2) est un problème NP-difficile. La résolution de ce type de problème peut s'effectuer en un temps polynomial à l'aide d'algorithme non déterministe (NP-complet).
La résolution exacte de ce type de problème n'étant pas envisageable pour des problèmes de taille industrielle, une solution approchée obtenue par une heuristique de plus faible complexité peut souvent être satisfaisante pour l'utilisateur.
Nous étudions donc une résolution approchée et rapide du problème d'ordonnancement d'un graphe de tâches.

La problématique d'ordonnancement revient à définir la façon d'agencer les exécutions des tâches sur chaque unité de calcul (logicielle ou matérielle) [133]. Un ordonnancement est faisable, si toutes les contraintes temporelles et de ressources sont respectées. Lorsqu'un tel ordonnancement existe, le système temps réel est dit ordonnançable.

En temps réel souple [133], les critères de validité d'un ordonnancement peuvent faire appel à des données numériques caractérisant le comportement du système. Parmi les métriques les plus courantes, citons:

i) Le taux de respect : c'est la proportion d'exécutions des tâches qui respectent leurs contraintes temporelles.
ii) Le débit : c'est le nombre d'exécutions d'une tâche qui respectent leurs contraintes temporelles par fenêtre de temps donnée. En multimédia par exemple, il peut s'agir de la mesure du nombre d'images rendues par seconde.
iii) Le seuil d'utilisation : c'est le taux d'utilisation du processeur au dessus duquel des tâches commencent à dépasser leurs contraintes temporelles.
iv) Les caractéristiques temporelles : telles que le retard, la gigue de démarrage, la laxité, le temps de réponse par exemple.
Nous proposons dans notre étude une heuristique d'ordonnancement dynamique dont le critère de validité est le débit.

Dans la littérature [133], nous distinguons deux approches d'ordonnancement : en ligne et hors ligne. Un ordonnancement hors ligne signifie que la séquence d'ordonnancement est prédéterminée à l'avance : dates de début d'exécution des tâches, de préemption/reprise éventuelle. En pratique, l'ordonnancement prend la forme d'un plan hors ligne (ou statique), exécuté de façon répétitive (on parle aussi d'ordonnancement cyclique).
Un ordonnancement en ligne correspond au déroulement d'un algorithme qui tient compte des tâches effectivement présentes dans la file d'ordonnancement lors de chaque décision d'ordonnancement. Les ordonnanceurs en ligne peuvent reposer sur la construction de plans d'ordonnancements en cours de fonctionnement, auquel cas ils sont dits à plan dynamique. Mais plus couramment, ces ordonnanceurs sont fondés sur la notion de priorité.

Il est par ailleurs possible de transformer un ordonnancement à plan statique en ordonnancement en ligne (par exemple à priorité statique préemptif ou non préemptif, ou à priorité dynamique). Les ordonnanceurs statiques fondent leurs décisions d'ordonnancement sur des paramètres assignés aux tâches du système avant leur activation. A l'inverse, les ordonnanceurs dynamiques fondent leurs décisions sur des paramètres qui varient en cours de fonctionnement du système.

Notre approche consiste à préparer les plans statiques d'ordonnancement, habituellement calculés hors ligne, au cours de l'exécution de l'application. A l'issue d'une itération et lorsque l'estimation fait appel au partitionnement, ce dernier fait appel à son tour à l'ordonnancement pour évaluer ses résultats.
La figure 5.11 ci-dessous donne le schéma d'exécution de trois tâches dont la condition de précédence exige l'exécution de la tâche 1 puis des tâches 2 et 3.
A l'itération I_{n-1}, l'estimateur prédit par exemple un dépassement du temps d'exécution total par rapport à la contrainte. Le module de partitionnement, après vérification de l'ordonnancement trouve un résultat qui respecte la contrainte. Ce résultat de partitionnement/ordonnancement est utilisé par l'architecture dans les itérations suivantes.

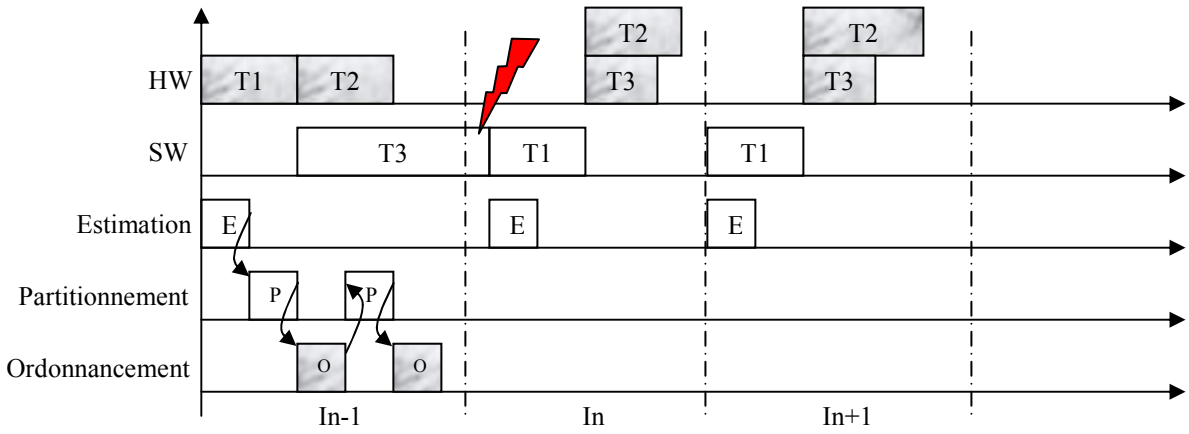


Figure 5.11: Exemple d'exécution de trois tâches avec intervention du partitionnement/ordonnancement

5.2.1 Heuristique d'ordonnancement : Principe de la méthode

L'idée de base consiste à ordonnancer les tâches d'un graphe selon trois critères : ASAP (As Soon As Possible), criticité et temps d'exécution.
En parcourant le graphe de flots de données, nous analysons tout d'abord le temps ASAP auquel la tâche est prête à s'exécuter. Si nous avons le choix entre deux tâches qui ont le même ASAP nous testons le critère de criticité qui définit la tâche qui doit être lancée à priori avant l'autre. La tâche avec la criticité la plus haute sera exécutée en premier. Si deux tâches ont toutes les deux les mêmes valeurs de ASAP et les mêmes criticités, nous comparons les temps d'exécution. La tâche ayant le temps d'exécution le plus élevé est prioritaire.

L'ordonnancement retenu opère des tâches racines du graphe vers les tâches feuilles. Lorsque les tâches sont indépendantes (pas de relation de dépendance et ne partagent pas la ressource logicielle), les tâches sont ordonnancées suivant leur date ASAP calculées à partir des dates de fin d'exécution de leurs prédécesseurs. Quand deux ou plusieurs tâches sont dépendantes, elles sont ordonnancées en tenant compte des fins d'exécutions des prédécesseurs et des critères de criticité et de temps d'exécution. Un cas particulier intervient pour le processeur: il n'exécute qu'une seule tâche à la fois, aussi l'ordonnancement des nœuds du graphe alloués au processeur doit tenir compte de la séquentialité introduite. Il est donc nécessaire dans ce cas de faire intervenir la disponibilité du processeur dans l'évaluation de la date au plus tôt d'exécution d'une tâche.

Ci-dessous l'algorithme de la méthode d'ordonnancement:

1. Calcul de criticité de toutes les tâches;

Pour toutes les tâches
{
Calculer la criticité de chaque tâche:
a. pour les tâches HW : Criticité = 0;
b. pour les tâches SW qui n'ont pas de successeurs: Criticité = 0;
c. Pour les autres tâches : Criticité = max (temps d'exécution HW des successeurs)
}

2. Ordonnancement du graphe

Tant qu'il y a des tâches non ordonnancées:
{
Pour les tâches prêtes à l'exécution: (leurs prédécesseurs sont ordonnancées ou la tâche n'a pas de prédécesseurs)
{
Calculer ASAP de chaque tâche:
ASAP = Temps d'exécution,
Pour tous les prédécesseurs de la tâche
{
ASAP = Max ($ASAP_{prédécesseurs}$ + Temps d'exécution)
}
Trier les tâches suivant leurs valeurs croissantes de ASAP
Si la valeur minimum de ASAP ne correspond qu'à une seule tâche
Alors Ordonnancer la tâche qui a Min (ASAP)
Sinon
{
Rechercher le maximum de criticité
S'il y a une seule tâche qui correspond à Max (Criticité)
Alors Ordonnancer la tâche qui a Max (Criticité)
Sinon
{
Rechercher le maximum du temps d'exécution
Ordonnancer la tâche qui a le Max (Temps d'exécution)
}
}
}
}

Comme le montre l'algorithme précédent, nous commençons par comparer les ASAPs de toutes les tâches prêtes à s'exécuter.

- **Calcul des ASAPs :**

L'ASAP d'une tâche (As Soon As Possible) est l'instant à partir duquel la tâche peut s'exécuter.
A chaque fois qu'on ordonnance une tâche, les ASAP des autres tâches doivent être mis à jour.

Exemple :

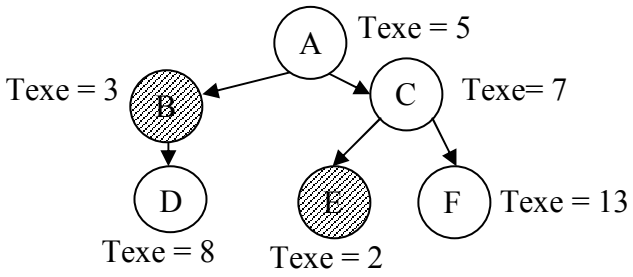


Figure 5.12: Exemple d'un DFG pour le calcul des temps ASAP

Dans cet exemple, les tâches A, C, D, F sont allouées au processeur, les tâches B et E sont exécutées par le reconfigurable. A l'issue de l'exécution de la tâche A, les dates ASAP des tâches B et C sont :

ASAP (C) = 5

ASAP (B) = 5

Si on exécute la tâche C après la tâche A, et en même temps on exécute la tâche B sur le reconfigurable, les dates ASAP des autres tâches sont :

ASAP (F) = 5 + 7 = 12

ASAP (E) = 5 + 7 = 12

ASAP (D) = max (5 + 3; 5 + 7) = 12.

Pour la tâche D on doit tenir compte de la date de disponibilité du processeur, soit après la fin de l'exécution de la tâche C.

- **Calcul des criticités :**

Ce critère favorise les branches qui contiennent des tâches matérielles afin de privilégier le parallélisme entre le logiciel et le matériel.
Si nous avons le choix entre deux tâches logicielles dont l'une précède une tâche matérielle, alors nous avons intérêt à lancer cette tâche logicielle en vue de favoriser le parallélisme entre son successeur et les tâches logicielles.
Dans la suite nous donnons quelques exemples d'ordonnancements pour illustrer ce principe.

Exemple 1: La tâche matérielle est une feuille

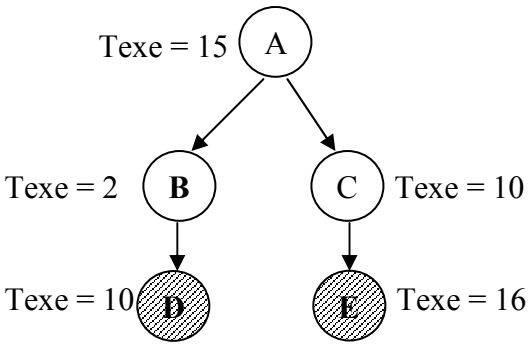


Figure 5.13: Exemple illustratif de la criticité

Dans cet exemple, les deux tâches B et C sont ordonnançables après la tâche A. La tâche C est plus critique que la tâche B et les valeurs de criticités sont :

Criticité (C) = 16
Criticité (B) = 10

Nous comparons la valeur du temps d'exécution obtenu en considérant les deux ordres possibles d'exécution des tâches : d'abord B puis C (Figure 5.14, 1er cas) et ensuite C puis B (Figure 5.14, 2ème cas).

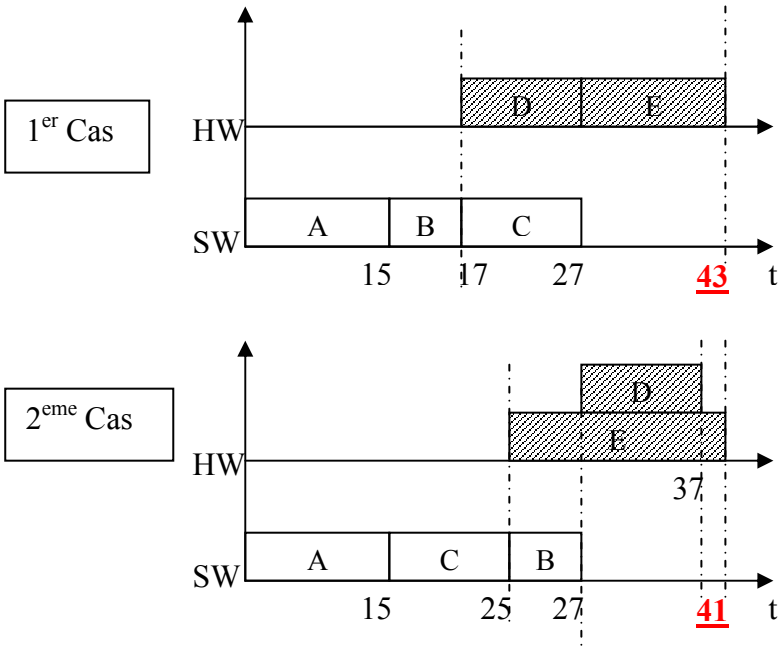


Figure 5.14: Ordonnancements obtenus en fonction de l'ordre d'exécution des tâches

Nous remarquons ainsi sur cet exemple que l'ordonnancement de la tâche qui possède la plus grande criticité donne le meilleur résultat.

Il est clair que la tâche C est plus critique car elle est succédée par la tâche matérielle qui a le temps d'exécution le plus long. Ceci est indépendant des valeurs de temps d'exécution de B et C.
L'utilisation de la criticité peut donner de meilleurs résultats qu'en considérant le chemin de longueur critique CPL (Critical Path Length).
Considérons l'exemple modifié de la figure 5.15.

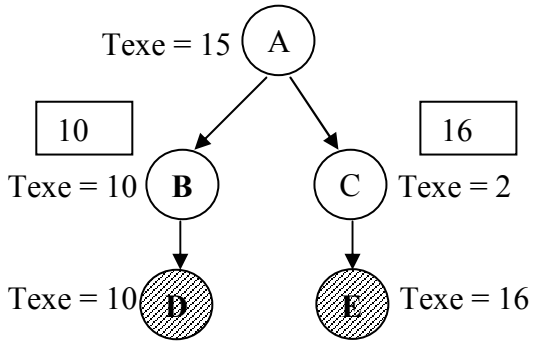


Figure 5.15: Exemple modifié illustrant le principe de la criticité

Sur la figure 5.16, on remarque que contrairement au choix guidé par la longueur des chemins, la tâche C, une fois qu'elle est exécutée permet un meilleur taux de parallélisme et donc donne un temps d'exécution global plus faible. La notion de criticité est basée sur cette remarque.

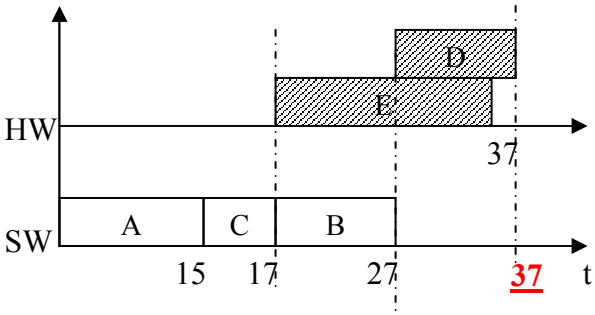


Figure 5.16: Ordonnancement tenant compte des criticités

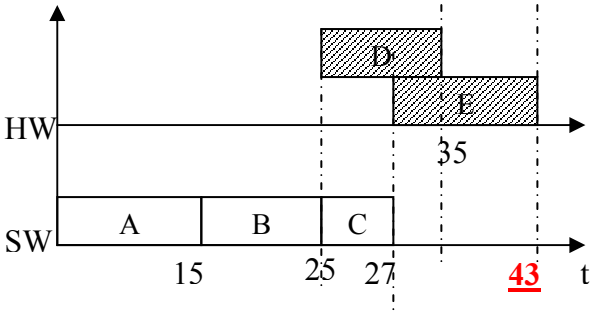


Figure 5.17: Ordonnancement tenant compte des CPL

Notons par ailleurs que la méthode basée sur le CPL n'est pas adaptée au cas des architectures parallèles.

Exemple 2 :

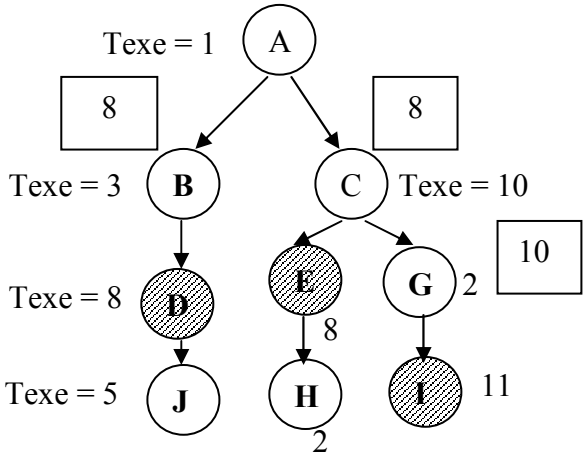


Figure 5.18: Exemple d'un DFG avec des tâches à criticités identiques

Les tâches B et C de l'exemple de la figure 5.18 ont les mêmes temps ASAP ainsi que les mêmes valeurs de criticités (Criticité (B) = Criticité (C)= 8). Dans ce cas de figure, notre approche d'ordonnancement favorise la tâche qui a le temps d'exécution le plus grand (ici c'est la tâche C qui est la plus prioritaire). Nous comparons la valeur du temps d'exécution obtenu en considérant les deux ordres possibles d'exécution des tâches : d'abord C puis B (Figure 5.19) et ensuite B puis C (Figure 5.20).

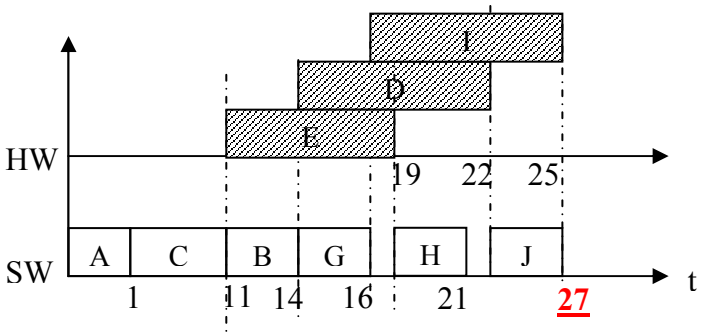


Figure 5.19: Ordonnancements commençant par la tâche C

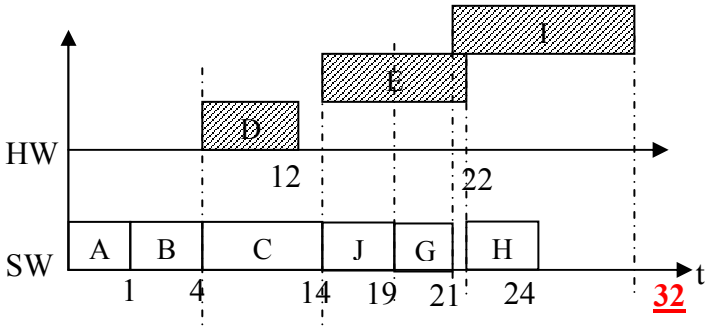


Figure 5.20: Ordonnancements commençant par la tâche B

Ces deux exemples de DFG montrent l'efficacité de notre heuristique d'ordonnancement. L'exemple 1 met en évidence la qualité d'ordonnancement par criticité. Dans l'exemple 2, nous montrons l'intérêt du troisième critère de l'approche : le maximum du temps d'exécution. Ainsi notre approche avec ses trois critères semble assez robuste pour plusieurs cas de figures de DFG différents.

5.2.2 Conception de l'ordonnanceur

L'objectif d'une implémentation matérielle de l'ordonnanceur est de réduire son temps d'exécution pour permettre une réactivité en adéquation avec l'objectif d'une qualité de service élevée. Une implémentation logicielle de l'ordonnanceur est très coûteuse en temps car l'ordonnanceur peut être plusieurs fois sollicité par le partitionnement et par l'estimation en vue de la migration de tâches.

La méthode d'ordonnancement telle que présentée ci-dessus tient compte de la structure du DFG à ordonnancer (les dépendances entre les tâches du graphe), des implémentations des tâches (logicielles ou matérielles) et de leurs temps d'exécution associés aux implémentations.
L'architecture matérielle proposée est construite en utilisant un IP (Intellectual Property) générique associé à chaque tâche. L'IP d'une tâche opère à partir de ses caractéristiques : Texe, ASAP, CT (Criticité) et son type d'implémentation.

Ces IPs sont connectés à deux autres modules matériels de l'ordonnanceur:
i) le gestionnaire des tâches logicielles (STM: Software Tasks Manager).
ii) le gestionnaire de la mise à jour du DFG (DFG Up: DFG Updating).
Si nous prenons l'exemple suivant d'un DFG à trois tâches T1, T2, T3, l'architecture globale de l'ordonnanceur est donnée par la figure 5.21.

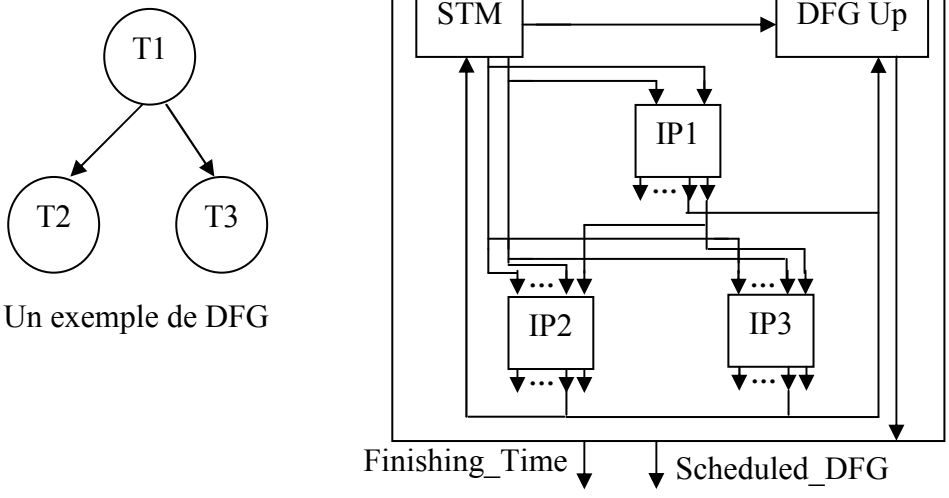


Figure 5.21: L'architecture matérielle de l'ordonnanceur correspondante à un DFG de trois tâches

A chaque tâche est associée une instance d'IP qui est reliée à un module STM et un module DFG Up.

Les IP sont connectées suivant les arcs de dépendances du DFG.
Le module STM permet la séquentialité des exécutions des tâches allouées au processeur. Le module DFG Update permet de traduire le résultat des choix d'ordonnancement. Les signaux d'entrée indiquent pour chaque tâche si son implémentation est logicielle (SW/Tâche) et son temps d'exécution (TEXE/ Tâche). Le signal de sortie scheduled_DFG indique la fin de l'ordonnancement. Les signaux internes de l'ordonnanceur sont détaillés dans les paragraphes suivants qui présentent chaque sous-module de l'ordonnanceur.

a) IP générique par tâche

L'IP décrit l'algorithme d'ordonnancement local à la tâche. Elle est détaillée dans la figure 5.22.

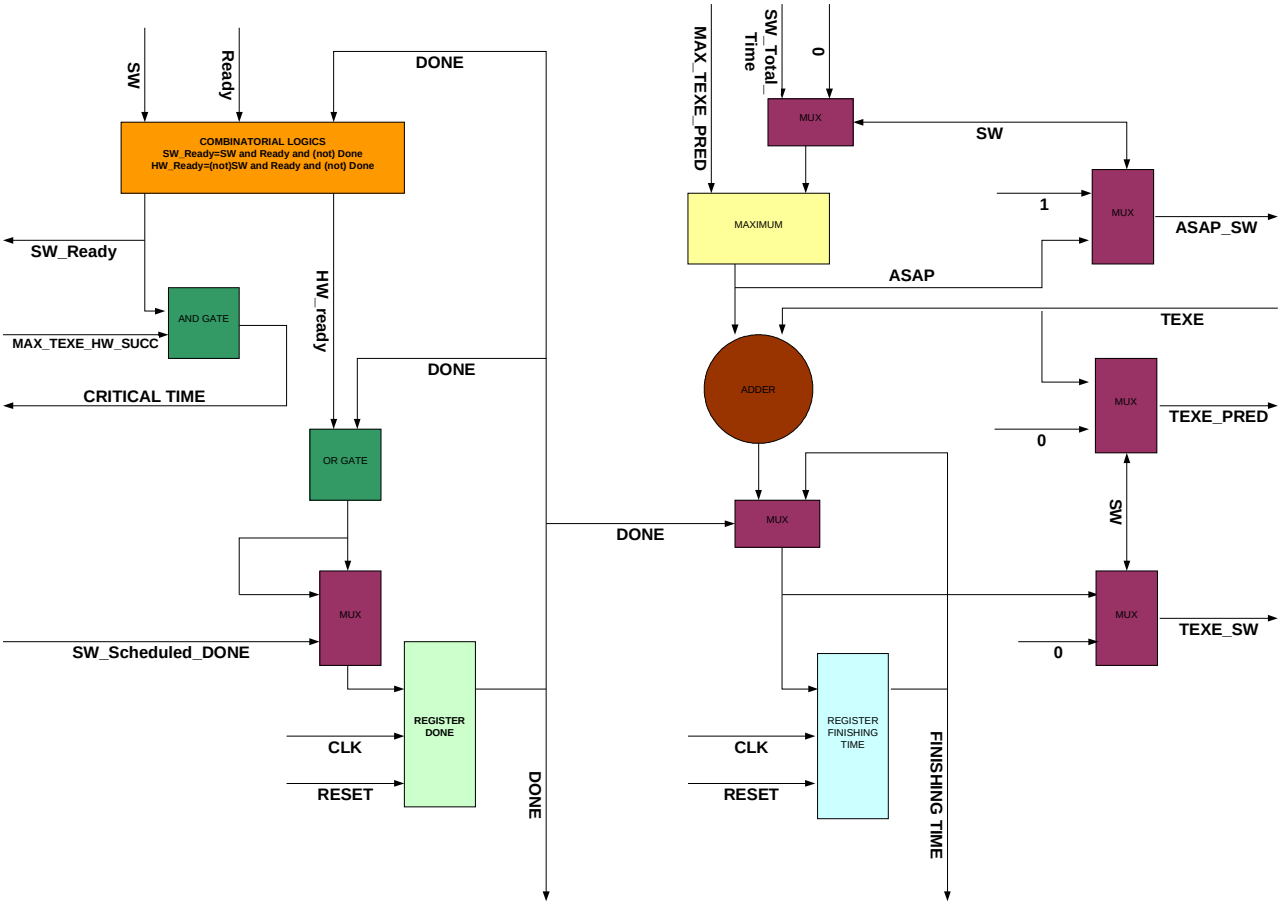


Figure 5.22 : L'architecture de l'IP générique de l'ordonnanceur

Nous trouvons en entrée de chaque IP l'implémentation retenue de la tâche (SW=1, la tâche est logicielle, sinon elle est matérielle) et le temps d'exécution correspondant (TEXE). Ces deux informations proviennent du module de

partitionnement (dans le cas d'une évaluation d'un résultat de partitionnement) ou bien du module d'estimation (dans le cas d'une évaluation du temps d'exécution total estimé).
Le temps ASAP est donné par le signal d'entrée MAX_TEXE_PRED dans le cas d'une tâche matérielle. Si l'implémentation de la tâche est logicielle, le temps où le processeur est disponible (SW_Total_Time) est également considéré pour obtenir le maximum des dates de fin d'exécution des prédécesseurs de chaque tâche. Dans le cas d'une tâche logicielle, le temps ASAP sera ainsi le maximum de deux signaux d'entrée : MAX_TEXE_PRED et SW_Total_Time. Ce dernier provient du sous-module STM qui modélise l'exécution séquentielle des tâches sur le processeur.
Le temps de la fin d'exécution d'une tâche (FINISHING_TIME) est la somme de TEXE et du temps ASAP. La valeur de ce temps doit être mémorisée après l'ordonnancement. Ceci explique le retour de cette valeur depuis son registre (Register Finishing Time) après la validation du signal DONE.
Une tâche peut avoir deux modes de type "Ready" : l'un matériel (HW_Ready) et l'autre logiciel (SW_Ready). Dans le premier mode, le signal HW_Ready est validé suivi du signal DONE après un cycle d'horloge. Dans le deuxième mode, le signal SW_Ready est validé et la validation du signal DONE dépendra du signal d'entrée SW_Scheduled_DONE qui provient du STM et qui indique que la tâche est ordonnancée. La tâche est prête à l'exécution lorsque le signal d'entrée Ready est validé c'est-à-dire lorsque tous les signaux DONE des tâches prédécesseurs sont valides.
Le sous-module STM a besoin de trois informations pour effectuer l'ordonnancement: l'ASAP, le temps de la fin d'exécution (FINISHING_TIME) et la criticité (CRITICAL_TIME). Lorsque le signal SW_Ready est valide, le signal CRITICAL_TIME est égal au signal MAX_TEXE_HW_SUCC (qui est le maximum des temps d'exécution des tâches matérielles successeurs). Ce dernier signal est aussi fourni par la partie logicielle (Partitionnement ou estimation).
Le signal ASAP_SW est égal au temps ASAP de début d'exécution pour les tâches logicielles et '1' (cycle) pour les tâches matérielles. Le signal TEXE_SW est le temps de la fin d'exécution de la tâche. Ce temps permet de mettre à jour le signal SW_Total_Time si la tâche est ordonnancée.
Les trois signaux : TEXE_PRED, FINISHING_TIME et DONE définissent les communications entre les IPs du DFG.

b) STM : Gestionnaire des tâches logicielles

L'objectif de ce sous-module est d'ordonnancer les tâches logicielles suivant l'algorithme donné ci-dessus. La figure 5.23 présente l'architecture du STM.
Le signal d'entrée ASAP_SW représente les temps ASAP de toutes les tâches. Le signal CRITICAL_TIME représente les criticités de toutes les tâches. Le signal SW_Ready représente tous les signaux Ready de toutes les tâches logicielles. Le signal MIN_ASAP_TASKS représente toutes les tâches "Ready" et qui ont les mêmes valeurs minium de temps ASAP. Le signal

MAX_CT_TASKS représente toutes les tâches "Ready" et qui ont les mêmes maximum de criticité. Les tâches qui présentent les deux critères précédents seront représentées par le signal Tasks_Ready. Le signal Task_Scheduled détermine la seule tâche logicielle qui va être ordonnancée. Avec ce signal, il est possible de choisir la bonne valeur du signal TEXE_SW et par la suite de fixer la nouvelle valeur du signal SW_Total_Time. Un cycle d'horloge est nécessaire pour ordonnancer une tâche logicielle.

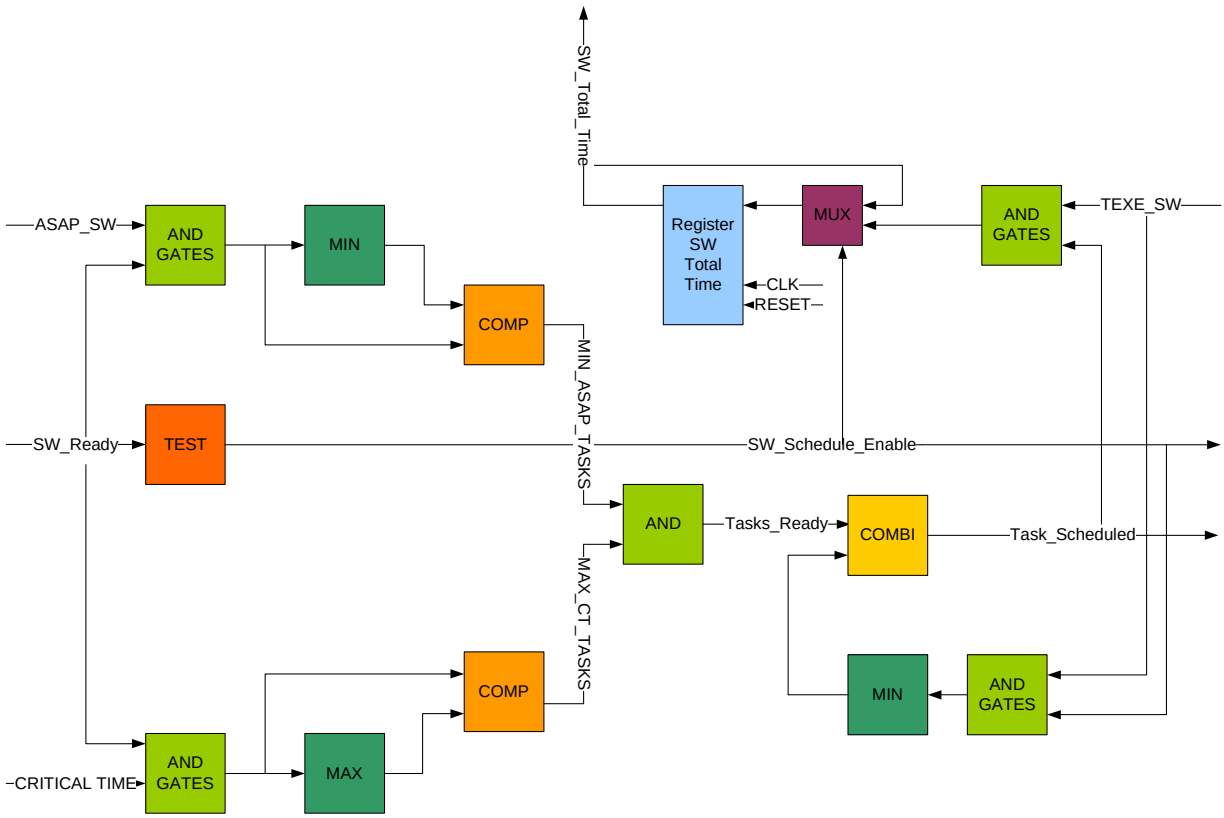


Figure 5.23: L'architecture de sous-module STM de l'ordonnanceur

c) DFG Updating : Mise à jour du DFG

Le rôle du sous-module DFG Updating est d'introduire des relations de dépendance entre les tâches ordonnancées afin de tenir compte des séquentialités induites par les choix d'ordonnancement. Les dépendances entre les tâches peuvent être représentées par une matrice de bits (0 ou 1) dont les lignes sont les prédécesseurs de la tâche et les colonnes sont les successeurs. En prenant l'exemple du DFG de trois tâches, la figure 5.24 montre la représentation matricielle des dépendances des tâches avant et après ordonnancement. Si l'exécution de la tâche T3 doit suivre celle de la tâche T2, un bit est ajouté dans la table pour décrire cette séquentialité.

Avec la représentation matricielle, la mise à jour des dépendances peut être faite en 1 cycle d'horloge.
La matrice de dépendance originale du DFG n'est pas modifiée. C'est uniquement la matrice de NEW SUCCESSORS qui est mise à jour et est remise à zéro au début de chaque nouvel ordonnancement. Le DFG final après ordonnancement de toutes les tâches est la combinaison des deux matrices, correspondant au OU logique des deux structures.

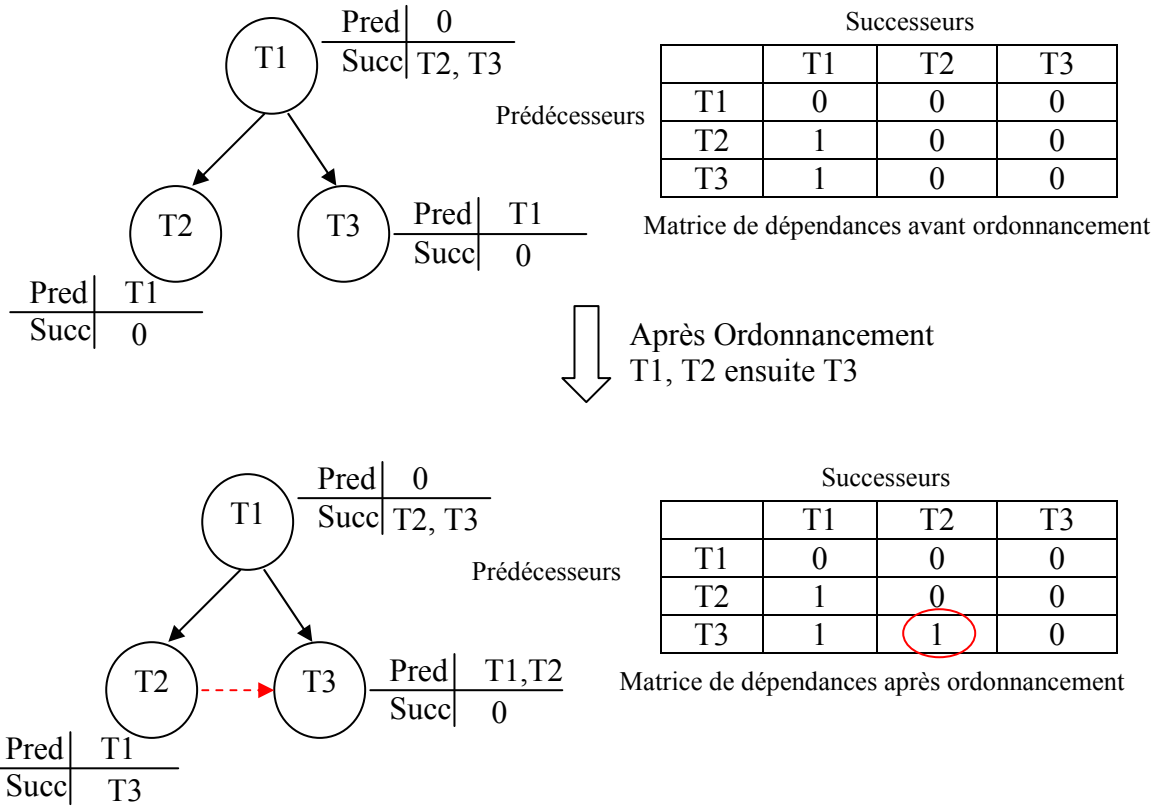


Figure 5.24 : Représentation matricielle de dépendances d'un DFG

La figure 5.25 présente l'architecture de ce sous-module. La table NEW SUCCESSORS est directement utilisée par le module "exécutif" pour contrôler l'exécution des tâches conformément au résultat d'ordonnancement trouvé.

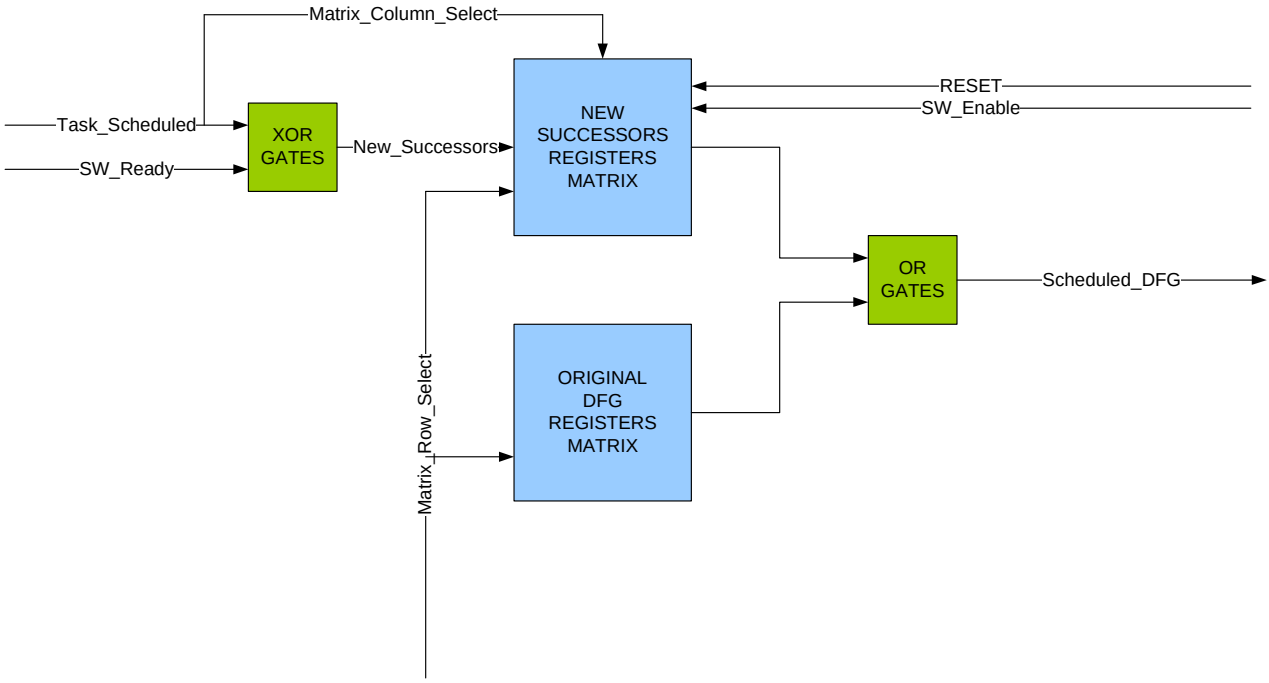


Figure 5.25: L'architecture de sous-module DFG Updating de l'ordonnanceur

Remarque : Avec un VIRTEX II Pro vp 100 et avec une taille des données égale à 10 bits l'ordonnanceur n'utilise que 3% de ressources pour un DFG de 20 tâches et avec un cycle d'environ 15ns.

Conclusion

Nous avons présenté dans une première partie de ce chapitre le principe de notre méthode de partitionnement logiciel/matériel. Elle est basée sur un concept de migration (réallocation) des tâches: migration directe des tâches visant à accélérer leurs traitements et migration inverse qui a pour objectif de libérer des ressources afin d'autoriser des prochaines migrations directes.

Dans une deuxième partie de ce chapitre nous avons présenté notre heuristique d'ordonnancement des tâches d'un DFG sur les cibles logicielles et matérielles de l'architecture. Le principe de la technique d'ordonnancement proposée se base sur l'affectation des priorités aux tâches du DFG suivant trois critères: le temps ASAP, la criticité et le temps d'exécution. Des exemples de DFG test ont été étudiés pour expliquer les caractéristiques de cette heuristique.

Dans le chapitre suivant nous exposons les expérimentations de notre méthodologie de partitionnement dynamique à l'application de traitement d'image de type flot de donnée : la détection de mouvement sur un fond d'image fixe dans une caméra intelligente (ICAM).

CHAPITRE 6

Expérimentations et résultats

Introduction

Dans ce chapitre nous présentons l'outil de simulation de l'approche OPA (Operating Partitioning Algorithm) développé en SystemC ainsi que les résultats d'estimation, d'ordonnancement et de partitionnement. A la fin de ce chapitre nous comparons notre approche de partitionnement avec une formulation du problème sous forme d'un programme linéaire en nombres entiers. Mais tout d'abord nous commençons par présenter l'application de traitement d'image ICAM qui sert de support à cette expérimentation.

6.1 Présentation de l'application de vidéo surveillance

L'application de vidéosurveillance considérée pour valider l'approche OPA a été proposée par le *CEA* dans le cadre du projet EPICURE. Elle est issue du projet ICAM1 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 et ayant une très forte puissance de calcul embarquée. Les applications qui ont motivé ce projet sont :

L'assistance au conducteur : Embarquée au sein d'un véhicule, ICAM permet de détecter des piétons (figure 6.1. a), des obstacles et rappelle au conducteur la signalisation.

- La vidéosurveillance : ICAM peut être utilisée aussi bien pour des opérations de vidéosurveillance d'un quai de gare (figure 6.1 b) que pour compter les véhicules ou détecter les accidents.

Comme illustré sur la figure 6.2, l'application effectue la détection et l'étiquetage des objets en mouvement par rapport à un fond d'image fixe (la figure 6.1 c donne l'exemple de la vidéosurveillance 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 6.3 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 N (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 de calcul de l'histogramme de l'image, CONVOLTABHISTO de calcul du 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 6.1 : Exemples d'applications du projet ICAM

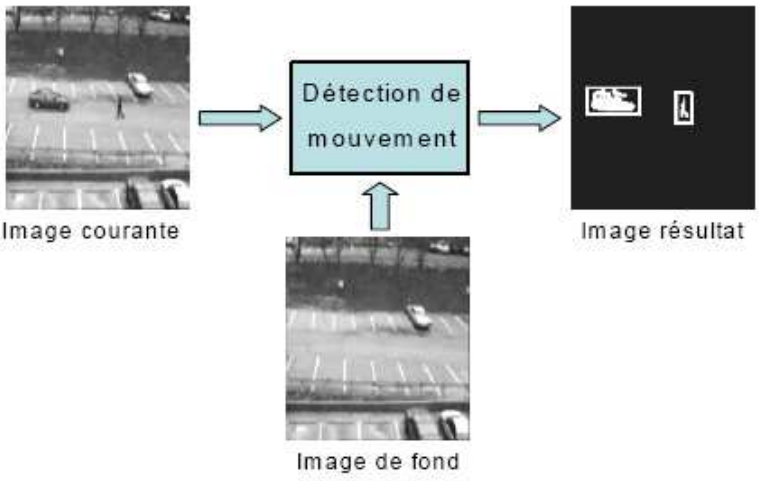


Figure 6.2 : Principe de la procédure de détection de mouvement

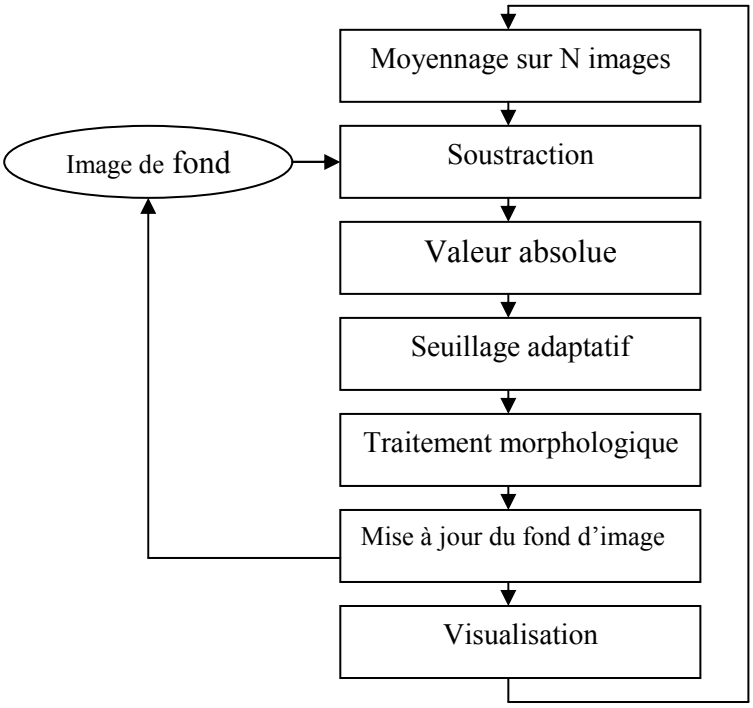


Figure 6.3 : Schéma de principe de l'application ICAM

- le traitement morphologique : permet de filtrer l'image seuillée pour faire disparaître les points isolés. Cette étape fait appel aux procédures : ERODBIN qui réalise 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 une enveloppe englobante est affichée tout autour comme illustré dans la figure 6.2 (procédure ENVELOP).

Pour expérimenter notre approche OPA sur l'application ICAM, nous avons préparé plusieurs séquences vidéo test de complexités différentes. Dans ces séquences, le nombre d'objets contenu et leur vitesse de déplacement varient pour révéler différentes situations.

6.2 Outil de simulation en SystemC de l'approche OPA

L'application ICAM comprend dix tâches. Chaque tâche peut être exécutée sur le Software (le processeur) ou bien sur le Hardware (FPGA). Pour chaque tâche nous considérons trois réalisations possibles. La modélisation en

SystemC de cette application doit comporter alors une partie logicielle et une partie matérielle.

Dans la partie logicielle, chaque tâche est représentée par une fonction lancée par le processeur chaque fois que cette tâche est exécutée en Software. Pour la partie matérielle, chaque tâche est modélisée par un module matériel indépendant. Ce dernier communique avec les autres tâches à l'aide des entrées\sorties.

L'exécution de l'application est contrôlée par une fonction appelée « exécutif » (voir définition OPA). Chaque module logiciel ou matériel doit communiquer directement avec cette fonction.

Les données traitées peuvent être échangées entre les différentes tâches à travers une mémoire. Toute communication entre le logiciel et le matériel doit passer par cette mémoire. Pour l'application ICAM, l'échange à travers la mémoire concerne les données d'images traitées. La communication avec la fonction exécutif concerne les données de contrôle (identité de la tâche, interruption,..), le temps d'exécution de la tâche et la valeur du paramètre de corrélation.

Le temps d'exécution est mesuré avec l'horloge du processeur de la plateforme de simulation. Le temps d'exécution est mesuré avec des instructions en assembleur qui permettent de calculer le nombre de cycle mis par la tâche, multiplié par la période du processeur. Ainsi nous obtenons un temps d'exécution avec une précision de l'ordre de la microseconde. Nous mesurons uniquement les temps d'exécution en logiciel. Les temps d'exécution matériels sont déduits du temps logiciel en les multipliant par des rapports constants. Des exemples des valeurs de ces rapports sont donnés par la formule suivante :

Texe SW= 0.2 * Texe HW1 = 0.1 Texe HW2 = 0.06 Texe HW3.

6.2.1 Une approche complètement matérielle de l'OPA

La première modélisation System de l'OPA a pour objectif d'analyser le comportement global du système. Tous les algorithmes de l'OPA sont implémentés sous forme des modules matériels indépendants (voir figure 6.4).

La communication entre ces modules est assurée par des FIFO (First In First Out). Chaque fonction de l'OPA est synchronisée sur l'événement de la FIFO et son temps d'exécution est mesuré par la même méthode que pour une fonction de l'application ICAM. Chaque FIFO réalise le transfert de donnée pour chaque direction de communication. Dans la configuration de la figure 6.4, il n'y a pas d'interruption des tâches sur le processeur : les tâches matérielles en exécution sont gérées par le module matériel « exécutif ».

Pour une tâche qui est prête à être exécutée sur le processeur, l'exécutif attend la libération du processeur s'il n'est pas libre. A la fin de l'exécution de l'ensemble des traitements sur une image, l'OPA met à jour la base de données et effectue les estimations du temps d'exécution.

L'inconvénient de ce modèle est le nombre de FIFO de communication : la conception de ce système n'est pas évidente. Par ailleurs l'estimateur emploie des opérations de division qui conduisent à une conception matérielle très coûteuse en ressources.

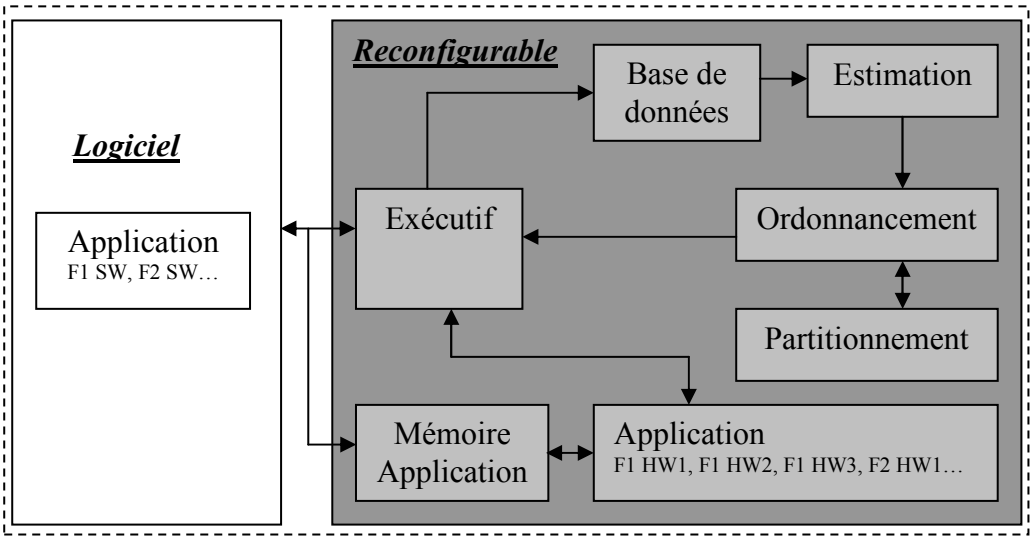


Figure 6.4 : Modélisation complètement matérielle de l'OPA

6.2.2 Une approche logicielle avec des interruptions

Les résultats d'analyse comportementale de l'approche matérielle précédente montrent qu'il faut partitionner l'approche OPA entre le logiciel et le matériel suivant la faisabilité et le coût en ressources de chacune des fonctions. Les mesures de temps moyen d'exécution logiciel (mesurées sur un processeur Centrino) des fonctions de l'OPA sont les suivantes :

- Exécutif : 0.18 ms
- Mise à jour de la base de données : 0.013 ms
- Estimation par l'approche Kppv : 0.04 ms
- Ordonnanceur : 0.51ms
- Partitionnement : 0.41ms pour un nouveau calcul et 0.002ms pour la lecture de la table de solutions.

La boucle partitionnement/ordonnancement peut être coûteuse en temps d'exécution en particulier si la solution n'est pas en tête de la liste. L'ordonnanceur est le plus critique en temps d'exécution. Une architecture dédiée pour ce dernier peut accélérer significativement son temps d'exécution. Les autres fonctions seront implémentées en logiciel vu leur complexité en matériel ou bien vue qu'elles ne sont pas très coûteuses en temps d'exécution. La figure 6.5 montre le partitionnement proposé de l'OPA.

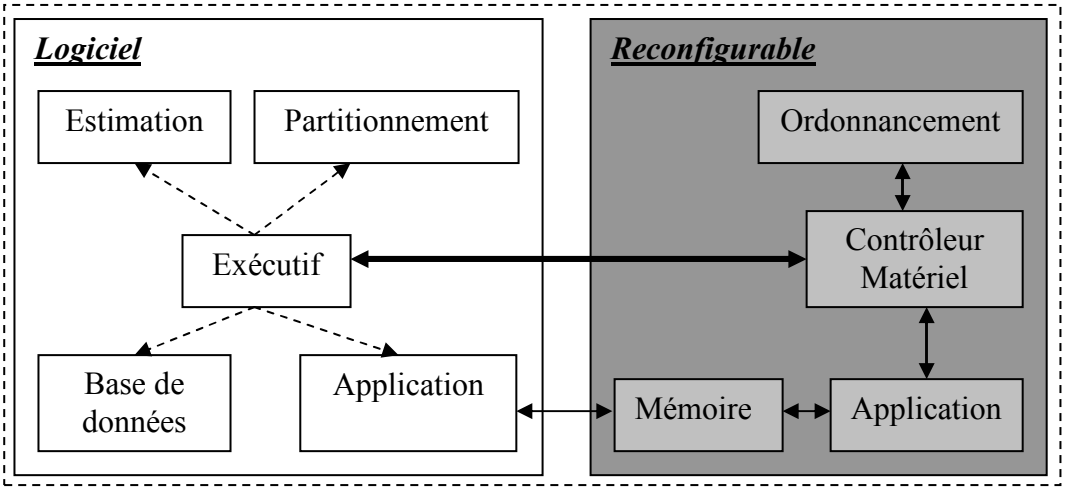


Figure 6.5 : Modélisation logiciel/matériel en SystemC de l'approche OPA

6.3 Résultats d'estimations

Pour estimer le temps d'exécution d'une tâche à l'itération suivante, il est nécessaire d'estimer tout d'abord la valeur du paramètre de corrélation dont il dépend. Donc l'erreur d'estimation du temps d'exécution est corrélée à l'erreur d'estimation de son paramètre de corrélation.

6.3.1 Estimation du paramètre de corrélation

La seule hypothèse considérée sur le comportement du paramètre de corrélation d'une fonction est qu'entre deux images, le paramètre de corrélation varie peu. Cette hypothèse est souvent vérifiée en traitement d'image tant qu'il n'y a pas de changement de scène par exemple. L'estimateur le plus simple est de multiplier par une constante le précédent paramètre de corrélation de manière à garantir une légère augmentation des estimées par rapport aux mesurées.

Nous avons testé les estimateurs suivants sur deux séquences vidéo :

Premier Estimateur: $\mathbf{Pc_{estimé}(n) = \alpha * Pc_{mesuré}(n-1)}$ avec ($\alpha > 1$) (6.1)

Deuxième Estimateur :

$$\mathbf{Pc_{estimé}(n)} = \mathbf{Pc_{mesuré}(n-1)} + \frac{1}{n}\sum_{i=1}^{n}\left[Pc_{mesuré}(n-i) - Pc_{mesuré}(n-i-1)\right] \quad (6.2)$$

La courbe ci-dessous (Figure 6.6) donne la moyenne de l'erreur entre les valeurs estimées et mesurées du nombre des pixels blancs dans une image pour la fonction étiquetage sur 239 images pour 6 valeurs de paramètres :

Estimateur 1 : $\mathbf{Pc_{estimé}(n) = 1.5 * Pc_{mesuré}(n-1)}$ (6.3)

Estimateur 2 : $\mathbf{Pc_{estimé}(n) = 1.1 * Pc_{mesuré}(n-1)}$ (6.4)

Estimateur 3 : $\mathbf{Pc_{estimé}(n) = 1.05 * Pc_{mesuré}(n-1)}$ (6.5)

Estimateur 4 : $\mathbf{Pc_{estimé}(n) = 1.01 * Pc_{mesuré}(n-1)}$ (6.6)

Estimateur 5 :

$$\mathbf{Pc_{estimé}(n)} = \mathbf{Pc_{mesuré}(n-1)} + \frac{1}{n}\sum_{i=1}^{n=7}\left[Pc_{mesuré}(n-i) - Pc_{mesuré}(n-i-1)\right] \quad (6.7)$$

Estimateur 6 :

$$\mathbf{Pc_{estimé}(n)} = \mathbf{Pc_{mesuré}(n-1)} + \frac{1}{n}\sum_{i=1}^{n=12}\left[Pc_{mesuré}(n-i) - Pc_{mesuré}(n-i-1)\right] \quad (6.8)$$

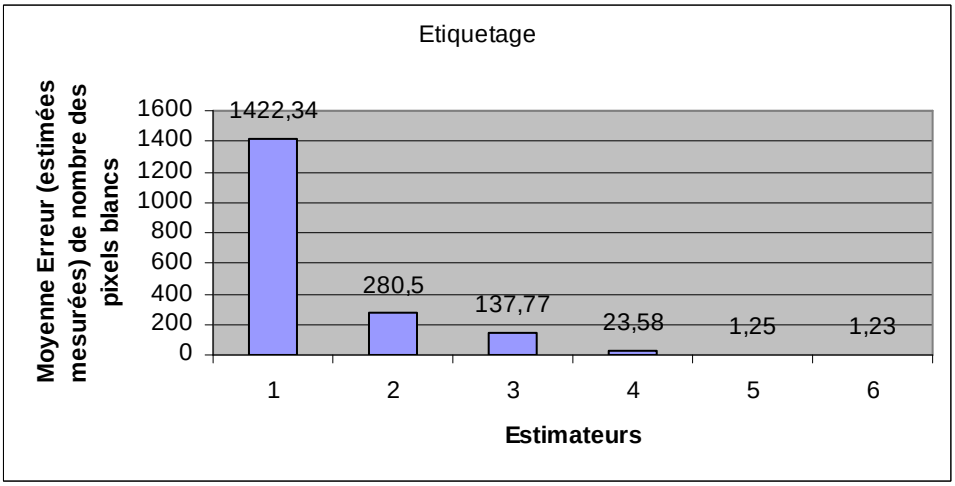


Figure 6.6 : Comparaison des résultats des estimateurs du paramètres de corrélation

Il est clair que l'estimateur 6 est le plus précis. Mais ce dernier (idem pour l'estimateur 5) peut avoir des erreurs (estimées – mesurées) négatives. Une sous-estimation de la valeur du paramètre de corrélation engendre nécessairement une sous-estimation du temps d'exécution de la fonction. Et comme nous l'avons indiqué précédemment une sous-estimation du temps d'exécution peut être la cause d'une perte de données à traiter (une image dans notre cas).

Pour l'application ICAM, l'estimateur 2 semble être un bon compromis pour ne pas sous-estimer le paramètre de corrélation ni le sur-dimensionner.

6.3.2 Estimation du temps d'exécution

6.3.2.1 Estimation par la méthode des k plus proches voisins

La qualité des estimées avec la méthode Kppv dépend des valeurs de ses paramètres. En fait les estimations sont corrélées au nombre de voisins considérés (la valeur de k), au choix des observations, à la taille de la fenêtre sur les observations, à la fonction affectant les poids aux observations…

Pour chaque fonction à temps d'exécution variable de l'application, nous avons fait varier le nombre de voisins à considérer (k) pour simplifier la mise en œuvre de cet estimateur. Nous mesurons l'erreur entre les valeurs mesurées et les valeurs estimées. Nous considérons trois critères pour caractériser l'estimateur Kppv qui sont : le biais de l'estimation, la variance et l'écart type.

Nous avons considéré que la taille de la fenêtre (DB) sur les observations a pour valeur le nombre des voisins k, c'est-à-dire que toutes les observations sont considérées comme des voisins. Ceci peut augmenter l'erreur des estimations dans les cas suivants :

- Une valeur de k trop petite, donc une taille de fenêtre assez petite, implique la perte des observations qui sont en dehors de la fenêtre et qui peuvent être significatives. (voir figure 6.7).
- Une valeur de k trop élevée, la fenêtre sur les observations est trop grande. Ceci implique que les estimations seront bruitées par toutes les observations.

Le biais de l'estimateur est défini par l'équation suivante :

$$E(T_{esti} - T_{mes}) = \frac{1}{n}\sum_{i=1}^{n}(T_{esti}(i) - T_{mes}(i)) \tag{6.9}$$

La variance de l'estimée T_{esti} est définie par l'équation :

$$V(T_{esti}) = E[(T_{esti} - T_{mes})^2] = \frac{n\sum_{i=1}^{n}(T_{esti}(i) - T_{mes}(i))^2 - \left(\sum_{i=1}^{n}(T_{esti}(i) - T_{mes}(i))\right)^2}{n^2} \tag{6.10}$$

L'écart type est défini par : $\sigma(T_{esti}) = \sqrt{V(T_{esti})}$ (6.11)

Les résultats d'estimation avec Kppv de l'application ICAM sur une séquence test contenant 239 images sont donnés dans les tableaux suivants :

Pour la fonction « Ouverture » :

K=DB	4	6	10	14	16
Biais	0.085	0.070	0.065	0.069	0.059
Variance	0.368	0.282	0.225	0.234	0.210
Ecart type	0.606	0.531	0.475	0.484	0.458

Pour la fonction « Enveloppe » :

K=DB	4	6	10	14	16
Biais	0.263	0.2668	0.020	0.107	0.026
Variance	2.454	3.990	0.271	0.785	1.082
Ecart type	1.566	1.997	0.521	0.886	1.040

Pour la fonction « Etiquetage » :

K=DB	4	6	10	14	16
Biais	0.023	0.022	0.013	0.049	0.011
Variance	0.402	0.353	0.063	0.626	0.259
Ecart type	0.634	0.594	0.252	0.791	0.509

Les courbes ci-dessous montrent qu'une taille de fenêtre égale à 50 permet de réduire les erreurs d'estimations sur la séquence test de 239 images. A l'inverse, une taille de fenêtre trop faible augmente significativement l'erreur d'estimation.

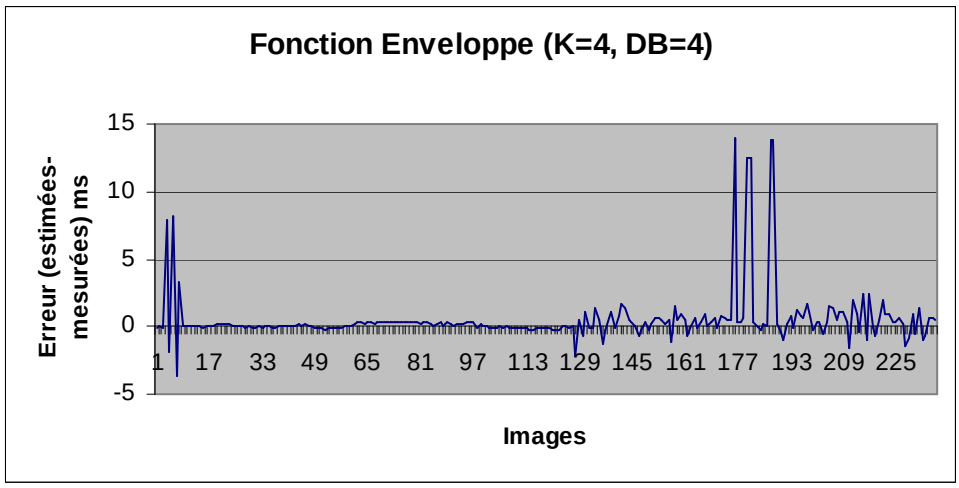


Figure 6.7 : Erreur d'estimation du temps d'exécution avec une taille de fenêtre égale à 4

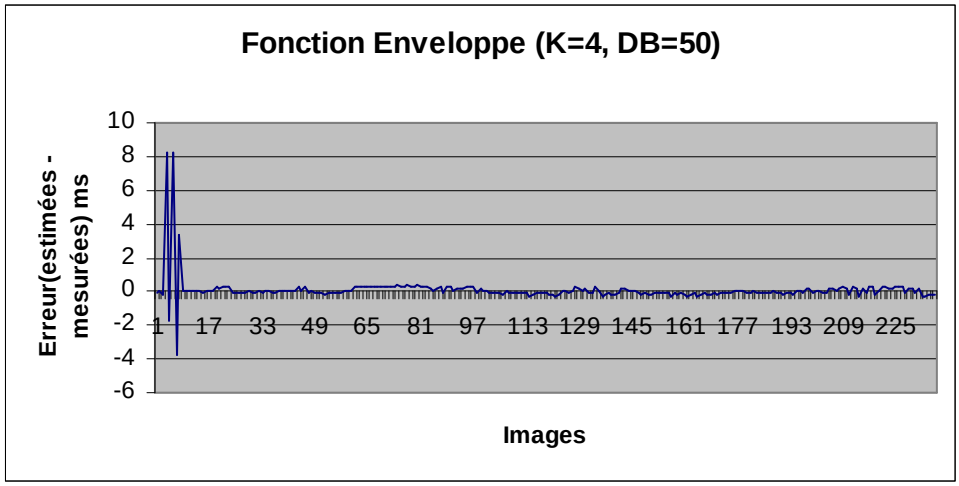


Figure 6.8 : Erreur de l'estimation du temps d'exécution avec une taille de fenêtre égale à 50

Le problème du choix de la taille de fenêtre sur les observations sera étudié en détails dans le prochain paragraphe.

D'après les courbes ci-dessous, nous remarquons que l'influence de la valeur de k (nombre des voisins) ne varie pas de la même façon pour les trois fonctions : ouverture, étiquetage et enveloppe.

Pour la fonction « ouverture », une augmentation de la valeur de k fait diminuer la valeur de biais de l'estimateur ainsi que la variance et l'écart type. Sachant que nous remarquons un minimum local faible pour le biais au niveau de la valeur 10, l'estimateur de la fonction « ouverture » ne varie pas significativement avec le nombre de voisins choisi.

Pour la fonction « étiquetage », la variation est plus claire et nous remarquons bien un minimum (qui peut être local) correspondant à la valeur 10 pour le biais, la variance et l'écart type.

Nous remarquons aussi pour cette fonction un maximum correspondant à la valeur 14 surtout pour la dispersion de l'erreur montrée par la courbe de l'écart type.

Pour la fonction « Enveloppe » le maximum de la variation et la dispersion de l'erreur de l'estimateur correspond à la valeur 6 et le minimum reste toujours pour la valeur 10.

Nous retenons d'après les expérimentations montrées sur les courbes ci-dessous que la valeur du nombre de voisins qui donne le minimum d'erreur d'estimation est égale à 10 pour une taille de fenêtre de même valeur.

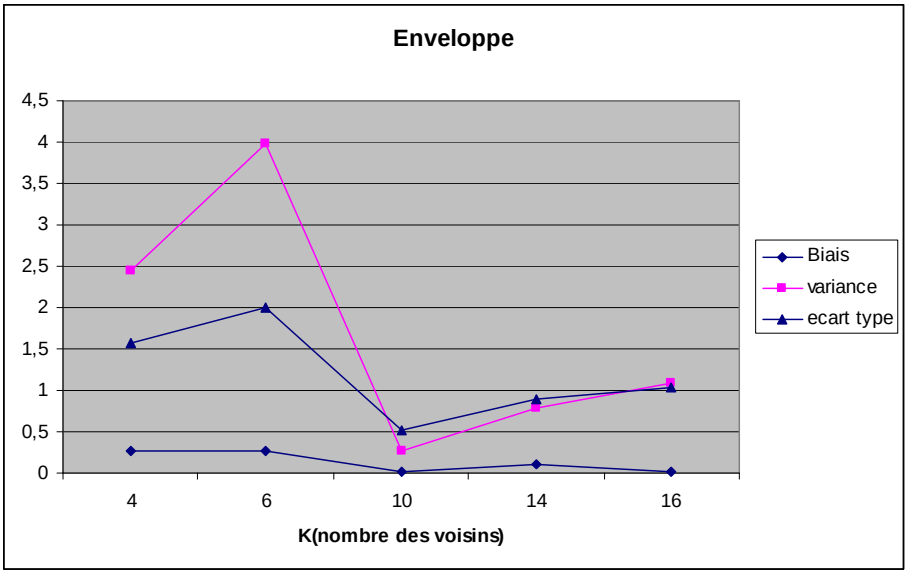


Figure 6.9 : Détermination expérimentale des paramètres de l'estimateur pour la fonction Enveloppe

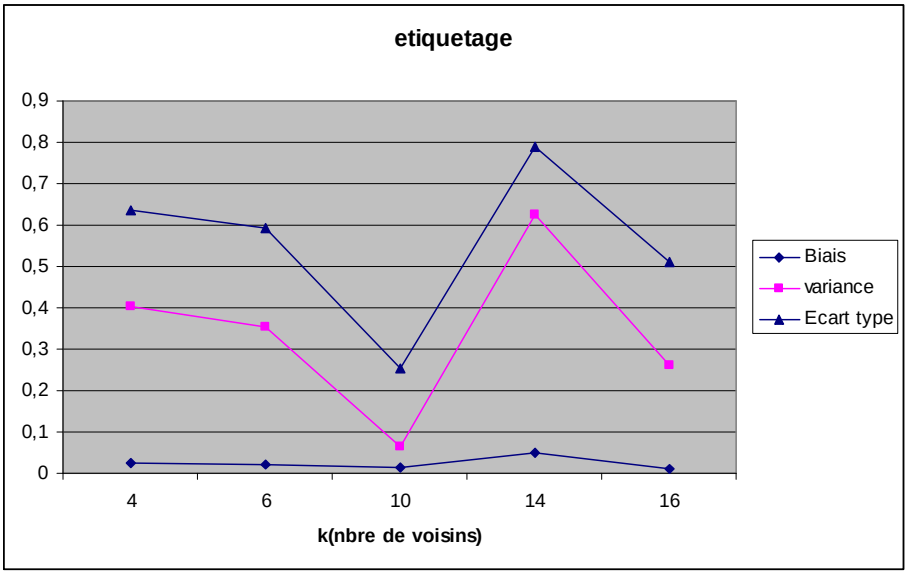


Figure 6.10 : Détermination expérimentale des paramètres de l'estimateur pour la fonction Etiquetage

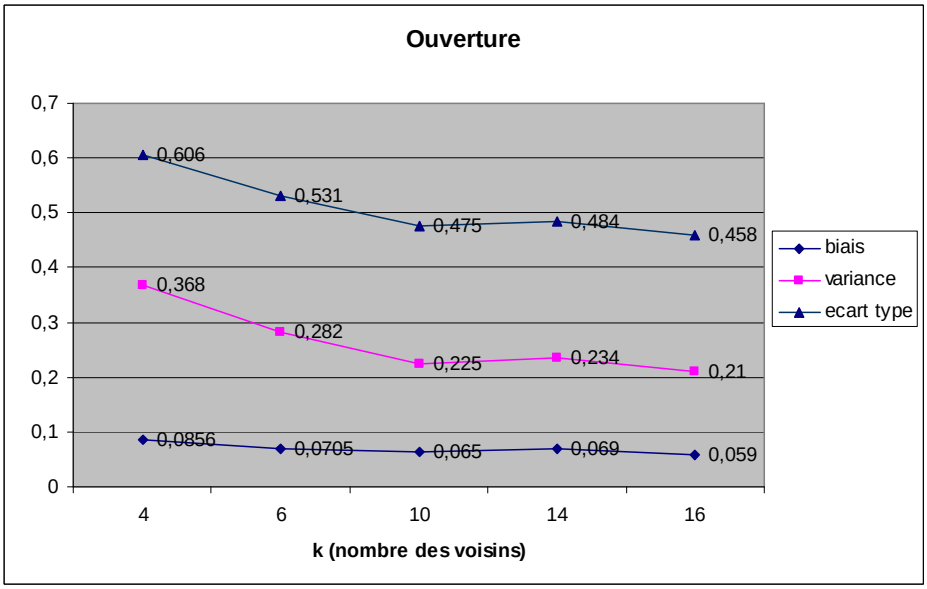


Figure 6.11 : Détermination expérimentale des paramètres de l'estimateur pour la fonction Ouverture

Vue que l'application ICAM est totalement séquentielle, les trois fonctions étudiées contribuent ensemble (par addition de leurs temps d'exécution) dans le temps d'exécution total de l'application.

Donc une étude de la moyenne de l'erreur d'estimation sur les trois fonctions est aussi significative car une fonction sous-estimée peut être compensée par une autre sur-estimée.
La courbe ci-dessous montre que la valeur **10** présente le biais le plus petit ainsi que la variance et l'écart type, mais en général pour des valeurs de k très petites la variance de l'erreur est très importante : de l'ordre de **1.5**.

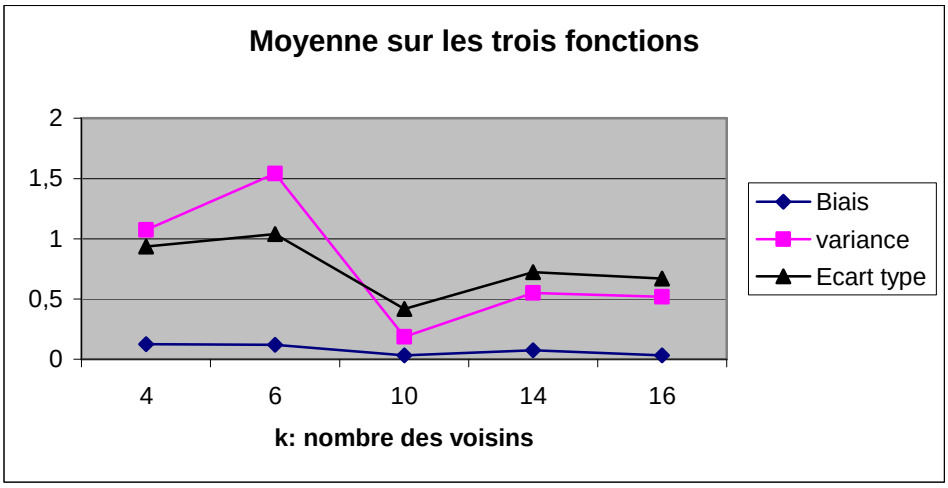


Figure 6.12 : Détermination de la valeur de K pour l'application ICAM

Maintenant, en choisissant une taille de fenêtre fixe égale à 50 et en effectuant les mêmes expérimentations, nous trouvons les résultats pour les trois fonctions récapitulés dans les tableaux ci-dessous :

Pour la fonction « Ouverture » :

K	4	6	10	14	16
Biais	0.327	0.31	0.235	0.216	0.187
Variance	0.455	0.358	0.307	0.353	0.253
Ecart type	0.675	0.598	0.554	0.594	0.503

Pour la fonction « Etiquetage » :

K	4	6	10	14	16
Biais	0.040	0.038	0.186	0.042	0.051
Variance	0.029	0.130	0.218	0.002	0.024
Ecart type	0.172	0.361	0.467	0.050	0.155

Pour la fonction « Enveloppe » :

K	4	6	10	14	16
Biais	0.084	0.295	0.328	0.627	0.268
Variance	0.712	0.357	0.381	0.849	0.404
Ecart type	0.843	0.597	0.617	0.921	0.636

Les expérimentations avec la taille de fenêtre égale 50 sont montrées sur les courbes ci-dessous.

Les trois fonctions de ICAM varient différemment en fonction de la valeur K. En se basant sur la courbe affichant la moyenne des trois fonctions, nous pouvons conclure que la valeur de biais augmente avec la valeur de K par contre la variance diminue. Donc une valeur de k trop élevée entraîne une augmentation de l'erreur et une valeur de k trop petite entraîne une augmentation de la variance de l'erreur d'estimation. Une valeur de k = 6 présente un minimum local du biais, de la variance et de l'écart type.
Contrairement aux résultats précédents lorsque la taille de fenêtre est égale au nombre des voisins, et avec une taille de fenêtre égale 50, le biais de l'estimateur varie entre 0.15 et 0.3 et le maximum de l'écart type est 0.55.
En conclusion, le couple "nombre de voisins, taille de fenêtre (6, 50)" donne un résultat d'estimation acceptable.

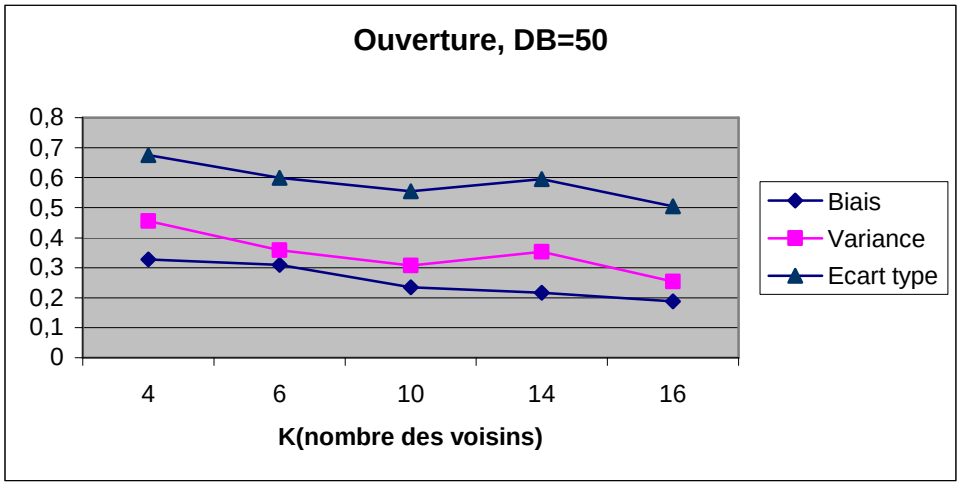


Figure 6.13 : Détermination de la valeur de k avec DB=50 pour la fonction Ouverture

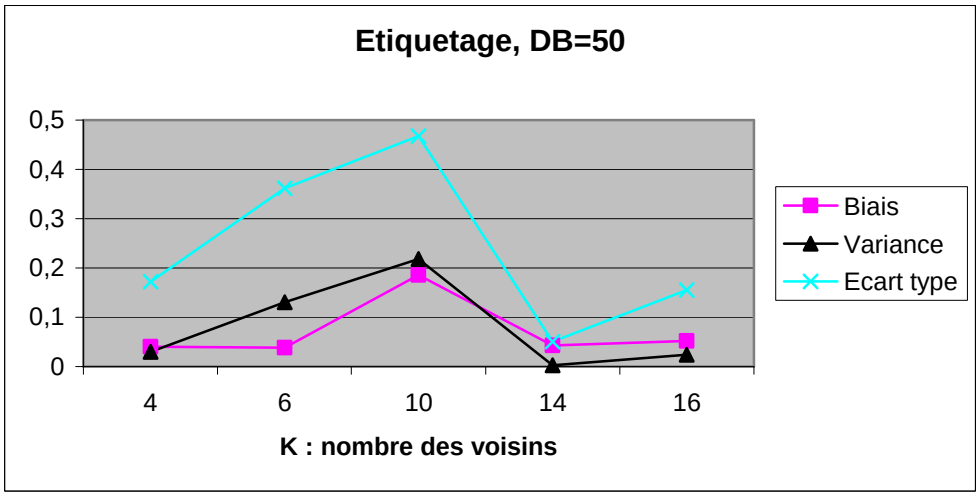


Figure 6.14 : Détermination de la valeur de k avec DB=50 pour la fonction Etiquetage

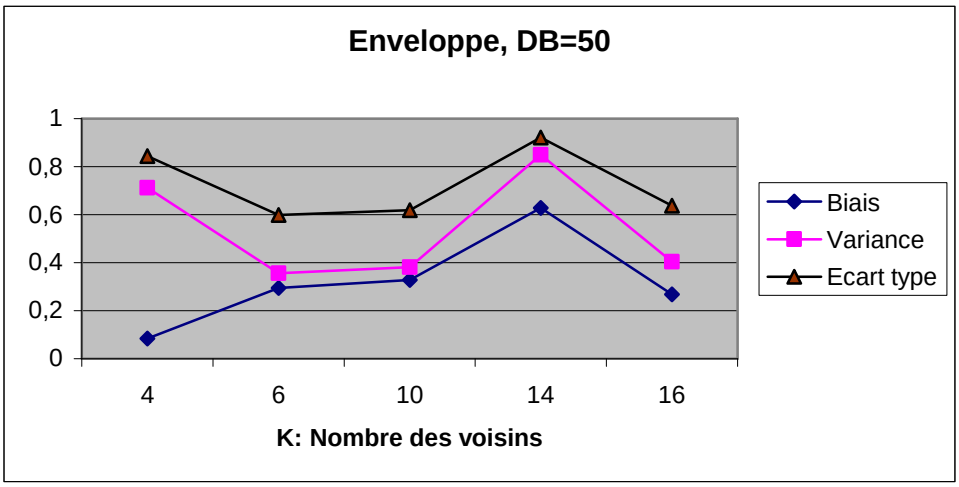


Figure 6.15 : Détermination de la valeur de k avec DB=50 pour la fonction Enveloppe

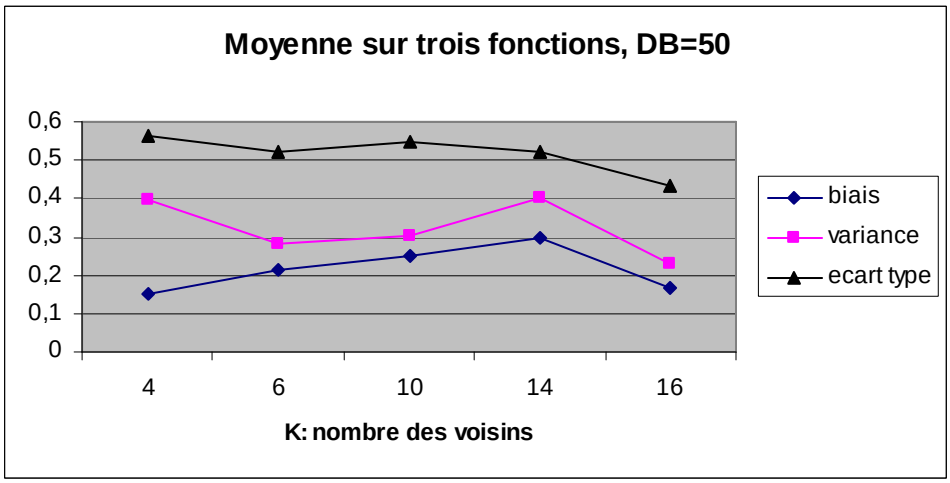


Figure 6.16 : Détermination de la valeur de k pour l'application ICAM avec DB=50

6.3.2.2 Estimation par une équation d'approximation

Une autre méthode d'estimation du temps d'exécution plus simple que l'estimateur Kppv est de modéliser hors ligne l'évolution du temps d'exécution d'une tâche par un polynôme qui lie le temps d'exécution au paramètre de corrélation. L'exemple de la fonction Enveloppe est illustré sur la figure 6.17. Ici le polynôme est une simple équation linéaire.

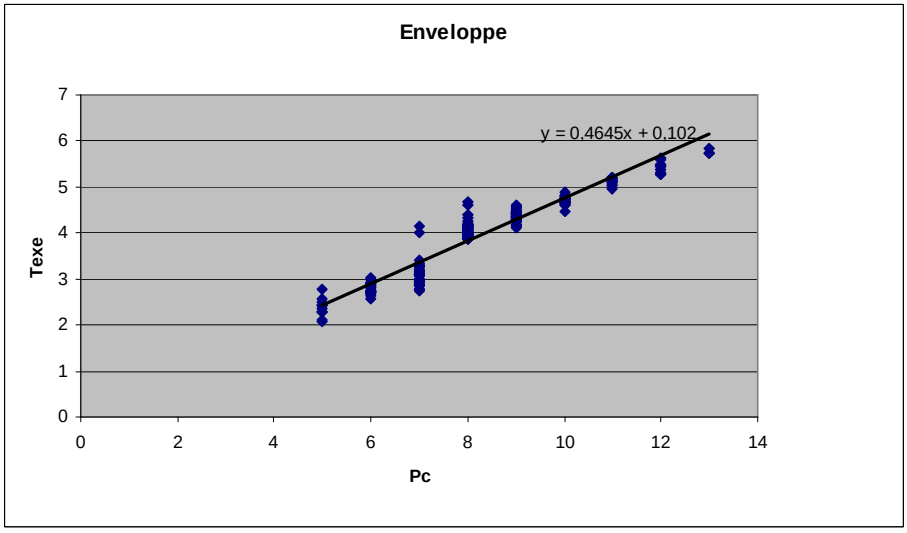


Figure 6.17 : Modélisation du temps d'exécution par une équation d'approximation

Cette méthode est beaucoup plus simple que l'approche Kppv car tout le calcul est fait hors ligne. Une fois le paramètre de corrélation prédit, le temps d'exécution de la tâche est estimé en ligne par un simple calcul d'équation.
Les résultats d'estimation sur l'application ICAM en utilisant la méthode par équation d'approximation sont irréprochables. Le paragraphe suivant présente une comparaison entre les deux estimateurs.

6.3.2.3 Etude comparative de deux estimateurs

Les deux méthodes d'estimations donnent des résultats différents pour la fonction « Enveloppe ». La méthode Kppv (figure 6.17) donne une moyenne d'erreur (sur une séquence de 239 images) plus faible que celle donnée par la méthode d'estimation suivant une équation d'approximation (figure 6.18).

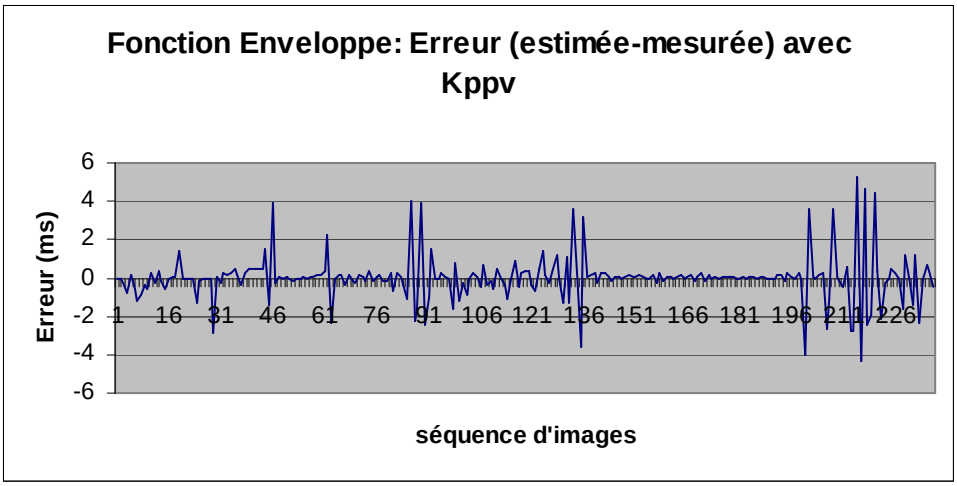


Figure 6.18 : Erreur d'estimation avec Kppv sur le temps d'exécution de la fonction Enveloppe

Cependant la variance de l'erreur obtenue par la méthode Kppv est plus importante qu'avec le deuxième estimateur. Ceci est le cas aussi pour la dispersion de l'erreur montrée par l'écart type sur le tableau ci-dessous.
Les résultats sont obtenus sur la même séquence d'images :

Méthode d'estimation	*Moyen de l'erreur*	*Variance*	*Ecart type*
Kppv	-0,006	1,426	1,194
Fonction d'interpolation	0,298	0,060	0,246

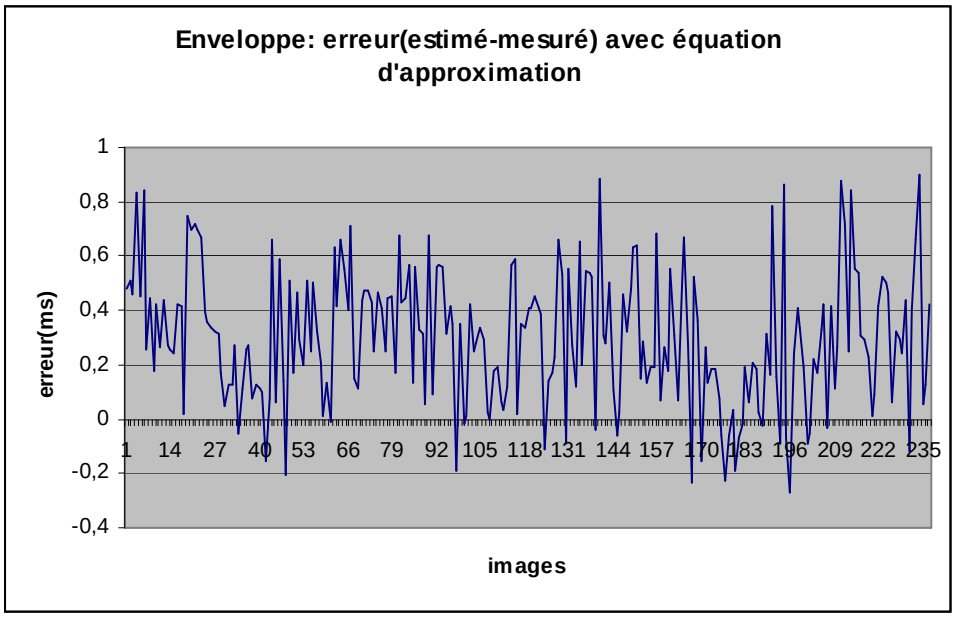


Figure 6.19 : Erreur d'estimation avec équation d'approximation du temps d'exécution de la fonction « Enveloppe »

Avec ces résultats nous ne pouvons pas décider : quel est le meilleur estimateur ? En fait, il faudrait raisonner par rapport à l'effet des erreurs d'estimation sur la méthode globale de partitionnement OPA (ce que nous verrons plus loin dans ce chapitre).

En conclusion, l'estimateur Kppv est plus compliqué à implémenter en ligne, mais il fournit un biais d'estimation faible grâce au suivi historique continu de l'évolution du temps d'exécution de chaque fonction. Un autre avantage de l'estimateur Kppv est sa généricité : en fait la méthode reste valable pour n'importe quelle fonction à temps d'exécution variable ; contrairement à la

deuxième méthode d'estimation qui nécessite une mise à jour à chaque changement de l'application.

L'estimation par équation d'approximation est très simple à implémenter en ligne mais en contre partie elle nécessite un travail préparatoire hors ligne. Le biais de ce dernier est beaucoup plus important que celui obtenu par l'estimateur Kppv mais reste faible.

6.4 Résultats d'ordonnancement

L'algorithme d'ordonnancement retenu est une heuristique mais sa complexité peut néanmoins conduire à des temps d'exécution d'un même ordre de grandeur que ceux des fonctions de l'application. Nous avons ainsi obtenu un temps d'exécution de 600 µs sur un Pentium 4 1.5 Ghz pour ordonnancer un graphe composé de 20 tâches. Sur un processeur embarqué le temps d'ordonnancement peut devenir rapidement prohibitif.

Ainsi nous avons profité de la structure locale de décision de l'algorithme d'ordonnancement pour en imaginer une réalisation matérielle. Nous décrivons succinctement dans la suite cette réalisation, une description plus complète est fournie en annexe.

6.4.1 Résultats de synthèse de l'ordonnanceur

La réalisation de l'ordonnanceur est modulaire et respecte la structure du DFG à ordonnancer. A chaque tâche dans le DFG, on associe un IP (Task IP), et tous les IPs sont liés entre eux en respectant les règles de précédences imposées par le DFG. Ce dernier est connecté au module « gestionnaire du logiciel » qui est à son tour connecté au module de mise à jour de DFG.

Le module « gestionnaire du logiciel » permet de représenter la séquentialité imposée par l'exécution des fonctions par le processeur. En entrée de chaque IP on place le temps d'exécution déterminé par le partitionnement.

Le pire cas du temps d'exécution de l'Ordonnanceur est lorsque toutes les tâches de l'application sont implémentées en logiciel. Pour notre application : ICAM avec les tâches virtuelles, nous avons en total 20 tâches. Une tâche est ordonnancée à chaque cycle d'horloge.

Le temps d'exécution logicielle de l'Ordonnanceur pour la même application et sur un Pentium 4 1.5 GHz est de 600 µs.

L'architecture matérielle proposée pour l'Ordonnanceur est donc très efficace même si elle est intégrée dans des systèmes embarqués à fréquence d'horloge relativement faible.

L'architecture de l'Ordonnanceur proposé est entièrement synthétisable. La synthèse a été faite par les outils de XILINX Virtex II Pro FPGA. La taille de l'architecture de l'Ordonnanceur synthétisé (nombre de CLB utilisés) dépend du nombre de tâches de l'application et du nombre d'arcs reliant les tâches du DFG.
Chaque IP détermine la date de début et de fin d'exécution de la tâche qu'il représente.
Pour représenter ces valeurs et les échanger entre les IPs il faut déterminer le nombre de bits nécessaires pour la présentation.
Par ailleurs, le choix de la taille des données influe beaucoup sur le résultat de synthèse. Les résultats suivants sont obtenus sur le Virtex II Pro vp7 :

Résultats de synthèse: Taille des données 16 bits		
Task IP		
Nombre de Slices	**Slices Flip Flops**	**LUT à 4 entrées**
82/ 4928 **1%**	**17/ 9856** **0%**	**147/ 9856** **1%**
ICAM DFG (20 tâches)		
Nombre de Slices	**Slices Flip flop**	**LUT à 4 entrées**
1889/ 4928 **38%**	**340/ 9856** **3%**	**3555/ 9856** **36%**
DFG update		
Nombre de Slices	**Slices Flip flop**	**LUT à 4 entrées**
442/ 4928 **8%**	**400/ 9856** **4%**	**440/ 9856** **4%**
Software manager		
Nombre de Slices	**Slices Flip flop**	**LUT à 4 entrées**
856/ 4928 **17%**	**16/9856** **0%**	**1617/9856** **16%**

Les résultats de synthèse indiquent une période minimale de fonctionnement de l'ordonnanceur matériel de 12,6 ns. L'ordonnancement de ICAM composée de 20 tâches s'effectue donc en un peu plus de 250 ns.

Avec des donnés sur 16 bits, nous pouvons présenter 2^{16} temps d'exécution : si le temps d'exécution maximum à mesurer est 40 ms, nous pouvons représenter des temps d'exécution avec une précision de 610 ns.

Une telle précision est plus que suffisante. Notre système n'a pas besoin d'être aussi précis, donc nous pouvons réduire la taille des données ce qui réduit le coût architectural de l'Ordonnanceur.

Avec des données codées sur 10 bits, les résultats de synthèse sont:

Résultats de synthèse: Taille des donnés 10 bits		
ICAM DFG (20 tâches)		
Nombre de Slices	**Slices Flip Flops**	**LUT à 4 entrées**
1677/ 4928 **34%**	**230/ 9856** **2%**	**3088/ 9856** **31%**

La période minimum de fonctionnement est alors de 11,4 ns.
Les données codées sur 10 bits permettent d'avoir une précision de 40 us si la valeur maximum du temps d'exécution est de 40 ms. Pour l'application ICAM, une telle précision est largement suffisante.

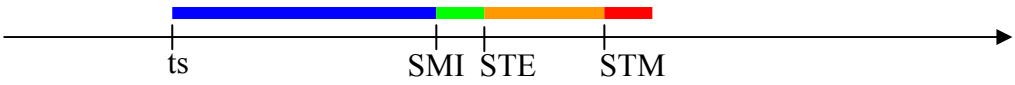
Sur le VIRTEX II Pro vp 7 l'ordonnanceur occupe 34% des ressources mais ce composant est relativement de petite taille. Avec un VIRTEX II Pro vp 100 et avec une taille des données égale à 10 bits l'ordonnanceur n'utilise que 3% de ressources (slices).

6.5 Résultats de partitionnement sur l'application ICAM

6.5.1 Effets de la variation des seuils

Comme mentionné dans le chapitre partitionnement (paragraphe 5.1.4), le seuil haut STM est égal à la contrainte temps réel diminué de l'overhead induit par les techniques mises en œuvre dans OPA. Il s'agit alors de définir trois différents seuils :

1. Seuil sur le Temps d'exécution Mesuré (STM)
2. Seuil sur le Temps d'exécution Estimé (STE)
3. Seuil de Migration Inverse (SMI)



ts : est le temps « start », début de l'exécution de la tâche.
Tout d'abord le temps d'exécution de la prochaine itération est estimé en appelant l'ordonnanceur à l'issue de chaque itération. Le système envisagera l'une des situations suivantes :

a) le temps estimé est : Testim, tel que : SMI< Testim< STE

Le système garde la configuration actuelle de l'architecture pour le traitement de la prochaine image.

b) Testim < SMI

Le système cherche à effectuer des migrations inverses possibles pour libérer du reconfigurable

c) Testim > STE

Le système cherche à effectuer des migrations directes pour diminuer le temps d'exécution total de l'application.

Une fois lancée la prochaine itération, elle ne doit dans aucun cas dépasser la contrainte STM. Et dans le cas où le temps d'exécution dépasse ce seuil, le système sera interrompu pour lancer l'image suivante. Une image est loupée.

Le seuil STM est fixé par la nature de l'application. Généralement, c'est la contrainte temps réel. Dans le cas du traitement d'une séquence vidéo ce seuil est égal à 40ms correspondant au traitement de 25 images par seconde.

La plateforme de simulation en SystemC de OPA est exécutée sur un PC qui exécute également les fonctions de l'application ICAM.
Nous utilisons les temps d'exécution de ces fonctions, mesurés sur le PC comme temps d'exécution logiciel pour évaluer le comportement de l'approche OPA.

Si nous fixons la contrainte STM à 40 ms, le PC à une performance suffisante et peut traiter toutes les images en gardant toutes les tâches sur le Software. Donc pour expérimenter notre algorithme de partitionnement OPA, nous baissons artificiellement le seuil STM à 26 ms. Notons que dans le cas d'un système embarqué les performances des processeurs sont souvent moins importantes. Aussi on peut considérer notre simulation comme une homothétie dans le temps d'un système réel.

Le seuil STE est obtenu par simulation. Nous avons expérimenté avec plusieurs seuils pour chercher celui qui donne un meilleur résultat du point de vue du nombre d'images non traitées avant la contrainte STM.

Nous avons considéré les conditions initiales suivantes :

- ✓ STM=26 ms.
- ✓ Taille maximum du FPGA= 300 CLB.
- ✓ Estimation par fonction d'interpolation
- ✓ Le paramètre de corrélation des fonctions à temps d'exécution variable est estimé par : PCestimé= PC mesuré* 1.5
- ✓ Nombre d'images dans la séquence test = 827 images.

Nous avons trouvé les résultats donnés par le tableau suivant :

STE	24	22	20	18
Images perdues	8	4	2	2

D'après les résultats de simulation le meilleur seuil STE est 20 ms. Ce seuil permet d'exploiter les ressources matérielles et de réduire le taux des images perdues. Le seuil 18 permet aussi d'avoir un minimum d'images perdues, mais le reconfigurable est saturé comme le montre la figure 6.21, ce qui réduit les capacités d'adaptation pour des variations importantes des paramètres de corrélation. On a donc intérêt à choisir un seuil STE le plus haut possible compte tenue de la qualité des résultats cherchée.

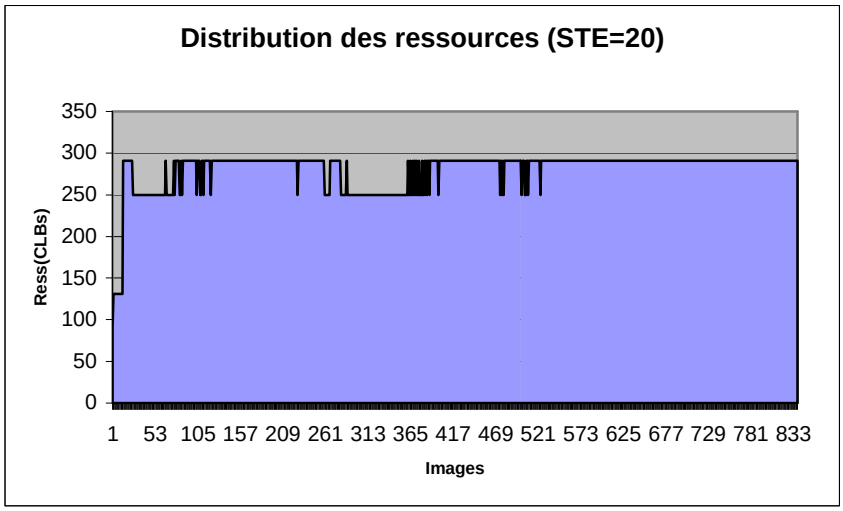


Figure 6.20 : Distribution des ressources matérielles avec seuil STE=20ms

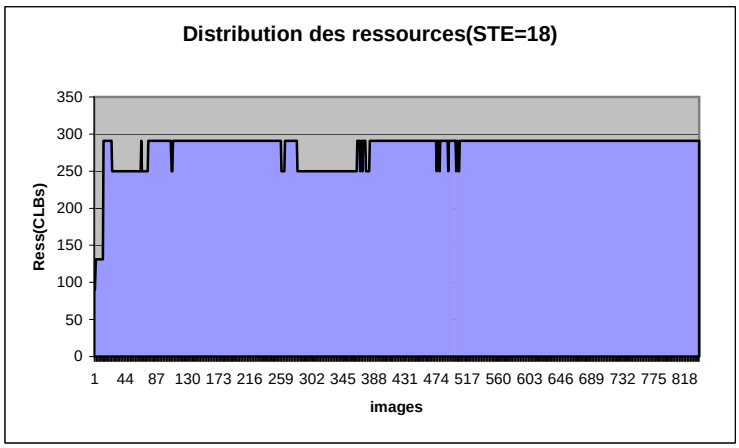


Figure 6.21 : Distribution des ressources matérielles avec seuil STE=18ms

6.5.2 Compromis taille du reconfigurable/ taux de respect

Sans qu'il soit trop pessimiste, l'estimateur prédit toujours un temps d'exécution légèrement supérieur à celui mesuré (voir courbe de différence).

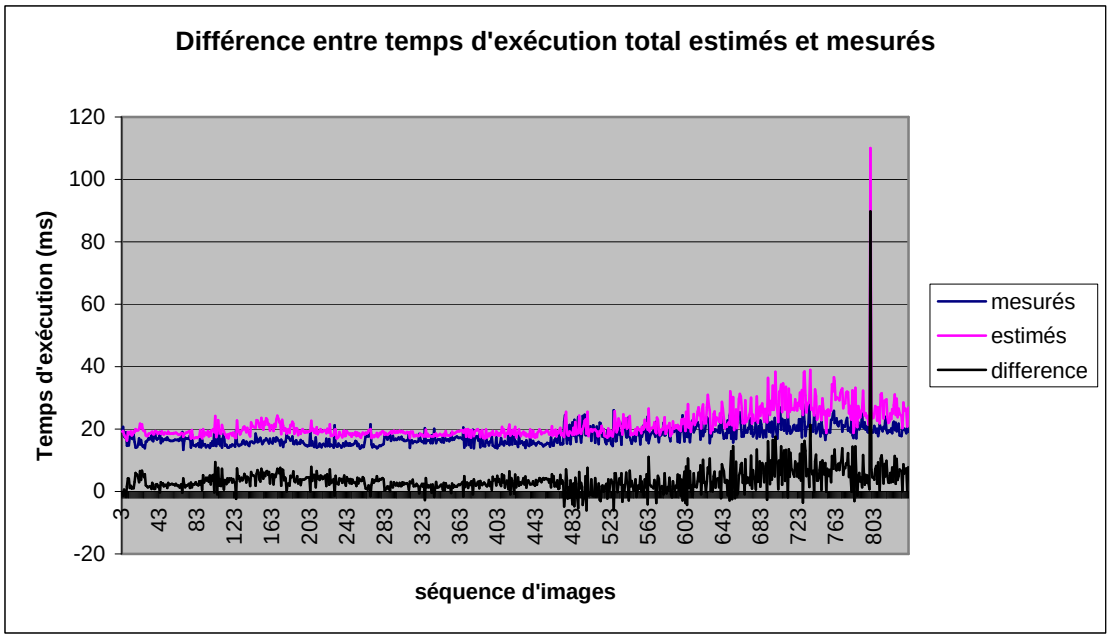


Figure 6.22 : Différence entre temps d'exécution total estimés et mesurés

Le système modifie la configuration dès que nécessaire (migrations directes) ou lorsque c'est possible (migrations inverses) dans le but d'obtenir la meilleure architecture en fonction de l'évolution de la charge de calcul.

Cependant, avec un nombre de ressources matérielles trop réduit, le système est incapable de trouver une configuration pour traiter en temps réel une image dont les paramètres de corrélation évoluent significativement par rapport à l'image précédente.
Ceci explique les plus nombreuses pertes d'images pour les petites tailles du reconfigurable (Taille < 300 CLB sur la courbe de la figure 6.23).

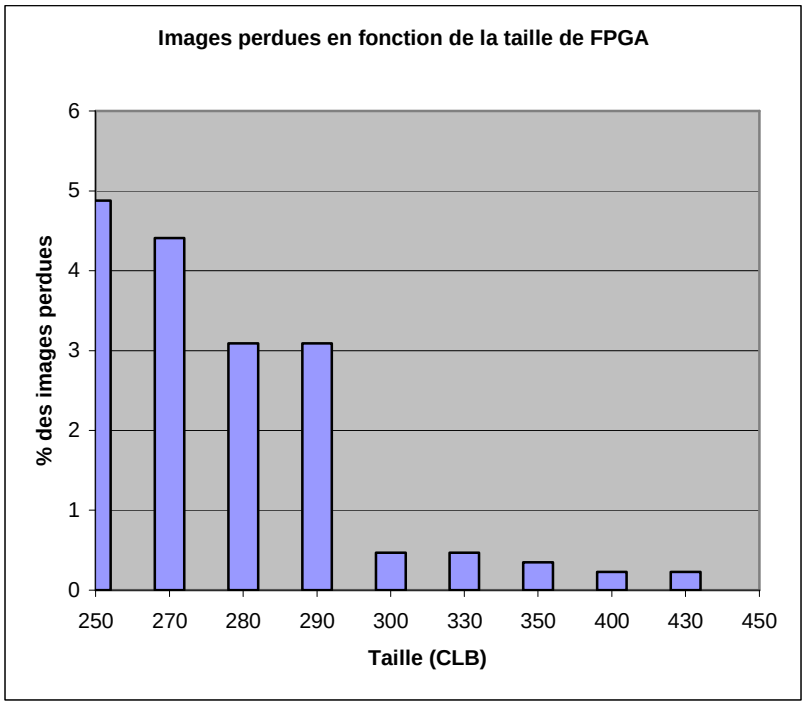


Figure 6.23 : Pourcentage des images perdues en fonction de la taille de FPGA

Le fait d'augmenter la taille du FPGA n'améliore pas toujours les résultats car pour effectuer des migrations directes il est nécessaire de disposer d'un nombre suffisant de ressources matérielles. Ceci explique les paliers qui se trouvent entre les valeurs 280 CLB et 290 CLB ou entre 300 CLB et 330 CLB.

Avec un FPGA de taille au moins égale à 450 CLB, le système arrive à trouver toutes les configurations qui permettent de traiter la totalité des images appartenant à la séquence de test considérée.

Avec une taille de 300 CLB il n'y a que 0,47% d'images perdues ce qui correspond à 4 images sur la séquence totale.

Nous pouvons tracer pour cette application la complexité des images définie par les paramètres de corrélation des tâches dont le temps d'exécution est variable. En fait nous avons uniquement deux paramètres de corrélation pour l'application ICAM, qui sont : le pourcentage des pixels blancs (ou bien le nombre des pixels blancs) et le nombre d'objets par image. Ces deux paramètres ne sont pas forcement eux-mêmes corrélés car nous pouvons avoir de nombreux pixels blancs qui construisent de gros objets et donc un nombre d'objets faible et inversement. Une multiplication de ces deux paramètres

donne une mesure de la complexité des images comme le montre la figure 6.24 (complexité).

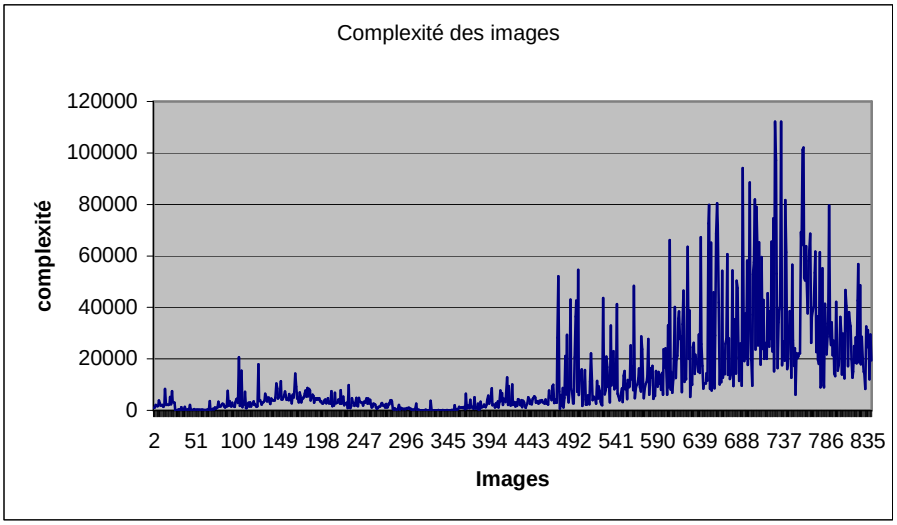


Figure 6.24 : Mesure de la complexité des images

D'après cette courbe de complexité, nous pouvons distinguer les séquences d'images à grande dynamique de complexité de celle à dynamique de complexité faible. Les images à dynamique complexe sont plutôt vers la fin de la séquence de test considérée. Pour tester l'estimateur de temps d'exécution qui donne le meilleur résultat sur les deux types d'images, nous avons effectué les expériences suivantes :

Pour différentes tailles de FPGA, nous testons pour chaque estimateur le nombre d'images perdues par l'approche OPA. Les résultats sont donnés dans le tableau ci-dessous.

Taille du reconfigurable	Kppv seul	Fonction d'approximation	**Kppv puis Fct**	Fct puis kppv
230 CLBs	46	41	**34**	50
250 CLBs	41	35	**35**	49
300 CLBs	1	3	**0**	5

Les nombres d'images perdues en considérant l'un ou l'autre des estimateurs sont assez similaires. Mais ceci ne renseigne pas sur le comportement des estimateurs par rapport à la dynamique de la complexité des images.
Nous avons expérimenté les deux solutions qui combinent les deux estimateurs: i) estimateur Kppv appliqué aux images à dynamique importante et estimateur par fonction d'approximation appliqué aux images à dynamique faible, ii) et l'approche inverse.

Il est clair que la meilleure combinaison des estimateurs est celle qui utilise l'estimateur Kppv pour les images à dynamique faible et lorsque les séquences d'images se complexifient nous employons l'estimateur par fonction d'approximation.

En utilisant les deux estimateurs proposés, nous arrivons avec uniquement 300 CLB à ne perdre aucune image.

Nous pouvons expliquer ce phénomène par le fait que l'estimateur Kppv se base sur les anciennes observations. Donc s'il y a une variation brusque de la complexité des images, cet estimateur ne peut pas suivre ces variations : dans la base de données il n'y a pas assez de voisins proches de la nouvelle valeur du paramètre de corrélation.

Cette étude est dépendante à la fois de la séquence de test considérée et de la nature de l'application (ICAM). Elle n'a pas vocation à être générique même si l'estimateur Kppv est relativement indépendant de l'application considérée. On montre par cette expérimentation que la qualité de l'estimation influence significativement les résultats et que des optimisations spécifiques sont toujours possibles en fonction de l'application cible.

6.6 Comparaison entre ILP et OPA

Dans cette partie nous souhaitons comparer l'approche OPA de partitionnement logiciel/matériel en ligne avec une formulation hors ligne du problème de partitionnement sous forme d'un programme linéaire en nombres entiers (ILP: Integer Linear Programming).

Contrairement à la méthode OPA, l'approche utilisant la formulation ILP est une approche de partitionnement hors-ligne basée sur les temps d'exécution pire cas (WCET).

Les temps d'exécution pire cas des fonctions de l'application ICAM dont le contrôle dépend des données ne sont pas aisés à déterminer. Il faudrait par exemple donner une borne maximum sur le nombre d'objets en mouvement. Afin d'obtenir des résultats comparables nous procédons comme suit.

Le principe consiste à prendre une séquence test d'images, appliquer l'algorithme ICAM avec la méthode de partitionnement OPA en simulation sous SystemC et relever les mesures des temps d'exécution maximum des tâches. Ces temps sont ensuite utilisés pour appliquer le partitionnement suivant l'approche ILP sur le DFG de l'application.
Nous fixons la limite supérieure du temps d'exécution global de l'application à Texe = 26 ms (c'est la contrainte temps réel considérée dans la simulation OPA).

On peut remarquer que l'application ICAM englobe dix tâches séquentielles, chaque tâche reçoit en entrée l'image résultat de la tâche précédente et fournit en sortie une image résultat qui sera l'entrée de la tâche suivante (voir figure 6.3). Donc le temps d'exécution total de l'application n'est autre que la somme des temps d'exécution de toutes les tâches. On utilise cette remarque pour simplifier la modélisation du problème d'optimisation.

6.6.1 Formulation du problème dans l'approche ILP

Le modèle que nous avons considéré est simple. Nous considérons les mêmes hypothèses que celles utilisées dans la simulation :

- les temps de communication entre les tâches sont nuls.
- Le DFG de l'application est séquentiel.
- Chaque tâche peut avoir 4 implémentations possibles, donc le temps d'exécution d'une tâche est :

$$T_i = a_{i,1} * T_i^s + \sum_{j=2}^{4} a_{i,j} * T_{i,j}^H \qquad (6.12)$$

Avec comme contrainte : $0 <= a_{i,j} <= 1$; $a_{i,j} \in N$ et $\sum_{j=1}^{4} a_{i,j} = 1$

Le temps d'exécution global de l'application est la somme des temps d'exécution des tâches.

$Texe = \sum_{i=0}^{i=n} T_i$ avec n le nombre de tâches du DFG.

- Le nombre de ressources d'une tâche est le suivant :

$$R_i = \sum_{j=2}^{4} a_{i,j} * R_{i,j}^H \qquad (6.13)$$

Le nombre total de ressources total utilisées dans l'application est la somme des ressources utilisées par chaque tâche.

La fonction objective consiste à minimiser le nombre total de ressources avec la contrainte que:

- Le temps d'exécution global de l'application est inférieur à 26ms.

Dans l'application ICAM, trois tâches ont leur temps d'exécution variable en fonction de la nature des données traitées: Enveloppe, étiquetage et Ouverture.

- Pour la tâche Enveloppe :

Le nombre maximum d'objets trouvés dans la séquence d'images est 24 (image n°650).
Le temps d'exécution maximum de la fonction enveloppe est donc 11,2ms.

- Pour la tâche Ouverture :

Le nombre maximum de pixels blancs calculés dans une image est 21077 pixels pour l'image n° 798

Donc pour cette tâche on considère un temps maximum de 64,6ms.

- Pour la tâche Etiquetage :

Le nombre maximum de pixels blancs dans une image à l'entrée de l'étiquetage est égal à 34018 pixels pour l'image 798. Son temps d'exécution maximum est 37,5ms.

- Pour chacune des tâches :

Tâche	Ressource 1	Ressource 2	Ressource 3	tsw	thw1	thw2	thw3
1	200	403	610	16,128	3,2256	1,6128	0,96768
2	41	100	153	5,312	1,06	0,531	0,318
3	52	100	155	1,721	0,3442	0,1721	0,10326
4	13	30	75	13,08	2,616	1,308	0,784
5	160	450	850	64,64	12,928	6,464	3,878
6	31	60	95	30,62	6,124	3,062	1,8372
7	120	290	560	37,51	7,502	3,751	2,25
8	130	300	690	11,238	2,247	1,123	0,674
9	43	84	135	0,2	0,04	0,02	0,012
10	60	121	184	1,9	0,38	0,19	0,114

Interprétation des résultats :

Nous avons mis les deux algorithmes ILP et OPA dans les mêmes conditions d'expérimentations :

- ✓ Même application : ICAM
- ✓ Mêmes seuils haut et bas du temps d'exécution
- ✓ Mêmes courbes d'implémentations

Les résultats sont récapitulés dans le tableau suivant :

Approche	Ressources utilisées	Nombre d'images perdues	Travail Hors ligne	Ressources ajoutées	Temps réel strict	flexibilité
ILP	1126	0	Oui	Non	Oui	Non
OPA	290	0	limité	Oui	Non	Oui

Nous remarquons que pour le même taux de respect de la contrainte temps réel (0 images perdues), l'approche ILP utilise beaucoup plus de ressources que notre approche OPA : cette dernière a besoin uniquement du quart des ressources pour fournir le même résultat qu'une approche de partitionnement basée sur ILP.
Le temps d'exécution déterminé par la méthode basée sur l'approche ILP est de 25,944 ms. Les temps d'exécution retournés par notre approche OPA oscillent entre 10,895 et 25,7.

A partir de la courbe relative au temps d'exécution (figure 6.24), il apparaît que l'algorithme basé sur l'approche ILP est souvent pessimiste sur l'estimation du temps d'exécution total de l'application en utilisant les valeurs des WCET. Ceci explique le surdimensionnement de l'architecture par l'utilisation importante des ressources matérielles.
En fait, et d'après les résultats de l'approche OPA, le temps d'exécution de ICAM sur la séquence test considérée ne dépasse que rarement 20 ms : sur 839 images, il y a 800 images dont le temps d'exécution est inférieur à 21ms.

De plus, l'approche basée sur ILP exige un travail hors ligne pour trouver le partitionnement adéquat, à l'inverse de l'algorithme OPA qui ne nécessite qu'un minimum de préparation hors ligne.
Ce travail réduit rend l'approche assez flexible ce qui permet de l'envisager pour de nombreuses applications sans intervention significative du concepteur.

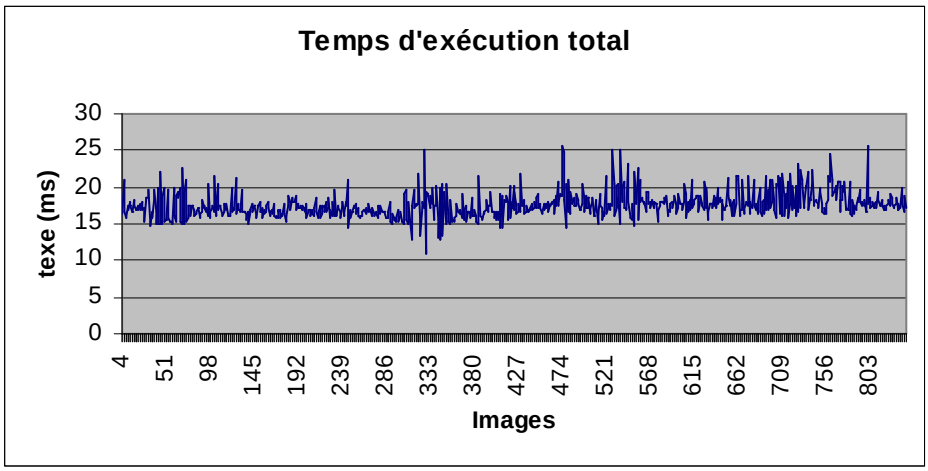


Figure 6.25 : Le temps d'exécution total de l'application avec l'approche OPA

En contre partie, l'approche OPA ajoute un overhead temporel et architectural dû à l'implémentation en ligne des algorithmes d'estimation, d'ordonnancement et de partitionnement. Ce qui n'est pas le cas pour l'approche ILP qui s'exécute hors ligne.

Conclusion

Dans ce chapitre nous avons présenté les résultats de notre méthodologie de partitionnement logiciel/matériel dynamique sur l'application test ICAM:

> L'application ICAM est une application de traitement d'images orientée flots de données qui présente trois tâches à temps d'exécution dépendant des valeurs des données traitées: i) la fonction "étiquetage des objets" dont le temps d'exécution dépend du nombre de pixels blancs dans l'image ii) la fonction "enveloppe englobante" dont le temps d'exécution dépend du nombre d'objets dans l'image et iii) la fonction "ouverture par reconstruction" dont le temps d'exécution dépend du nombre de pixels blancs dans l'image.

Les résultats de simulation sur une plateforme modélisée en SystemC ont permis de valider l'estimateur de paramètre de corrélation, l'estimateur du temps d'exécution, l'ordonnanceur, et le partitionnement. Enfin, une comparaison de l'approche OPA avec l'approche ILP basée sur des temps d'exécutions identifiés comme WCET, a permis de valider la globalité de la méthodologie de partitionnement dynamique.

La comparaison des résultats des heuristiques d'estimation du paramètre de corrélation a permis de choisir l'estimateur qui présente le minimum d'erreur en tenant compte de la simplicité d'implémentation en ligne.

Les études expérimentales effectuées sur les deux méthodes d'estimation du temps d'exécution d'une tâche à savoir la méthode Kppv et l'approximation par une équation, ont permis de définir les paramètres du premier estimateur et le contexte d'utilisation du deuxième. Nous avons trouvé par la suite un critère de complexité des images qui permet d'employer le premier estimateur ou le deuxième.

La synthèse de l'ordonnanceur matériel sur la cible VIRTEX II Pro vp 100 et en considérant une échelle de temps exprimée sur 10 bits occupe uniquement 3% des ressources (slices). Ceci est très encourageant étant donnée l'efficacité de cette implémentation matérielle.

Les résultats de simulation du partitionnement/ordonnancement sur plusieurs séquences d'images tests ont permis d'analyser finement le choix des seuils pour l'application ICAM.

Conclusion générale

OBJECTIFS

Les objectifs que nous nous sommes fixés dans cette étude sont principalement les suivants:

- Apporter des solutions d'aide au développement d'applications basées sur une plateforme reconfigurable auto-adaptative constituée d'un processeur connecté à une unité reconfigurable dynamiquement.

- Proposer un flot global d'implémentation intégrant la méthode et les outils permettant d'effectuer dynamiquement le partitionnement logiciel/matériel en ligne.

BILAN DES TRAVAUX

Nous avons développé dans cette étude un flot générique d'une méthodologie de partitionnement logiciel/matériel en ligne. Le principe de la méthode est rappelé sur la figure 2.

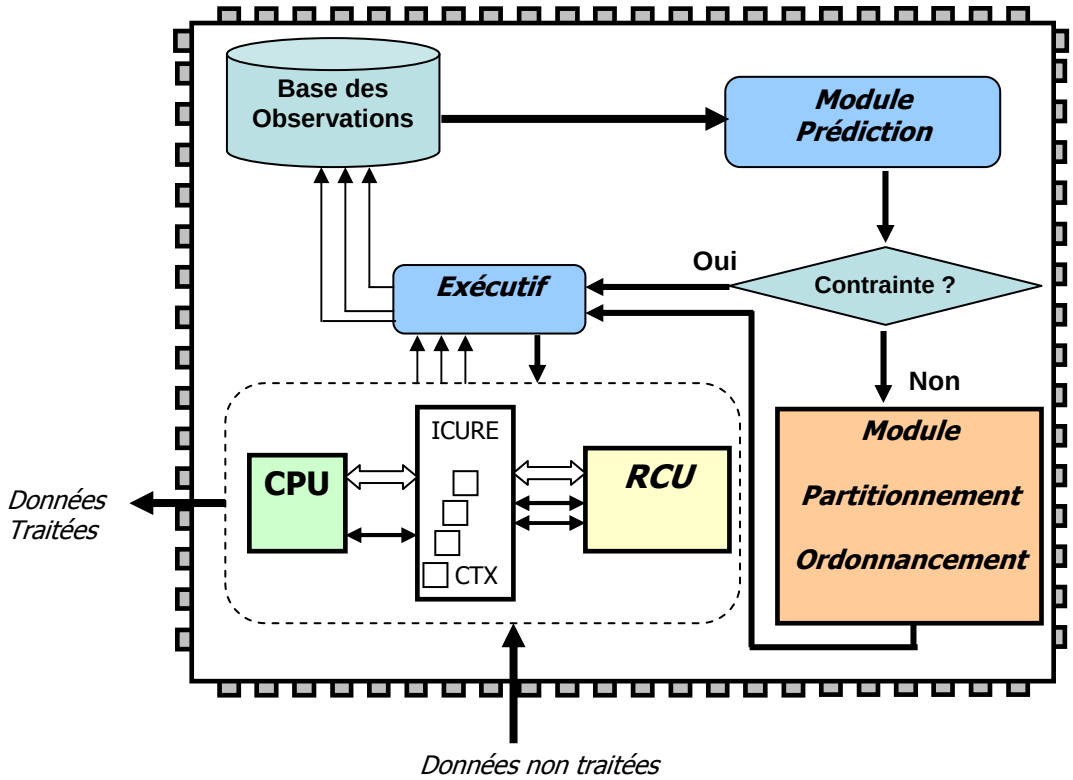


Figure 2 : Méthodologie de partitionnement dynamique en ligne

Notre approche de partitionnement vise des applications à distribution variable de la charge de calculs. L'idée principale proposée consiste à estimer les temps d'exécution des tâches pour les exécutions suivantes afin de prédire un éventuel non respect de la contrainte temps réel et d'éviter un dépassement de cette échéance par une réallocation des tâches qui deviennent critiques sur le reconfigurable. Ainsi l'architecture s'auto-adapte aux besoins de traitement.

Le flot global comporte trois modules principaux:

- Le module d'estimation des temps d'exécution des tâches avec deux techniques différentes: la première est basée sur le principe des K voisins plus proches et peut être appliquée à des séquences de données complexes; la deuxième technique est basée sur l'approximation des temps d'exécution par des équations déterminées hors ligne.

- Le module de partitionnement se base sur le principe de migrations directes et inverses des tâches. Les migrations directes ont pour objectif d'accélérer le traitement de l'application sur l'architecture. Les migrations inverses ont pour objectif de préparer l'architecture à de futures migrations directes en libérant les ressources du reconfigurable utilisées par des tâches non critiques.

- L'évaluation d'un résultat de partitionnement est basée sur une heuristique d'ordonnancement logiciel/matériel. Cette heuristique d'ordonnancement affecte des priorités aux tâches de l'application suivant trois critères. Le premier critère est le temps ASAP, ensuite la criticité et enfin les temps d'exécution des tâches.

Nous avons validé en simulation sur une plateforme modélisée en SystemC ce flot sur l'application test *ICAM* : une application, orientée flots de données, de détection de mouvements dans une séquence d'images prise par une caméra fixe.
Les résultats de partitionnement de cette application sur plusieurs séquences vidéo montrent que notre méthodologie est capable de traiter des systèmes avec un nombre de ressources matérielles réduit qui s'adaptent aux besoins de traitement.

Le flot original dont les principes d'auto-adaptation ont été montrés dans nos travaux peut être amélioré de plusieurs façons.

PERSPECTIVES

Une des extensions à considérer dans notre flot serait d'adopter une fonction coûts multi-objectifs. Ainsi il pourrait être intéressant d'intégrer

l'objectif de la minimisation de la consommation d'énergie ou des pics de puissance pour cibler des équipements sans fil.
Dans ce cas, il serait nécessaire d'évaluer le bilan énergétique lié au gain en ressources reconfigurables induit par notre approche par rapport au surcoût introduit par la reconfiguration dynamique des migrations des tâches.

Une autre perspective serait d'ajouter à notre modèle les temps de communication et les temps de reconfiguration pour pouvoir reproduire fidèlement les comportements de l'architecture dans les estimateurs et ordonnanceurs.

Il serait également intéressant de tester notre approche de partitionnement logiciel/matériel en ligne sur d'autres applications différentes (dans le domaine de la radio logicielle ou de la téléphonie mobile…) pour évaluer la généricité de la méthode et/ou d'en déduire des extensions pour mieux appréhender ces domaines d'applications.

Références bibliographiques

[1] R. ERNST, J. HENKEL, T. BENNER, "Hardware-Software Cosynthesis for Microcontrollers", IEEE Journal Design and Test of Computers,Vol. 10, N° 4, pp. 64-75, December 1993.

[2] R. GUPTA, G. DE MICHELI, "Hardware-Software Cosynthesis for Digital Systems", IEEE Journal Design and Test of Computers, pp 29-41, september, 1993.

[3] D. Gajski, F. Vahid, "Specification and Design of Embedded System", IEEE Design & Test of Computers, pp. 53-67, Spring 1995.

[4] Jorg Henkel, Rolf Ernst, "The Interplay of Run-Time Estimation and Granularity in HW/SW Partitioning", 4th Int. Workshop on Hardware/Software Co-design (CODES/CASHE'96) March 18-20,1996 Pittsburgh, Pennsylvania.

[5] Jörg Henkel, Rolf Ernst, "A Hardware/Software Partitioner Using a Dynamically Determined Granularity", DAC 1997, pp 691-696.

[6] R.K. Gupta and G.D. Micheli, System-level Synthesis using Re-programmable Components, IEEE/ACM Proc. of EDAC'92, IEEE Comp. Soc. Press, pp. 2–7, 1992.

[7] E.Barros, W.Rosentiel, and X. Xiong, "A Method for Partitioning UNITY Language in Hardware and Software", Proc. European Design Automation Conference (EuroDAC), IEEE CS Press, Grenoble, France, September 1994, pp. 580-585.

[8] A. Jantsch, P. Ellervee, J. Oeberg et. al., Hardware/Software Partitioning and Minimizing Memory Interface Traffic, IEEE/ACM Proc. of The European Conference on Design Automation (EuroDAC) 1994, pp. 220–225, 1994.

[9] P. M. Athanas, H. F. Silverman, "Processor Reconfiguration through Instruction-Set Metamorphosis," Computer, March 1993, pp. 11-18.

[10] Z. Peng, and K. Kuchcinki, "An Algorithm for Partitioning of Application Specific Systems", Proc. European Design & Test Conference (EDAC-ETC-EuroASIC), IEEE CS Press, pp. 316-321, February 1993.

[11] J. Madsen, P. V. Knudsen, LYCOS Tutorial, Handouts from Eurochip course on Hardware/Software Codesign, Denmark, 14.–18. Aug. 1995.

[12] R. Niemann, P. Marwedel, Hardware/Software Partitioning using Integer Programming, IEEE/ACM Proc. of EDAC'96, pp.473–479, 1996.

[13] I. Karkovski, R. H. J. M. Otten,An Automatic Hardware–Software Partitioner Based on the Possibilistic Programming, IEEE/ACM Proc. of EDAC'96, pp.467–472, 1996.

[14] F. Vahid, D.D. Gajski, J. Gong, A Binary–Constraint Search Algorithm for Minimizing Hardware during Hardware/Software Partitioning, IEEE/ACM Proc. of The European Conference on Design Automation (EuroDAC) 1994, pp. 214–219, 1994.

[15] F. Vahid,D. D. Gajski, Clustering for improved system–level functional partitioning, IEEE/ACM Proc. of 8th. International Symposium on System Synthesis, pp. 28–33, 1995.

[16] T. Y. Yen, W. Wolf, Multiple–Process Behavioural Synthesis for Mixed Hardware–Software Systems, IEEE/ACM Proc. of 8th. International Symposium on System Synthesis, pp. 4–9, 1995.

[17] J. K. Adams, D. E. Thomas Multiple–Process Behavioural Synthesis for Mixed Hardware–Software Systems, IEEE/ACM proc. of 8th. International Symposium on System Synthesis, pp. 10–15, 1995.

[18] Kalavade, E. A. Lee, "A Global Criticality/Local Phase driven Algorithm for the Constrained Hardware/Software Partitioning Problem", Proc. of Codes/CASHE'94, Third Intl. Workshop on Hardware/Software Codesign, Sept. 22-24, 1994, pp. 42-48

[19] F. VAHID, Modifying Min-Cut for hardware and software functional partitioning, Workshop on Hardware/Software Codesign, Braunschweig, Allemagne, 1997.

[20] J. TEICH, T. BLICKLE, L. THIELE, An evolutionary approach to system level synthesis, Codes/ CASHE, Braunschweig, Allemagne, mars, 1997.

[21] I. KARKOWSKI, H. CORPOORAL. Design space exploration algorithm for heterogeneous multi-processor embedded system design. DAC 98, San Francisco, CA, juin, 1998.

[22] Y. LI, T. CALLAHAN, E. DARNELL, R. HARR, U. KURKURE, J. STOCKWOOD, Hardware-software co-design of embedded reconfigurable architectures, Design Automation Conference, Los Angeles, juin 2000.

[23] R. MAESTRE, J. KURAHI, N. BAGHERZADEH, H. SINGH, R. HERMIDA, M. FERNANDEZ, Kernel Scheduling in reconfigurable scheduling, DATE, Munich, march, 1999.

[24] M.KAUL, R. VEMURI, S. GOVINDARAJAN, I. OUAISS, An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis of DSP applications. Design Automation Conference, New Orleans, LA, 1999.

[25] A. KALAVADE, E. LEE, The extended partitioning problem: hardware/software mapping and implementation bin selection, 6th International Workshop on Rapid Systems Prototyping, 1995.

[26] T. YEN YEN, W. WOLF, Sensitivity-driven cosynthesis of distributed embedded systems, Internation Symposium on System Synthesis, 1995.

[27] T. GRANDPIERRE, C. LAVARENNE, Y. SOREL Optimized rapid prototyping for real time embedded heterogeneous multiprocessors. CODES'99 7th International Workshop on Hardware/ Software Co-Design, Rome, mai 1999.

[28] P.BJORN-JORGENSEN, J. MADSEN, Critical path driven cosynthesis for heterogeneous target architecures,.

[29] B. DAVE, G. LAKSHMINARAYANA, N. LHA, COSYN: hardware/software co-synthesis of embedded systems, Design Automation Conference, Anaheim, 1997.

[30] H. OH, S. HA, A hardware/software co-synthesis technique based on heterogeneous multiprocessor scheduling, Codes'99, Rome, 1999.

[31] J.K. Adams, H. Schmitt, and D.E.Thomas, "A Model and Methodology for Hardware-Software Codesign", Handouts of Int'l Wshp on Hardware-Software Co-design, Cambridge, Massachusetts, IEEE CS Press, October 1993.

[32] D.E. Thomas, J.K Adams, and H. Schmit, "A Model and Methodology for Hardware-Software Codesign", IEEE Design & Test of Computers, Vol. 10, N° 3, pp. 6-15, September 1993.

[33] E.Barros, and W .Rosentiel, "A Method for Hardware Software Partitioning" , IEEE Comp. Euro,1992.

[34] M.B. Srivastava, and R.B. Brodersen "Rapid-prototyping of Hardware and Software in a Unified Framework", Proc. Int'l Conf. On Computer-Aided Design (ICCAD), IEEE CS Press, pp. 152-155, 1991.

[35] N.L. Rethman, and P.A. Wilsey, "RAPID : A Tool for Hardware/Software Tradeoff Analysis", Proc. IFIP Conf. Hardware Description Languages (CHDL), Publ. Elsevier Science, Ottawa, Canada, April 1993.

[36] S. Prakash, and A. Parker, Synthesis of application-specific multiprocessor architectures, in Proceedings of the Design Automation Conference (DAC'91), pp. 8-13, 1991.

[37] B. DAVE, G. LAKSHMINARAYANA, N. LHA, COSYN: hardware/software co-synthesis of embedded systems, Design Automation Conference, Anaheim, 1997.

[38] M. Auguin, L. Capella, F. Cuesta, and E. Gresset, CODEF: a system level exploration tool, ICASSP, Salt Lake City, 7-11 mai 2001.

[39] Gilberto Fernandes MARCHIORO, "Découpage Transformationnel pour la conception de systèmes Mixtes Logiciel/Matériel", Thèse de doctorat, Institut National polytechnique de Grenoble, Novembre 1998, 189p.

[40] A. Jerraya and K. O'Brien, "SOLAR: An Intermediate Format for System-Level Modeling and Synthesis," in Codesign: Computer-Aided Software/Hardware Engineering, J. Rozenblit and K. Buchenrieder, Eds. Piscataway, NJ: IEEE Press, 1995, pp. 145-175.

[41] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vicentelli, "A Formal Specification Model for Hardware/Software Codesign," Technical Report UCB/ERL M93/48, Dept. EECS, University of California, Berkeley, June 1993.

[42] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," in Science of Computer Programming, vol. 8, pp. 231-274, June 1987.

[43] C. G. Cassandras, Discrete Event Systems: Modeling and Performance Analysis. Boston, MA: Aksen Associates, 1993.

[44] E. A. Lee, “Modeling Concurrent Real-Time Processes using Discrete Events,” Technical Report UCB/ERL M98/7, Dept. EECS, University of California, Berkeley, March 1998.

[45] J. Peterson, Petri Net Theory and the Modeling of Systems. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[46] G. Dittrich, “Modeling of Complex Systems Using Hierarchical Petri Nets,” in Codesign: Computer-Aided Software/Hardware Engineering, J. Rozenblit and K. Buchenrieder, Eds. Piscataway, NJ: IEEE Press, 1995, pp. 128-144.

[47] E. Stoy, “A Petri Net Based Unified Representation for Hardware/Software Co-Design,” Licentiate Thesis, Dept. of Computer and Information Science, Linköping University, Linköping, 1995.

[48] E. A. Lee and T. Parks, “Dataflow Process Networks,” in Proc. IEEE, vol. 83, pp. 773-799, May 1995.

[49] T. Parks, J. L. Pino, and E. A. Lee, “A Comparison of Synchronous and Cyclo-Static Dataflow,” in Proc. 29th Asilomar Conference on Signals, Systems and Computers, 1995, pp. 204-210.

[50] C. A. R. Hoare, Communicating Sequential Processes. Englewood Cliffs, NJ: Prentice-Hall, 1985.

[51] F. Boussinot and R. de Simone, “The ESTEREL Language,” in Proc. IEEE, vol. 79, pp. 1293-1304, Sept. 1991.

[52] M. AUGUIN, L. BIANCO, L. CAPELLA, E. GRESSET, "Partitioning conditional data flow graphs for embedded system design", International Conference on Application Specific Systems, Architectures and Processors, ASAP, Boston, p.337, July 10-12, 2000.

[53] R.K. GUPTA, C.N. Coelho Jr., and G. DeMicheli, "Program Implementation Schemes for Hardware-Software Systems", Wshp Handouts of Int’l Wshp on Hardware-Software Co-Design, IEEE CS Press, October 1992.

[54] R.K. GUPTA, C.N. Coelho Jr., and G. DeMicheli, "Program Implementation Schemes for Hardware-Software Systems", IEEE Computer, Vol. 27, N° 1, pp. 48-55, January 1994.

[55] D. Gajski, F. Vahid, and S. Narayan, "A system-Design Methodology : Executable-Specification Refinement", Proc. European Design & Test Conference (EDAC-ETC-EuroASIC), Paris, France, IEEE CS Press, pp. 458-463, February 1994.

[56] J. Gong, D. Gajski, and S.Narayan, "Software Estimation from Executable Specifications", Proc. European Design Automation Conf. (EuroDAC), IEEE CS Press, Grenoble, France, pp. 47-57, September 1994.

[57] A. KALAVADE, E.A. LEE, "A Hardware-Software Codesign Methodology for DSP Applications", IEEE Design & Test of Computers, Vol. 10, N° 3, pp. 16-28, September 1993.

[58] A. Kalavade, and E.A. Lee, "Manifestations of Heterogeneity in Hardware/Software Codesign", Proc. 31 st Design Automation Coference (DAC), IEEE CS Press, pp. 437-438, June 1994.

[59] M.B. Srivastava, Rapid-prototyping of Hardware and Software in a Unified Framework, Ph.D. thesis, University of Calif. Berkeley, June 1992.

[60] M.B. Srivastava, and R.B. Brodersen, "Using VHDL for High-Level, Mixed-Mode Simulation", IEEE Design & Test of Computers, pp. 31-40, September 1993.

[61] M. Chiodo, P. Giusto, A. Jurecska, L. Lavagno, H. Hsieh, A. Sangiovanni-Vincentelli, "A Formal Specification Model for Hardware/Software Codesign", Wshp Handouts of Int'l Wshp on Hardware-Software Co-Design, Cambridg, Massachusetts, IEEE CS Press, p. 53, October 1993.

[62] P. Hilfinger, "A High-Level Language and Silicon Compiler for Digital Signal Processing," Proc. IEEE CICC, Portlant, pp. 213-216, May 1985.

[63] P. Paulin, C. Liem, T. May, and S. Starwala, "DSP Design Tool Requirements for Embedded Systems: A Telecommunication Industrial Perspective," in journal of VLSI Signal Processing (special issue on synthesis for real-time DSP), Kluwer Academic Publishers, 1994.

[64] K. Buchenrieder, and C. Veith, "CODES: A practical Concurrent design Environment," Int'l Wshp on Hardware/Software Co-Design, Estes Park, Colorado, IEEE CS Press, October 1992.

[65] K. Buchenrieder, A. Sedlmeier, and C. Veith, "HW/SW Co-Design with PRAMs using CODES," Proc. IFIP Conf. Hardware Description Languages (CHDL), Publ. Elsevier Science, Ottawa, Canada, April 1993.

[66] K. Buchenrieder, "A prototyping Environment for Control-Oriented HW/SW Systems using State-Charts, Activity-Charts and FPGA's," Proc. Euro-DAC with Euro-VHDL, Grenoble, France, IEEE CS Press, pp. 60-65, September 1994.

[67] K. A. Olukotun, R. Helaihel, J. Levitt, R. Ramirez, "A Software-Hardware Co-Synthesis Approach to Digital System Simulation," IEEE Micro, August 1994, pp. 48-58.

[68] M. Edwards, J. Forrest, "A Development Environment for the Cosynthesis of Embedded Software/Hardware Systems," in Proc. European Design Automation Conference EDAC, 1994, pp. 469-473.

[69] T. Ben Ismail, A.A. Jerraya, "Synthesis Steps and Design Models for Codesign," Computer, February 1995, pp. 44-52.

[70] P. H. Chou, R. B. Ortega, G. Boriello, "The Chinook Hardware/Software Co-Synthesis System," in Proc. International Symposium on System Synthesis, 1995, pp. 22-27.

[71] S. Kirkpatrick, C. D. Gelatt, Jr., andM. P. Vecchi. Optimization by simulated annealing. Science, 220(4598):671–680, May 1983.

[72] D. E. Goldberg. Genetic Algorithms in Search, Optimization,and Machine Learning. Addison-Wesley, 1989.

[73] K. BEN CHEHIDA "Méthodologie de Partitionnement Logiciel/Matériel pour Plateformes Reconfigurables Dynamiquement" thèse de doctorat, Laboratoire I3S, Université de Nice Sophia Antipolis France, Novembre 2004.

[74] F. Glover, E. Taillard, and D. de Werra. A user' s guide to tabu search. Annals of Operations Research, 41:3–28, 1993.

[75] R. Gupta and G. De Micheli, ``Vulcan: a System for High-Level Partitioning of Synchronous Digital Circuits," CSL Report, CSL-TR-91-471, 1991.

[76] Peter V. Knudsen and Jan Madsen. Pace: A dynamic programming algorithm for hardware/software partitioning. In Proceedings of 4th International Workshop on Hardware/Software Codesign, Codes/CASHE'96, pages 85 - 92, March 1996.

[77] Aviral Shrivastava, Mohit Kumar, Sanjiv Kapoor, Shashi Kumar, M. Balakrishnan, "Optimal Hardware/Software Partitioning for Concurrent Specification Using Dynamic Programming" 13th International Conference on VLSI Design , January 04 - 07, 2000, Calcutta, India.

[78] 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.

[79] 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.

[80] Ralf Niemann, Peter Marwedel, Hardware/Software Partitioning using Integer Programming, In Proccedings of the European Design and Test Conference (ED \& TC), 1996

[81] F.GHAFFARI, M.BENJEMAA, M.AUGUIN. Algorithms for the Partitioning of Applications containing variable duration tasks on reconfigurable architectures. IEEE Int. Conf. AICCSA 2003 Tunis, TUNISIA 14 – 18 July 2003.

[82] Yu-Kwong Kwok, Anthony A. Maciejewski, Howard Jay Siegel, Arif Ghafoor, Ishfaq Ahmad: Evaluation of a Semi-Static Approach to Mapping Dynamic Iterative Tasks onto Heterogeneous Computing Systems. ISPAN 1999: 204-209

[83] F.GHAFFARI, "Etude du partitionnement logiciel/materiel d'applications à distribution variable de charge de calcul", mémoire de DEA, ENIS Tunisie et Laboratoire I3S, France, Juin 2002.

[84] Yu-Kwong Kwok, Anthony A. Maciejewski, Howard Jay Siegel, Ishfaq Ahmad, Arif Ghafoor: A semi-static approach to mapping dynamic iterative tasks onto heterogeneous computing systems, January 2006, Journal of Parallel and Distributed Computing, Volume 66 Issue 1.

[85] F.GHAFFARI, M.AUGUIN, M.BENJEMAA. Etude du partitionnement logiciel/matériel d'applications à distribution variable de charge de calcul. Renpar'14 /ASF/SYMPA pp. 334-338 Hammamet, TUNISIE 10 – 13 avril 2002.

[86] Q. Wang, K.H. Cheng, List scheduling of parallel tasks, Inform. Process. Lett. 37 (5) (March 1991) 291–297.

[87] D.M. Nicol, J.H. Saltz, Dynamic remapping of parallel computations with varying resource demands, IEEE Trans. Comput. 37 (9) (September 1988) 1073–1087.

[88] Greg Stitt, Frank Vahid: Energy Advantages of Microprocessor Platforms with On-Chip Configurable Logic. IEEE Design & Test of Computers 19(6): 36-43 (2002)

[89] Greg Stitt, Frank Vahid: A Decompilation Approach to Partitioning Software for Microprocessor/FPGA Platforms. DATE 2005: 396-397.

[90] Roman L. Lysecky, Frank Vahid: A Configurable Logic Architecture for Dynamic Hardware/Software Partitioning. DATE 2004: 480-485.

[91] Thilo Streichert, Christian Haubelt, Jürgen Teich: Online hardware/software partitioning in networked embedded systems. ASP-DAC 2005: 982-985.

[92] Peter Waldeck, Neil W. Bergmann: Dynamic Hardware-Software Partitioning on Reconfigurable System-on-Chip. IWSOC 2003: 102-105.

[93] J.P. Diguet, Exploration de l'espace de conception de SOC, de l'asservissement à la cooperation, HDR, Université de Bretagne Sud, Septembre 2005.

[94] J.P. David, Architecture synchronisée par les données pour système reconfigurable, thèse de doctorat, Université Catholique de Louvain, Juin 2002.

[95] Yann THOMA, Tissu numérique cellulaire à routage et configuration dynamique, thèse de doctorat, Ecole Polytechnique Fédérale de Lausanne, 2005.

[96] S. J.E. Wilton. Architectures and Algorithms for Field Programmable Gate Arrays with Embedded Memory. Ph.D Thesis, University of Toronto, Canada, 1997.

[97] S. Rubini, D. Lavenier, Les architectures Recoonfigurables, Claculateurs Parallèles, 9(1), 1997.

[98] Yan Solihin, Kirk W. Cameron, Yong Luo, Dominique Lavenier, and Maya Gokhale, Mutable Functional Units and Their Applications on Microprocessors. International Conference on Computer Design 2001 (ICCD), Austin, Texas, Sep 23-26, 2001.

[99] Xilinx. MicroBlaze Hardware Reference Guide, Octobre 2001.

[100] Altera. Nios Soft Core Embedded Processor, June 2000.

[101] 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.

[102] D. Demigny, N. Boudouani, N. Abel, L. Kessal, La rémanence des architectures reconfigurables : un critère significatif de classification des architectures, Proceedings of the JFAAA, page 49--52 - Dec. 2002.

[103] D. MESQUITA, F. MORAES, J. PALMA, L. MÖLLER et N. CALAZANS. « Remote and Partial Reconfiguration of FPGAs : Tools and Trends ». Dans Proc. International Parallel and Distributed Processing Symposium (IPDPS'03), pages 177–185. IEEE, 2003.

[104] C. STEIGER, H.WALDER et M. PLATZNER. « Heuristics for Online Scheduling Real-Time Tasks to Partially Reconfigurable Devices ». Dans P. Y. K. CHEUNG, G. A. CONSTANTINIDES et J. T. de SOUSA, éditeurs, Proc. of the 13th International Conference on Field Programmable Logic and Applications (FPL'03), volume 2778 de LNCS, pages 575–584, Berlin, Heidelberg, 2003. Springer Verlag.

[105] S. Trimberger, D. Carberry, A. Johnson and J. wong. A Time-Multiplexed FPGA. IEEE Symposium on FPGAs for Custom Computing Machines , FCCM 97, pp 22-28, 1997.

[106] M. Auguin, K. Ben Chehida, J.P. Diguet, X. Fornari, A.M. Fouilliart, C. Gamrat, G. Gogniat, P. Kajfasz, Y Le Moullec. Partitionning and CoDesign Tolls & Methodology for Reconfigurable Computing : the EPICURE Philosophy. In Proceeding of the Third International Workshop on Systems, Architectures, Modeling Simulation, SAMOS 03, Samos, Greece, July 2003.

[107] L. Kessal, D. Demigny, R. Bourguiba, N. Boudouani. Architecture reconfigurable méthodologie et modélisation VHDL pour la mise au point d'applications. 6ème Symposium en Architectures Nouvelles de Machines, SympA'6, Besançon, France, 19-22 juin 2000.

[108] S. Trimberger, Scheduling Designs into a Time-Multiplexed FPGA, FPGA'98 Proceedings, Monterey, CA, February 22-24, 1998, p. 153-160.

[109] 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.

[110] 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.

[111] 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.

[112] Chameleon Systems. http://www.chameleonsystems.com/

[113] D. Bursky, Upgraded DSP Core Tackles Future Communication Needs, Electonic design, 48 (8): 66-68, Avril 2000.

[114] Jennifer Eyre, Jeff Bier: DSP Processors Hit the Mainstream. IEEE Computer 31(8): 51-59 (1998).

[115] P.Kievits, E. Lambers, C.Morman, and R. Woudsma, R.E.A.L. DSP Technology for Telecom Baseband Processing. Technical Report, Philips Semiconductors, ASIC Servie, Group, 1998.

[116] E. Van, der, Horst, W. Kloosterhuis, and J. van der Heyden. A C Compiler for the Embedded R.E.A.L DSP. In International Conference on signal Processing (ICSPAT 98), Toronto, Canada, September 1998.

[117] http://www.stretchinc.com/

[118] H. Singh, M-H Lee, G. Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M.C. Filho. Morphosys : An integrated recon_gurable system for data-parallel and computation-intensive applications. In IEEE transactions on computers, volume 49, No. 5, pages 465{481, May 2000.

[119] Chameleon Systems. Inc. "Chameleon systems { your communications platform". http ://www.chameleonsystems.com.

[120] G. Sassatelli. Architectures Reconfigurables Dynamiquement pour les Systèmes sur Puce. Ph.D. Thesis, Université de Montpellier, France, April 2002.

[121] G. Sassatelli, L. Torres, P. Benoit, T. Gil, C. Diou, G. Cambon, J. Galy. Highly Scalable Dynamically Reconfigurable Systolic Ring-Architecture for DSP Applications. In IEEE Design Automation and Test in Europe, DATE 02, pp. 553-557, Paris, France, March 2002.

[122] A. Laffely, J. Liang, P. Jain, N. Weng, W. Burleson, R. Tessier. Adaptative System on a Chip (aSoc) for Low-Power Signal Processing. Thirty-Fith Asilomar Conference on Signals, and Computers, Nov 4-7, 2001.

[123] W. Burleson, R. Tessier, D. Goeckel, S Swaminathan, P. Jain, J Euh, S. Venkatraman, V. Thyagarajan. Dynamically Parameterized Algorithms and Architectures to Exploit Signal Variations for Improved Performance and Reduced Power. In International Conference on Acoustic, Speech, and Signal Processing, ICASSP 01, 2001.

[124] W. Burleson, P. Jain, S. Venkatraman. Dynamically Parameterized Architecture for Power-Aware Video Coding Motion Estimation and DCT. Second USF International Workshop on Digital Computational Video, DCV 01, 2001.

[125] J. Liang, A. Laffely, S. Swaminathan, R. Tessier. An Architecture for Saclable On-Chip Communication. Technical Report, University of Massachusetts, Amherts, USA, September 2002.

[126] R. David. Architecture reconfigurable dynamiquement pour applications mobiles. Ph.D. thesis, Univesité de Rennes 1, juillet 2003.

[127] S. Pillement, Méthodologies d'évaluation et de prototypage des systèmes numériques integers; PhD thesis, université de Montpellier II, Décembre 1998.

[128] J. Hauser, Augmenting a microprocessor with reconfigurable Hardware, PhD thesis, University of California, Berkeley, 2000.

[129] C. Rupp, M. Landgnth, T. Graverick, E.Gomersall and H. Holt, The NAPA Adaptative Processing Architecture. In IEEE Symposium on Field Programmable custom Computing Machines (FCCM98) pages 28-37, April 1998.

[130] J-Y. Mignolet, V. Nollet, P. Coene, D.Verkest, S. Vernalde, R. Lauwereins, "Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip", Proceedings of Design, Automation and Test in Europe (DATE) Conference, pp. 986-991, Munich, Germany, March 2003.

[131] Roman L. Lysecky, Frank Vahid: A Configurable Logic Architecture for Dynamic Hardware/Software Partitioning. DATE 2004: 480-485.

[132] B. Dave, N. Jha, CASPER, concurrent hardware-software co-synthesis of hard real time aperiodic and periodic specifications of embedded system architectures, DATE'98, Paris, pp 118-124, 1998.

[133] David Decotigny, une infrastructure de simulation modulaire pour l'évaluation de performances de systèmes temps réel, thèse de doctorat, Université de Rennes 1, Avril 2003.

[134] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, R. Lauwereins, "Interconnection Networks Enable Fine-Grain Dynamic Multi-Tasking on FPGA's", FPL'02, p795-805, Montpellier, France.

[135] P. Merino, M. Jacome, J.C. Lopez, "A Methodology for Task Based Partitioning and Scheduling of Dynamically Reconfigurable Systems", Proc. IEEE Symp. on FPGA's for Custom Computing Machines (FCCM), p324-325, 1998.

[136] G. Wigley, D. Kearney, "Research Issues in Operating Systems for Reconfigurable Computing", Proc. Of the 2nd International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'02), p10-16, Las Vegas, Nevada, USA, June 2002.

[137] http://csapp.cs.cmu.edu/public/ch9-preview.pdf

[138] D. Stewart, Measuring Execution Time and Real-Time Performance, Embedded Systems Conference (ESC), 2001.

[139] Programmation Linux WARREN W.GAY, Paris, 1999, ISBN 2-7440-0727-7

[140] R.A Quinnell, "debugging real-time systems, technical editor. ISSN 0012-7515, EDN -BOSTON THEN DENVER journal.

[141] Y. Thoma et E. Sanchez. « CoDeNios : A Function Level Co-Design Tool ». Dans Workshop on Computer Architecture Education, WCAE 2002, Workshop Proceedings, pages 73–78, Anchorage, Alaska, 2002.

[142] B. Reistad and D. K. Gifford. "Static dependent costs for estimating execution time". In Proc. of the 1994 ACM Conference on LISP and functional programming, pages 65–78. ACM Press, June 1994.

[143] Frank Vahid and Daniel D.Gajski. « Incremental Hardware Estimation During Hardware/Software Functional Partitioning » IEEE Transactions on very large scale integration (VLSI) Systems. 3, No. 3, September 1995. pp 459-464.

[144] G. Stitt, R. Lysecky, F. Vahid. Dynamic Hardware/Software Partitioning: A First Approach. Design Automation Conference (DAC),pp 250-255, June 2003

[145] R. Freund. Optimal selection theory for super concurrency. In Proceedings of the 1989 Supercomputing Conference, pages 13–17. IEEE Computer Society Press, 1989.

[146] A. Khokhar, V. Prasanna, M. Shaaban, and C.-L.Wang. Heterogeneous supercomputing: Problems and issues. In Proc. of the 1992 Workshop on Heterogeneous Processing, pages 3–12. IEEE Computer Society Press, March. 1992.

[147] A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C.-L. Wang. Heterogeneous computing: Challenges and opportunities. IEEE Computer, 26(6):18–27, June 1993.

[148] D.Pease, A. Ghafoor, I. Ahmad, D. L. Andrews, K. Foudil-Bey, T. E. Karpinski, M. A. Mikki, and M. Zerrouki. PAWS: A performance evaluation tool for parallel computing systems. IEEE Computer, 24(1):18–29, January. 1991.

[149] H. J. Siegel. Heterogeneous computing. Annual Research Summary 5.92, http://ece.www.ecn.purdue.edu/ Researchsummary/Section5/sec5 92.html, 1994.

[150] J. Yang, I. Ahmad, and A. Ghafoor. Estimation of execution times on heterogeneous supercomputer architectures. In the 1993 Inter. Conf. on Parallel Processing, volume 1, pages 219–226. CRC Press, Aug. 1993.

[151] T. Yang and A. Gerasoulis. DSC: Scheduling tasks on an unbounded number of processors. IEEE Trans. Parallel and Distributed Systems, 5(6):951–967, Sept. 1994.

[152] M. V. Devarakonda and R. K. Iyer. Predictability of process resource usage: A measurement-based study on UNIX. IEEE Trans. Software Engineering, 15(12):1579–1586, Dec. 1989.

[153] M. A. Iverson, F. ¨ Ozg¨uner, and G. Follen. Run-time statistical estimation of task execution times for heterogeneous distributed computing. In Proc. of the 1996 High Performance Distributed Computing Conference, pages 263–270, Syracuse, NY, Aug. 1996.

[154] T. Kidd, D. Hensgen, L. Moore, R. Freund, D. Charley, M. Halderman, and M. Janakiraman. Studies in the useful predictability of programs in a distributed and homogeneous environment. The Smartnet Home Page (http://papaya.nosc.mil:80/SmartNet/), 1995.

[155] Michael A. Iverson, Füsun Özgüner, Lee C. Potter, Statistical Prediction of Task Execution Times Through Analytic Benchmarking for Scheduling in a Heterogeneous Environment, Eighth Heterogeneous Computing Workshop April 12 - 12, 1999 San Juan, Puerto Rico.

[156] A. Ali, A. Anjum, J. Bunn, R. Cavanaugh, F. van Lingen, R. McClatchey, M. Atif Mehmood, H. Newman, C. Steenberg, M. Thomas, I. Willers. "Predicting the Resource Requirements of a Job Submission". Computing in High Energy Physics, Interlaken, Switzerland, 2004, paper 273.

Publications Personnelles

Revues internationales

[1] GHAFFARI Fakhreddine , AUGUIN Michel, ABID MOHAMED, BEN JEMAA Maher, "Dynamic and On-line Design Space Exploration for Reconfigurable Architectures", To appear in the special issue of the HiPEAC Transactions on High-Performance Embedded Architecture and Compilers.

Conférences internationales

[2] GHAFFARI Fakhreddine , AUGUIN Michel, "An efficient on-line Approach for On-Chip HW/SW Partitionner and Scheduler", to appear in DRS Workshop 2006 Frankfurt, Germany, Mars 12th - 17, 2006

[3] GHAFFARI Fakhreddine , AUGUIN Michel, ABID MOHAMED, BEN JEMAA Maher, "An adaptive HW/SW Partitioning for soft real time reconfigurable Systems", 8th EUROMICRO CONFERENCE ON DIGITAL SYSTEM DESIGN Porto, Portugal, August 30th - September 3rd, 2005

[4] F. Ghaffari, M. Benjemaa, M. Abid, M, M. Auguin, « On line HW/SW Partitioning and scheduling for data dependent execution time applications », 30th EUROMICRO Conference/Proceedings of the Work In Progress Session ISBN 3-902457-05-8, pp 43-44. Rennes – France, 1- 3 September 2004

[5] F.GHAFFARI, M.ABID, M.BENJEMAA, M.AUGUIN. Algorithme de prédictions statistiques du temps d'exécution basée sur la méthode de KPPV, IEEE Int congrès SCS'2004, pp. 265-271 Monastir, Tunisie 18-21 Mars 2004

[6] F.GHAFFARI, M.BENJEMAA, M.AUGUIN. HW/SW Partitioning of Embedded Applications with Variable execution time on a Reconfigurable Architecture. IEEE Int. Conf. SSD 2003 Sousse, TUNISIA 26 – 28 March 2003.

[7] F.GHAFFARI, M.BENJEMAA, M.AUGUIN. Algorithms for the Partitioning of Applications containing variable duration tasks on reconfigurable architectures. IEEE Int. Conf. AICCSA 2003 Tunis, TUNISIA 14 – 18 July 2003.

[8] Fakhfakh-Ghribi, F. Ghaffari, M. Ben Jemaa and M. Abid, "Execution Time Assessment of an Application on a Heterogeneous Architecture. Case of Study: Motion Detection" IEEE Int. Conf. SSD 2005 Sousse, TUNISIA 21 – 24 March 2005.

Conférences nationales

[9] F.GHAFFARI, M.AUGUIN, M.BENJEMAA. M.ABID Approche de partitionnement en ligne d'applications à temps d'exécution variable, Renpar'15 /ASF/SYMPAAA. pp. 365-371, La colle sur loup, FRANCE 14 – 17 Octobre 2003.

[10] F.GHAFFARI, M.AUGUIN, M.BENJEMAA. Etude du partitionnement logiciel/matériel d'applications à distribution variable de charge de calcul. Renpar'14 /ASF/SYMPA pp. 334-338 Hammamet, TUNISIE 10 – 13 avril 2002.

[11] F.GHAFFARI, M.AUGUIN, M.BENJEMAA. Partitionnement d'applications à temps d'exécution variable sur architectures reconfigurables. JFAAA'2002 Monastir, TUNISIE 16 – 18 Décembre 2002.

[12] F.GHAFFARI, M.AUGUIN, M.BENJEMAA. Portage d'un noyau multitâche temps réel sur un processeur embarqué. GEI 2003 Mahdia, TUNISIE 18 – 20 Mars 2003.

Résumé

Les défis actuels du développement des systèmes embarqués complexes tels que les systèmes intégrés de traitement d'image, consistent à réaliser avec succès des produits fiables, performants, efficaces quelles que soient les conditions d'utilisation et peu coûteux. Relever ces défis passe par un bon choix d'architecture, de méthodes et outils adaptés aux applications visées et aux technologies cibles. Pour de nombreuses applications, en particulier en télécommunication et multimédia, des réalisations temps réel souple sont souvent suffisantes, c'est-à-dire des implémentations visant à obtenir une qualité de service adaptée aux besoins.
Au lieu de s'appuyer sur des temps d'exécutions pire cas ou des séquences de test souvent peu représentatives pour concevoir ces systèmes, notre approche vise une plate-forme auto-adaptative capable de s'auto-configurer au cours de l'exécution de l'application (donc en ligne). On peut citer comme exemples d'applications le cas d'une caméra fixe de télésurveillance qui adapte ses traitements en fonction de la nature des images acquises ou un terminal mobile multimodal qui change de norme de transmission si la qualité du canal de communication l'exige.
Les composants reconfigurables ont des niveaux de performances et une flexibilité qui les rendent très attractifs dans un nombre croissant de développements. La reconfiguration dynamique (partielle ou complète) offre la possibilité de réutiliser les mêmes ressources matérielles pour une succession de traitements, et ce de façon analogue à une réalisation logicielle. Nous proposons une approche permettant d'allouer et d'ordonnancer dynamiquement les tâches d'une application flot de données en fonction d'une estimation de leurs temps d'exécution afin de respecter au mieux les contraintes de temps. Cette reconfiguration en ligne nécessite des recherches de compromis complexité/efficacité de l'allocation et de l'ordonnancement afin d'optimiser la qualité de service et de réduire leurs coûts de réalisation.

Mots-clés: Partitionnement logiciel/matériel, architecture reconfigurable dynamiquement, plateforme auto-adaptative, ordonnancement en ligne, prédictions des temps d'exécution, temps réel souple

Abstract

The current challenges of the development of the complex embedded systems such as the integrated systems of image processing, consist to successfully realizing products reliable, powerful, inexpensive and effective whatever the conditions of use. To take up these challenges passes by a good choice of architecture, methods and tools adapted to the applications concerned and target technologies. For many applications, in particular in telecommunication and multi-media, soft real time realizations are often sufficient, i.e. implementations aiming to obtaining a quality of service adapted to the needs. Instead of being based on the Worse Case Execution Times (WCET) or sequences of test often not very representative to design these systems, our approach targets a self-adapting platform able to be configured during the execution of the application (thus on line). We can quote as examples of applications the case of a fixed camera of remote monitoring which adapts its processing according to the nature of the acquired images or a multimode mobile terminal which changes its standard of transmission if the quality of the communication channel requires it.

The reconfigurable components have levels of performances and flexibility which make them very attractive in a growing number of developments. Dynamic reconfiguration (partial or full) makes it possible to re-use the same hardware resources for a succession of processing, and this in similar way to a software realization. We propose an approach allowing to allocate and schedule dynamically the tasks of a data flow application according to an estimate of their execution times in order to respect the time constraints as well as possible. This on line reconfiguration requires research of compromise complexity/effectiveness of the allocation and scheduling in order to optimize the quality of service and to reduce their costs of realization.

Key words: Hardware/software partitioning, dynamically reconfigurable architecture, self-adaptive platform, on-line scheduling, prediction of execution time, soft real time.