Ordonnancement pour processeurs à parallélisme d'instructions en utilisant des techniques de recherche de motifs

Hervé Yviquel

Université de Rennes1
IFSIC
Master Recherche en Informatique

Rapport de stage

Ordonnancement pour processeurs à parallélisme d'instructions en utilisant des techniques de recherche de motifs

par

Hervé Yviquel

Encadrant : François Charot (Équipe CAIRN - INRIA)

Rennes, 2010

Table des matières

Introduction

A l'heure où le monde du système embarqué est en plein développement de par son utilisation dans de plus en plus d'appareils de la vie courante, il est important de concevoir des composants matériels adaptés ainsi que les outils permettant de les exploiter. Les applications déployées sur de tels dispositifs sont principalement dédiées au multimédia et au traitement de l'image ou du signal. Ce type d'application induit un certain nombre de contraintes matérielles telles que le coût de production, la maîtrise de la consommation ou le traitement d'un volume conséquent d'informations numériques.

Les processeurs à parallélisme d'instructions (*Very Long Instruction Word* ou VLIW) apparaissent comme une réponse intéressante à ces défis. En effet, la simplicité de leur architecture induit un coût de conception plus faible. De plus, l'exécution de plusieurs opérations à chaque cycle du processeur permet une accélération significative des performances en comparaison de solutions plus classiques de type RISC (*Reduced Instruction Set Computer*). Enfin, leur faible consommation en font des candidats idéaux dans le domaine de l'embarqué.

Dans le but de satisfaire les différentes contraintes matérielles, une exploration architecturale peut permettre de définir les paramètres optimaux d'un processeur VLIW pour une application donnée tels que le nombre d'unités fonctionnelles, le nombre de registres, etc... Les paramètres du processeur sont ajustés en fonction du niveau de parallélisme d'instructions de l'application. De la même manière l'utilisation de jeux d'instructions spécifiques à une application est adaptée à une utilisation au sein des systèmes embarqués qui sont, dans la majeure partie des cas, dédiés à un traitement spécifique. Toutes ces spécialisations permettent d'améliorer efficacement le rapport entre performance, surface et consommation.

Ce rapport présente un nouvel outil dont le but est de définir les paramètres optimaux d'un processeur de type VLIW pour une application donnée en termes de dimensionnement, d'organisation et de spécialisation de son jeu d'instructions. Cet outil repose sur la modélisation des problèmes à résoudre en utilisant la programmation par contraintes et exploite la technique de couverture de graphe à l'aide de motifs de calculs. Différentes structures de processeurs pourront alors être comparées en termes de performance et de complexité matérielle.

Ce rapport est organisé de la manière suivante : le premier chapitre réalise un état de l'art sur les processeurs VLIW, leurs organisations architecturales et les différentes méthodes de spécialisation ; le deuxième chapitre présente la technique de programmation par contraintes et la couverture de graphe ; enfin, le dernier chapitre décrit en détail l'outil développé, en d'autres termes le modèle de processeur utilisé, la modélisation détaillée du problème à résoudre et les expérimentations réalisées.

Chapitre 1

État de l'art

Cet état de l'art s'articule de la manière suivante : le première section introduit le concept de processeur à instructions longues et décrit certaines techniques d'ordonnancement associées, la deuxième section présente les différentes études portant sur l'organisation d'un processeur VLIW en plusieurs groupes d'unités fonctionnelles et la dernière détaille les différents travaux portant sur la spécialisation des processeurs de par leur jeu d'instructions et leur organisation.

1.1 Processeur VLIW

Contrairement à beaucoup d'autres architectures comme le x86 d'Intel, le modèle de processeur VLIW permet d'exposer le parallélisme d'instruction au sein de l'architecture. Ainsi, le parallélisme n'est pas réalisé au niveau des processus, ni sur un traitement par lots de nombreuses données, mais réellement au niveau de chaque instruction du programme.

1.1.1 Principe

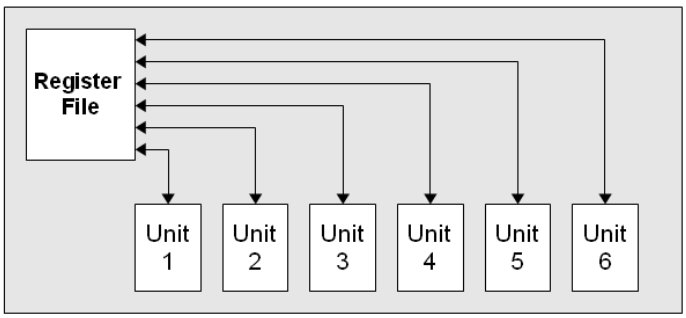


FIGURE 1.1: VLIW traditionnel

Le concept initial de processeur à instructions longues a été inventé par Joseph A. Fisher de l'université de Yale en 1983 [4]. Une architecture de type VLIW contient de nombreuses unités fonctionnelles comme l'illustre la figure 1.1. Une instruction consiste en plusieurs champs et chaque champ définit l'opération réalisée par une des unités fonctionnelles de l'architecture. Afin d'exploiter ce parallélisme et malgré des dépendances entre certaines instructions, un

compilateur destiné au VLIW doit être capable d'effectuer un ordonnancement plus complexe qu'un compilateur classique. Ainsi l'avantage d'un processeur VLIW varie en fonction du niveau de parallélisme (*Instruction-Level Parallelism* ou ILP) présent au sein de l'application qu'il exécute. La simplicité de l'architecture est le résultat d'un transfert de complexité vers la compilation.

Il est possible de rapprocher les processeurs VLIW des processeurs de type superscalaire : en effet ces processeurs exécute aussi plusieurs instructions par cycle. Cependant dans ces derniers, les dépendances de données et de contrôles sont gérées par le matériel et non par le compilateur, ce qui implique une complexification de l'architecture.

1.1.2 Ordonnancement

Afin d'exploiter au maximum un processeur VLIW, il est nécessaire de faire ressortir le parallélisme d'instruction du code. En effet, cette architecture est basée sur un modèle simple dans lequel peu de mécanismes matériels sont offerts. La phase de compilation devient alors plus complexe car elle requiert de nombreux traitements.

Il a été démontré que trouver un ordonnancement optimal est un problème NP-complet. Afin de s'en rapprocher au maximum, des heuristiques ont été développées.

Certaines méthodes, communément admises comme les plus adaptées aux applications destinées au VLIW, sont décrites dans cette partie. Elles utilisent principalement la théorie des graphes dans le but de réaliser un ordonnancement efficace.

List Scheduling

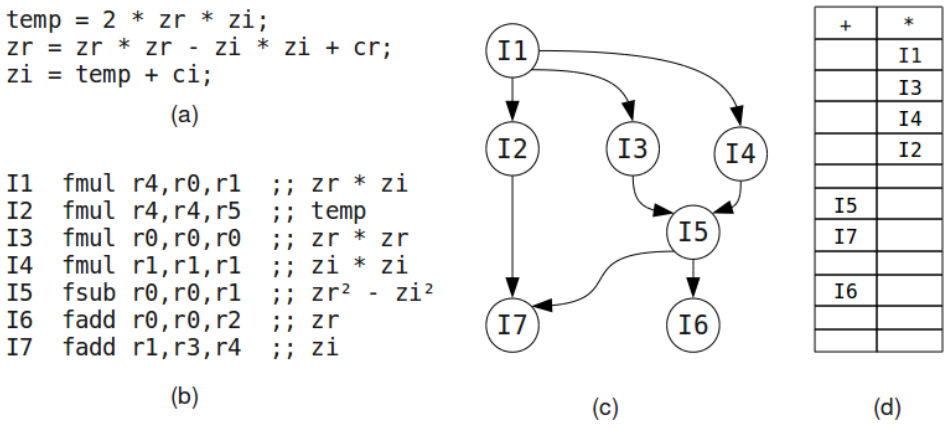


FIGURE 1.2: Exemple d'application du list scheduling

La technique de *list scheduling* est un grand classique de l'ordonnancement [6]. Elle est destinée à organiser l'ordre d'exécution des instructions au sein d'un bloc de base, c'est-à-dire une partie de code qui ne contient pas de branchement.

La figure 1.2 va permettre d'expliquer le principe de l'algorithme sur un exemple. Les parties (a) et (b) représentent le code d'un bloc de base. (a) est sa version pseudo-code et (b) une version simplifiée du code assembleur.

Le principe est assez simple, il suffit de réaliser un graphe de dépendance de données (c) de la manière suivante : chaque nœud du graphe représente une instruction et il y a un arc entre deux nœuds si les deux instructions qu'il représente utilisent un ou plusieurs registres

identiques. L'orientation de l'arc dépend donc de l'ordre dans lequel doivent être exécutées les instructions. Il est ainsi possible d'avoir plusieurs graphes distincts si ils n'ont aucune dépendance entre eux.

Il faut ensuite ordonnancer ces instructions de manière à les exécuter le plus tôt possible tout en respectant le fait qu'une instruction ne peut être exécutée que si tous ses prédécesseurs ont déjà été exécutés. L'algorithme est détaillé dans la figure 1.3. La partie (d) de la figure 1.2 représente l'ordonnancement obtenu sur une machine pipelinée qui contient deux unités fonctionnelles : une pour l'addition et l'autre pour la multiplication, tout en considérant un temps de traitement des instructions de trois cycles.

```
void list-scheduling(){
  g = buidDependanceGraphe(prog)
  l = new list()
  for(each root r in g; sorted by priority)
    enqueue(r)
  while(notEmpty(l)){
    h = dequeue()
    schedule(h)
    for(each successor s of h){
      if(all successors of s have been scheduled)
        enqueue(s)
      }
    }
}
```

FIGURE 1.3: Algorithme de list scheduling

Trace Scheduling

Au niveau d'un bloc de base, l'ILP est souvent insuffisant, c'est pourquoi il est nécessaire d'observer le code dans son ensemble et donc de prendre en compte les branchements. C'est dans ce contexte que des techniques comme le *trace scheduling* ont été développées.

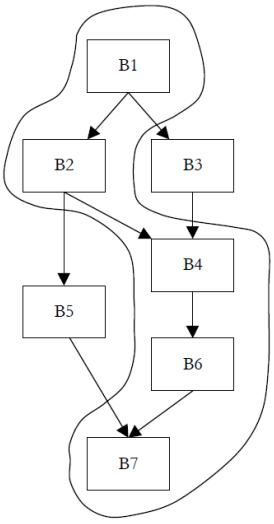


FIGURE 1.4: Trace de programme

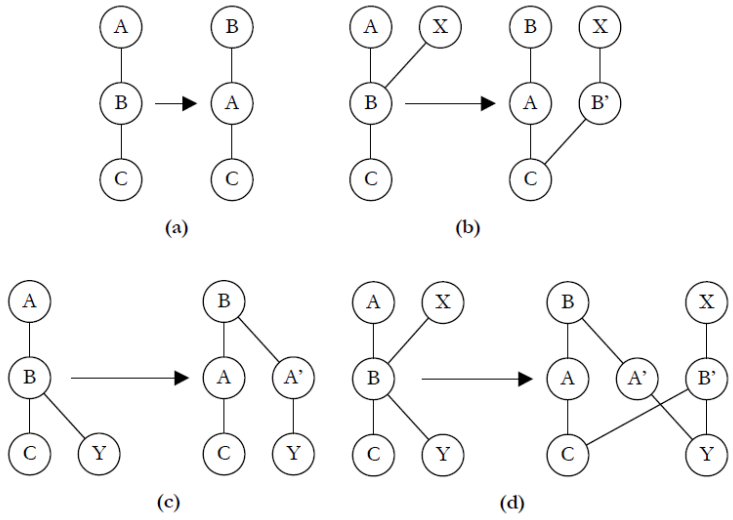


FIGURE 1.5: Code de compensation

Joseph Fisher a introduit une technique basée sur le *list scheduling* et l'a adaptée à l'ordonnancement d'un programme complet destiné au VLIW. Cette technique, appelée *trace scheduling*, est issue du compilateur dédié aux processeurs Trace de Multiflow [3].

Afin d'utiliser cette technique, il est nécessaire de générer le graphe de flot de contrôle du programme. La première partie consiste à sélectionner une trace ou plus exactement un chemin acyclique dans ce graphe. La figure 1.4 décrit une de ces traces. La trace sélectionnée est celle qui a le plus de probabilité d'être exécutée. On utilise ensuite la technique de *list scheduling* sur cette trace, puis on sélectionne une autre trace et on itère ainsi le traitement jusqu'à ce que chaque trace du graphe ait été couverte.

Cependant, avec une telle technique, une difficulté peut se poser au niveau des branchements. En effet, une instruction peut être déplacée par l'ordonnancement avant ou après le branchement et ne plus être exécutée. C'est pourquoi, lorsqu'une instruction au niveau d'un branchement, séparation ou jointure, est réordonnée, il nécessaire d'ajouter du code de compensation. Les différents cas sont décrits par la figure 1.5 qui expose le code avant puis après le déplacement d'une instruction avec éventuellement l'ajout de code de compensation.

L'impossibilité d'exécuter l'algorithme sur un graphe cyclique apparaît comme une des contraintes or le graphe de flot en contient dès que des boucles sont utilisées dans le code du programme. Afin de palier à ce problème, il est possible de remplacer chaque boucle par un unique nœud dans le graphe, obtenant ainsi une représentation abstraite de la boucle. La figure 1.6 décrit cette modification du graphe sur un exemple avec deux boucles imbriquées.

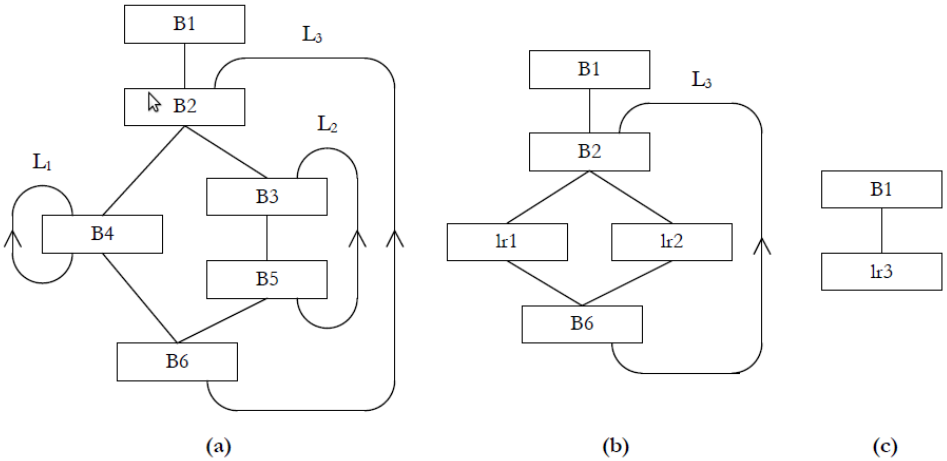


FIGURE 1.6: Exemple de suppression de boucle

Diverses optimisations sont possibles afin d'améliorer cet algorithme [6] comme la suppression du code de compensation inutile afin de diminuer la taille du code ou encore le *superblock scheduling* qui consiste à supprimer tous les branchements de type jointure afin de faciliter le *trace scheduling* ainsi que certaines optimisations classiques de compilation comme la propagation de copie.

Il est aussi possible d'ajouter des fonctionnalités matérielles au processeur afin de faciliter l'ordonnancement [6].

Ainsi, l'exécution spéculative permet de faciliter le déplacement d'instructions. En effet, dans une architecture classique, il est nécessaire de vérifier plusieurs propriétés avant de pouvoir déplacer une instruction : elle ne doit pas causer d'exception, réécrire sur un registre dont aura besoin un de ses successeurs ou encore modifier la mémoire de la machine. Il est possible d'utiliser le matériel pour limiter ces contraintes et ainsi faciliter le *trace scheduling*.

L'exécution prédictive supprime les branchements conditionnels et les remplace par une instruction qui définit un prédicat, toute la structure devenant alors un bloc de base avec des instructions s'exécutant selon la valeur du prédicat. La gestion de ces prédicats doit être une fonctionnalité ajoutée au matériel.

Enfin, une autre technique appelée *delayed issue* consistant à transmettre au matériel le délai à l'instruction pourra être exécutée. Elle permet ainsi une pseudo-exécution dans le désordre [1] et facilite le *trace scheduling*. Il est alors nécessaire d'implémenter au sein du processeur une structure permettant d'insérer les instructions à exécuter dans des files d'attente spécifiques à chaque unité fonctionnelle.

Cependant, ces techniques sont possibles uniquement sur des architectures très spécifiques. Aussi, il est important de remarquer que le développement d'une architecture doit être fait en cohésion avec celui des techniques et des outils.

1.2 Groupement d'unités fonctionnelles (ou clustering)

En informatique, la manière la plus simple d'améliorer les performances passe souvent par l'ajout de nouveaux composants (augmentation du nombre de transistors, ajout de nouveau cœur de processeur, ...). Cependant, cette solution se heurte le plus souvent au problème de passage à l'échelle. Ce type de problème se pose dans le cas du nombre d'unités fonctionnelles au sein d'un processeur VLIW. La séparation du processeur en multiples groupes d'unités fonctionnelles, ou clusters, est une solution à ce problème.

1.2.1 Principe

Le processeur VLIW traditionnel, tel que l'a défini Fisher en 1983, est constitué d'une unique file de registres où chacune des unités fonctionnelles est connectée à l'ensemble des registres. Dans ce contexte, l'augmentation du nombre d'unités fonctionnelles implique une explosion de la complexité de la connectique du processeur et une diminution de ses performances [18].

Une solution consiste alors en une fragmentation de la file de registres en plusieurs parties et en une connexion de chaque partie à un sous-ensemble des unités fonctionnelles. Une séparation du processeur en plusieurs parties appelées clusters est ainsi mise en évidence. La complexité est alors transférée au niveau de la compilation qui doit mettre en œuvre la communication entre ces différents clusters sans pénaliser le temps d'exécution des applications.

L'architecture PAC DSP développée récemment par le *SoC Technology Center of Industrial Technology Research Institute* à Taiwan est un exemple [12] de processeur dont l'architecture utilise une telle stratégie. Ce processeur est principalement destiné à une utilisation multimédia au sein d'un système embarqué. La figure 1.7 présente l'architecture du cœur de ce processeur. Cette dernière est constituée de trois parties distinctes : deux clusters, formés par une unité arithmétique et une unité mémoire, ainsi qu'une unité scalaire qui sert principalement à contrôler les sauts et branchements. C'est donc une architecture 5 voies, c'est-à-dire qui est capable de réaliser cinq instructions en parallèle.

Les registres sont répartis sur la puce : certains accessibles uniquement par une seule unité fonctionnelle, d'autres partagés entre deux. Les échanges de données au niveau des registres partagés sont décrits par la figure 1.8. Ces derniers, nommés fichiers de registres ping-pong, sont découpés en deux parties distinctes : D0-D7 et D8-D15. Chaque partie est accessible par une seule unité fonctionnelle à un moment donné. De plus, le transfert de données entre clusters ou avec l'unité scalaire nécessite d'utiliser des instructions spécifiques.

1. Out-of-order execution

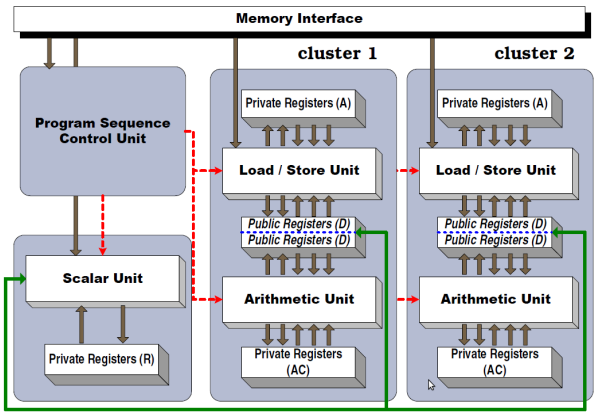


FIGURE 1.7: Architecture du PAC DSP

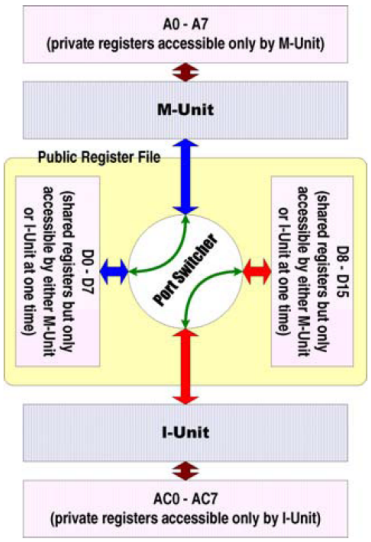


FIGURE 1.8: Registres partagés du PAC DSP

L'utilisation de banques de registres distribuées complexifie le travail de compilation. En effet, il faut prendre en compte tout au long du processus que les échanges de données entre les différents fichiers de registres sont soumis à de nombreuses contraintes : accès limités, transferts de données entre registres assez lents, etc. Par conséquent, il est nécessaire d'utiliser des algorithmes d'ordonnancement qui prennent en compte les caractéristiques de l'architecture ainsi que ses contraintes.

1.2.2 Communication inter-clusters

Ces dernières années, différentes approches de communication inter-clusters ont été étudiées [18]. Si on se place au niveau de la compilation, celles-ci peuvent être classifiées en deux groupes principaux : premièrement, celles qui nécessitent l'ajout d'instructions supplémentaires pour réaliser la copie d'un registre d'un cluster vers un autre ; secondement, celles qui le font de manière plus transparente où les unités fonctionnelles ont accès à tous les registres, sous certaines contraintes et grâce à des mécanismes architecturaux plus complexes.

L'architecture Lx [2], sur laquelle sont basés les processeurs de la gamme ST200 de STMicroelectronics, est un exemple d'implémentation de la première catégorie de communication. Dans cette architecture, si une communication entre deux clusters est nécessaire alors une opération de type `copy r3→r1[2]` est utilisée. Le PAC DSP utilise une légère variante de cette technique en utilisant les opérations `send r3` et `receive r1` dans chacun des clusters. Ces opérations sont généralement réalisées sur les unités fonctionnelles qui gèrent les interactions avec la mémoire.

Il est aussi possible de citer le Texas Instruments TMS320C6X qui lui utilise le deuxième type de communication.Dans ce cas, les opérations peuvent directement utiliser le registre d'un autre cluster comme l'un de leurs opérandes, par exemple l'opération `op1 r1[1],r2→r3` réalise une opération entre deux registres appartenant à deux clusters différents. Lors de l'exécution, le pipeline est gelé momentanément lorsqu'une telle communication intervient le temps qu'elle soit réalisée.

1.2.3 Ordonnancement et allocation de registres

A titre d'exemple, il est possible d'évoquer une technique mise au point pour le PAC DSP [12] : elle consiste en un algorithme combinant ordonnancement d'instructions et allocation de registres.

La première phase de l'algorithme sépare de manière aléatoire les instructions et les assigne à chacun des clusters. Ensuite une version modifiée du *list scheduling* est exécutée afin d'ordonnancer les instructions et d'insérer la gestion des communications inter-clusters. La dernière phase est itérative : le partitionnement est modifié et les instructions ré-ordonnancées. Enfin une méthode traditionnelle d'optimisation, appelée *Simulated Annealing*, est utilisée : elle considère la valeur *energy* du résultat qui représente l'évaluation du partitionnement, c'est-à-dire concrètement le nombre de cycles nécessaires à l'exécution de l'ordonnancement. Si le résultat est satisfaisant, l'ordonnancement est conservé ; dans le cas contraire il est rejeté. Cette phase est ensuite répétée sur les résultats conservés jusqu'à ce que la valeur *energy* atteigne un seuil où l'ordonnancement sera définitivement accepté.

Cette stratégie est particulièrement aléatoire, il est donc difficile de définir le temps que mettra le compilateur pour obtenir un résultat satisfaisant.

1.3 Spécialisation d'un processeur VLIW

Étant donné l'amélioration de la densité d'intégration des circuits sur le silicium, il est de plus en plus envisageable pour les industriels d'utiliser des composants matériels spécifiques à leurs besoins. C'est dans cette optique que l'utilisation de processeurs spécialisés devient intéressante : leur production à moins grande échelle impose un surcoût réel mais apporte un gain de performance non négligeable.

Un processeur VLIW peut être spécialisé de deux façons : il est possible de modifier l'architecture en elle même, par exemple en changeant le nombre d'unités fonctionnelles et leur organisation au sein de clusters ; il est également possible d'utiliser des processeurs dont le jeu d'instructions est spécialisé pour une application spécifique, communément appelés *Application Specific Instruction-set Processor* (ASIP).

1.3.1 Dimensionnement et organisation architecturale

Le dimensionnement et l'organisation architecturale, dans le cadre des processeurs à instructions longues, permet principalement d'exploiter le parallélisme d'instructions au maximum afin d'obtenir la meilleure accélération. D'autres paramètres peuvent entrer en considération tels que la taille de la puce, sa consommation, ou encore la quantité de mémoire nécessaire.

Différentes approches ont été étudiées pour résoudre ces problèmes. Dans [17], les auteurs réalisent une recherche exhaustive à l'aide d'un algorithme de *brute-force* ; le temps de recherche peut alors être particulièrement important. Dans [1], les auteurs développent une technique d'exploration utilisant un algorithme génétique qui apprend au fur et à mesure de ses exécutions afin d'accélérer ses prochaines résolutions. Le problème reste que le temps d'apprentissage avant de pouvoir résoudre des problèmes complexes peut être important.

1.3.2 Jeux d'instructions spécialisés

Les ASIP sont des processeurs dont le jeu d'instructions a été spécialisé, c'est-à-dire que des instructions complexes y ont été ajoutées. Ces instructions complexes sont composées de plusieurs opérations simples. Par exemple l'instruction `MAC`, généralement utilisé dans les

processeurs dédiés au traitement du signal (*Digital Signal Processor* ou DSP), permet de réaliser une multiplication et une accumulation à l'aide d'une unique instruction.

Au niveau des processeurs VLIW, plusieurs avantages sont identifiables à un tel procédé [15] : en premier lieu, si certains opérandes sont de taille limitée alors les unités fonctionnelles peuvent être simplifiées, ce qui les rend plus rapides ; en second lieu, l'enchaînement de plusieurs opérations au sein d'une unité fonctionnelle peut réduire le temps d'exécution ; ensuite, les opérations concurrentes sont plus facilement parallélisables au sein d'une unité fonctionnelle qu'entre deux différentes ; pour finir, le fait de regrouper plusieurs opérations successives au sein d'une même unité fonctionnelle évite l'accès aux registres pour stocker et récupérer les résultats intermédiaires ce qui permet de diminuer la pression sur les registres.

Il existe déjà dans le monde de l'industrie des exemples concrets de processeurs VLIW possédant des parties reconfigurables comme le Trimedia développé par Philips [19], ou encore le plus récent multi-processeur Venezia de Toshiba qui intègre un co-processeur VLIW reconfigurable [16]. Tous ces processeurs sont développés pour un usage multimédia au sein d'un système embarqué.

Ces processeurs sont constitués de deux parties distinctes : un première partie équivalente à un processeur VLIW standard et une deuxième partie composée de plusieurs unités reconigurables qui exécutent les instructions spécifiques. Ainsi les auteurs de [13] étudie l'extension du jeu d'instructions de ce type de processeur. Elle est réalisée à l'aide de la recherche de motif de la même manière que les travaux réalisés au cours de ce stage. Un algorithme de *Branch and Bound* est utilisé afin de trouver une solution qui contienne des groupes de motifs pouvant être exécutés en parallèle.

Chapitre 2

Contexte du stage

L'objectif de ce stage est de pouvoir, dans un cadre d'exploration architecturale, comparer différentes structures de processeurs VLIW dans le but d'être capable d'identifier la structure la plus appropriée pour un jeu d'algorithmes donné (compromis entre performance, complexité matérielle de réalisation...). Comme discuté dans le chapitre précédent, la spécialisation d'un processeur à instructions longues pour une application peut être réalisée de deux façons : modifier l'organisation et le dimensionnement du processeur ou spécialiser son jeu d'instructions.

Les travaux présentés dans ce rapport s'appuient sur des travaux récemment réalisés, au sein de l'équipe CAIRN de l'INRIA Rennes, sur le thème de la conception d'ASIP. En effet, l'équipe CAIRN a développée une méthodologie et des outils pour l'extension du jeu d'instructions de cœurs de processeurs de type RISC, l'extension étant mise en œuvre dans un bloc de logique reconfigurable fortement couplé au processeur, comme par exemple le processeur NIOS II d'Altera [14]. La difficulté se situe dans le choix des nouvelles instructions qui s'exécuteront sur les extensions, en effet celles-ci doivent être choisies de manière à accélérer l'exécution de l'application.

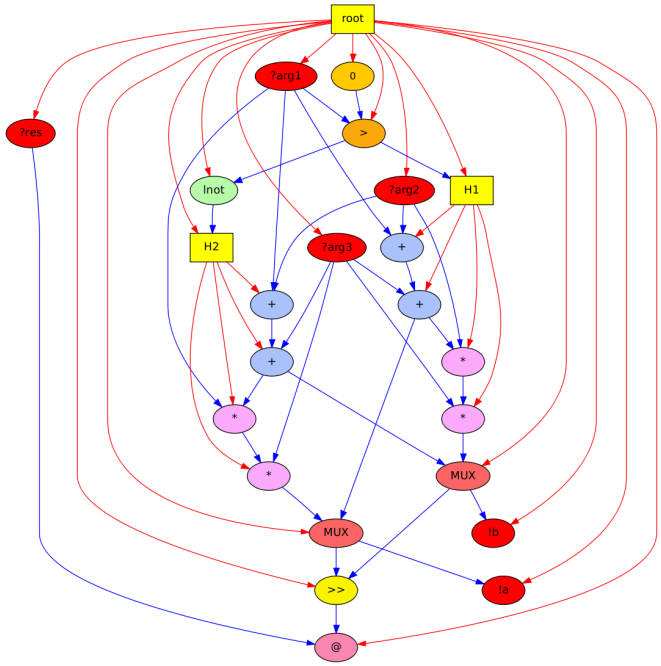
La technique développée à cette intention se base sur la représentation de l'application par un graphe suivie de la couverture de ce graphe à l'aide de motifs de calculs. Les motifs représentent les différentes instructions du processeur et donc, potentiellement, les instructions spécialisées. Dans ce but, il est nécessaire de générer un ensemble de motifs candidats à partir du graphe puis de sélectionner les motifs les plus pertinents. Différentes méthodologies existent pour réaliser la couverture de graphe à l'aide de motifs. Les travaux de l'équipe se basent sur la modélisation des problèmes à résoudre en utilisant la programmation par contraintes. Une fois les problèmes résolus, il ne reste qu'à générer l'architecture à l'aide de la synthèse de haut niveau.

Ce chapitre est composé de plusieurs sections : tout d'abord, la représentation intermédiaire utilisée est présentée ; ensuite, la programmation par contraintes est décrite brièvement ; et enfin, une explication plus détaillée de la manière dont est réalisée la couverture de graphe à l'aide de motifs, avec dans un premier temps la génération des motifs de calculs et ensuite la sélection des plus pertinents.

2.1 Représentation d'une application à l'aide d'un graphe

La représentation sur laquelle sera appliquée l'ensemble des traitements est appelée graphe hiérarchique aux dépendances conditionnées (*Hierarchical Conditional Dependency Graph* ou

HCDG). Cette représentation à été précédemment utilisée afin de réaliser de la synthèse d'architecture de haut niveau [10].




```
int foo(int arg1,int arg2,int arg3,
        int* res) {
  int a;
  int b;
  if(arg1 > 0){
    a = arg1 + arg2 + arg3;
    b = a * arg2 * arg3;
  }
  else{
    b = arg1 + arg2 + arg3;
    a = b * arg1 * arg3;
  }
  *res = a >> b;
}
```

FIGURE 2.1: Exemple d'un programme C et du HCDG généré correspondant

Un HCDG est un graphe dirigé et acyclique qui expose de manière hiérarchique les contrôles et représente explicitement les dépendances de donnée et de contrôle. Cette représentation est adaptée pour modéliser le flot de donnée et de contrôle d'un programme complexe. La figure 2.1 montre un HCDG généré à partir du programme présenté : les blocs rectangulaires représentent les gardes du programme, les rouges sont des entrées-sorties, les arcs bleus expriment le flot de donnée et enfin les arcs rouges le flot de contrôle. Dans la plupart des cas le graphe obtenu est plus simple que sa représentation sous la forme plus classique de graphe de flot de donnée et de contrôle (*Control-Data Flow Graph* ou CDFG), ce qui permet l'application des algorithmes de transformation de graphe et de la plupart des traitements de manière simplifiée.

2.2 Programmation par contraintes

Les travaux réalisés au cours de ce stage et précédemment au sein de l'équipe reposent en partie sur l'utilisation de méthodes de satisfaction de contraintes mises en œuvre dans le solveur de contraintes JaCoP [9].

Dans cette approche, il est nécessaire de définir chacun des problèmes à résoudre. Un problème de satisfaction de contraintes (*Constraint Satisfaction Problem* ou CSP) est défini par un triplet $S = (V, D, C)$ où :

- $V = \{x_1, x_2, ..., x_n\}$ est un ensemble de variables à domaines finis (*Finite Domains Variables* ou FDV) ;
- $D = \{D_1, D_2, ..., D_n\}$ est un ensemble de domaines finis (*Finite Domains* ou FD) de ces variables ;
- et C est un ensemble de contraintes.

Chaque variable est définie par son domaine, c'est-à-dire les différentes valeurs qu'elle peut prendre. Un domaine est défini ici à l'aide d'entiers. Une contrainte $c(x_1, x_2, ..., x_n) \in C$ sur des

variables de V est un sous-ensemble de $D_1 \times D_2 \times ... \times D_n$ qui restreint chaque combinaison de valeurs que les variables peuvent prendre simultanément. Ces contraintes peuvent prendre différentes formes telles que des équations, des inégalités et même des programmes. C'est pourquoi, dans la suite de ce rapport, les différents problèmes à résoudre sont modélisés à l'aide de contraintes.

Une solution à un CSP est une affectation totale et valide des variables, c'est-à-dire que chacune des variables est associée à une valeur de son domaine de manière à ce que toutes les contraintes soient satisfaites. Le problème spécifique à modéliser va déterminer si le solveur recherche une solution optimale, n'importe quelle solution ou toutes les solutions étant donnée une fonction de coût définie par le biais des variables, par exemple le temps d'exécution dans le cas de la modélisation d'un ordonnancement.

Le solveur utilise des méthodes de consistance de contraintes qui tentent de supprimer les valeurs incohérentes des domaines de façon à obtenir un ensemble de domaines réduits tels que leurs combinaisons soient des solutions valides. A chaque fois qu'une valeur est supprimée d'un domaine fini (FD), toutes les contraintes qui contiennent cette variable sont revues.

Cependant, la plupart des techniques de consistance ne sont pas complètes et le solveur a besoin d'explorer les domaines restants par une recherche pour trouver une solution. Cette exploration est effectuée par l'affectation systématique à une variable de chacune des valeurs de son domaine. Ensuite la méthode de consistance est appelée dès que les domaines des variables pour une contrainte donnée sont réduits. Enfin, si une solution partielle enfreint n'importe laquelle des contraintes, un algorithme de retour arrière (*backtrack*) réduit la taille de l'espace de recherche restant.

2.3 Génération de motifs

La génération de motifs est définie pour un graphe acyclique $G = (N, E)$ où N est un ensemble de nœuds et E un ensemble d'arcs. Concrètement, ces graphes sont des HCDG, décrits dans la section 2.1, dont les informations de contrôle ont été supprimées. Seuls les nœuds concernant les instructions et les données sont conservés ainsi que les arcs représentant le flot des données.

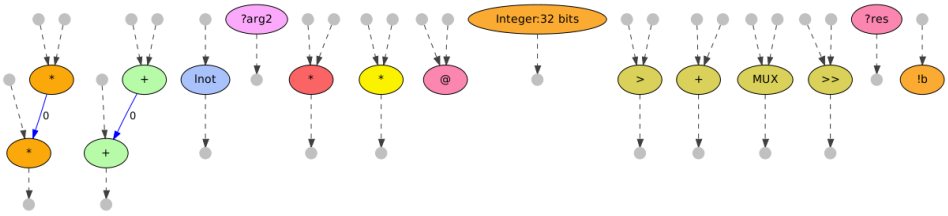


FIGURE 2.2: Motifs générés pour l'exemple de la figure 2.1

Un motif est défini comme un sous-graphe $P = (N_p, E_p)$ du graphe tel que $N_p \subseteq N$ et $E_p \subseteq E$. Un motif doit aussi être un sous-graphe isomorphe du graphe G. Cet isomorphisme doit être topologique, leurs structures sont identiques, mais aussi sémantique, ils réalisent des opérations équivalentes. Une exception est faite au niveau de l'isomorphisme topologique : deux motifs restent isomorphes si leur nombre de sorties diffèrent : en effet, le nombre d'utilisations de leurs données de sortie ne doit pas être un facteur déterminant. La figure 2.2 donne des exemples de motifs générés à partir du HCDG de la figure 2.1. Les deux premiers motifs de cette liste sont des exemples de motifs représentant des instructions complexes définies dans la section 1.3.2.

La détection de l'isomorphisme entre deux graphes est réalisée en utilisant une contrainte de correspondance de graphe appelée *GraphMatch*. Cette contrainte a été implémentée à l'aide d'un algorithme d'élagage développé pour cet objectif précis.

Pour satisfaire certains besoins architecturaux, différentes contraintes ont été ajoutées à la génération de motifs telles que le nombre d'entrées et de sorties, le nombre de nœuds que comprend le motif et le temps d'exécution de son chemin critique [14].

```
// Entrées : G=(N,E) - le graphe d'application
// EMDI - Ensemble de Motifs Définitivement Identifiés
// EMC - Ensemble de Motifs Courants
// EMT - Ensemble de Motifs Temporaires
// n_s - Nœud "graine" (ou seed) du motif
EMDI ← ∅
pour chaque n_s ∈ N
   EMT ← ∅
   EMC ← TrouverTousLesMotifs(G, n_s)
   pour chaque p ∈ EMC
      if ∀_{pattern∈EMT} p ≢ pattern
         EMT ← EMT ∪ {p}
         NMP_p ← TrouverToutesLesOccurrences(G, p)
      end if
   end pour
   NMP_{n_s} ← TrouverToutesLesOccurrences(G, n_s)
   pour chaque p ∈ EMT
      if coef.NMP_{n_s} ≤ NMP_p
         EMDI ← EMDI ∪ {p}
      end if
   end pour
end pour
```

FIGURE 2.3: Algorithme d'identification des motifs

L'algorithme de génération de motifs est illustré par la figure 2.3. La première étape consiste à détecter tous les motifs de calcul, formés autour de chaque nœud graine $n_s \in N$, qui satisfont les différentes contraintes précédemment posées. Cette détection est réalisée grâce à la méthode $TrouverTousLesMotifs(G, n)$ implémentée en utilisant la programmation par contraintes. Lors de l'étape suivante, l'ensemble des motifs temporaires (EMT) est progressivement élargi par les motifs non-isomorphes venant de l'ensemble des motifs courants (EMC). Enfin, les motifs ajoutés à l'ensemble des motifs définitivement identifiés (EMDI) sont ceux dont le nombre d'occurrences dans le graphe de l'application G est suffisamment grand comparé au nombre d'occurrence du nœud graine (modulo un coefficient de filtrage compris entre 0 et 1). Le nombre d'occurrences d'un motif donné dans le graphe G est calculé grâce à la méthode $TrouverToutesLesOccurrences(G, p)$ implémenté en utilisant la programmation par contraintes et en particulier la contrainte *GraphMatch* précédemment décrite.

2.4 Sélection de motifs, couverture de graphe et ordonnancement

La sélection de motifs et l'ordonnancement sont réalisés au cours de la même étape car cela permet de sélectionner les motifs véritablement adaptés à une exécution rapide [5]. Il est aussi possible de réaliser ces opérations en deux étapes moins complexes, mais il serait alors difficile de connaître les motifs réellement intéressants.

La sélection de motifs est réalisée à partir de la liste des motifs identifiés lors de l'étape précédente et du programme représenté sous la forme d'un HCDG. Pour réaliser la sélection, il est nécessaire de savoir quel motif peut couvrir quel nœud du graphe. L'algorithme décrit par la figure 2.4 permet d'obtenir ces informations. Après l'exécution de cette procédure, chaque nœud $n \in N$ est associé à une liste $matches_n$ qui contient l'ensemble des occurrences pouvant le recouvrir.

```
// Entrées : G=(N,E) - le graphe d'application
// EMDI - Ensemble de Motifs Définitivement Identifiés
// M_p - Ensemble des occurrences pour un motif p
// M - Ensemble de toutes les occurrences
// matches_n - Ensemble des occurrences qui peut couvrir un nœud n
M ← ∅
pour chaque p ∈ EMDI
   M_p ← TrouverToutesLesOccurrences(G, p)
   M ← M ∪ M_p
end pour
pour chaque m ∈ M
   pour chaque n ∈ m
      matches_n ← matches_n ∪ {m}
   end pour
end pour
```

FIGURE 2.4: Algorithme trouvant toutes les occurrences qui couvrent un nœud dans un graphe

De plus, à chaque nœud $n \in N$ est associée la variable n_{match} qui contient l'ensemble des index des occurrences recouvrant n, à savoir $n_{match} :: \{i \mid m_i \in matches_n\}$. Cette variable est définie comme une variable de décision, c'est-à-dire que le solveur doit impérativement lui attribuer une unique valeur avant la fin de la résolution : cela permet d'assurer qu'un nœud n'est recouvert que par une unique occurrence. Par exemple, la figure 2.5 illustre les occurrences identifiées et sélectionnées de la figure 2.6.

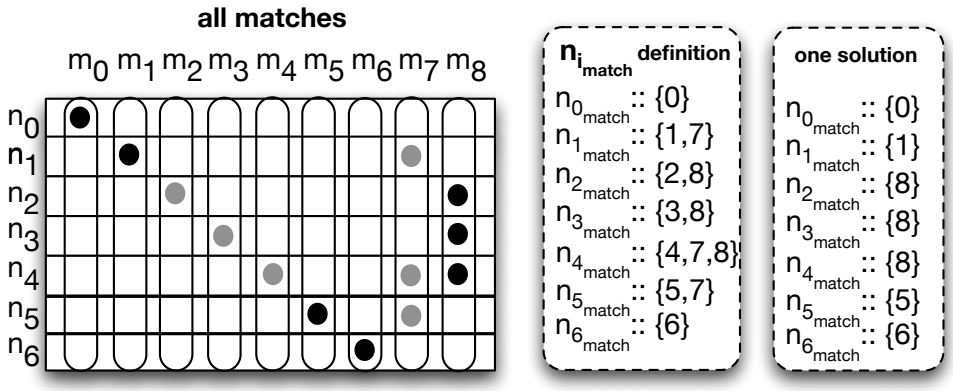


FIGURE 2.5: Occurrences identifiées et sélectionnées pour l'exemple de la figure 2.6

Enfin la contrainte 2.1 calcule le nombre d'utilisations d'une occurrence par ces différents nœuds. La variable m_{set} correspond ici à l'union de tous les n_{match} des nœuds n appartenant à m. Il suffit ensuite de définir la variable réifiée m_{sel} qui illustre la sélection ou non d'une occurrence

à partir de la contrainte 2.2. En effet, si l'un des nœuds qui compose cette occurrence a choisi de la sélectionner alors tous ses nœuds doivent en faire autant.

$$\forall m_i \in M : Count(i, m_{set}, m_{count}) \tag{2.1}$$

$$m_{count} > 0 \Leftrightarrow m_{sel} \tag{2.2}$$

La sélection des différents motifs est réalisée dans le but d'obtenir le temps d'exécution minimum en particulier dans le cas d'un ordonnancement parallèle. Il est donc nécessaire de modéliser l'écoulement du temps et les dépendances entre les différentes opérations. C'est pourquoi de nouvelles variables sont ajoutées : d'abord n_{start} et m_{start} qui représentent le début de l'exécution du nœud n ou de l'occurrence m respectivement et ensuite n_{delay} et m_{delay} qui représentent leur durée d'exécution respectives.

La fonction de coût qui va permettre au solveur d'orienter sa résolution est modélisée par la contrainte 2.3 qui calcule la durée de l'exécution du programme en déterminant quelle occurrence est exécuté en dernier. Le but du solveur est évidemment ici de minimiser cette fonction de coût.

$$max(m_{1start} + m_{1delay}, ..., m_{Nstart} + m_{Ndelay}) \tag{2.3}$$

Pour que ces variables aient un sens il est nécessaire d'associer les valeurs des nœuds et celles de leurs occurrences associées. Les contraintes 2.4 et 2.5 définissent cette relation, par exemple la durée d'exécution d'un nœud correspond à la durée d'exécution de l'occurrence qu'il a sélectionné.

$$\forall n \in N,\ n_{start} = List_{start_n}[n_{match}]\ avec\ List_{start_n} = [m_{start} \mid m \in M \wedge n \in m] \tag{2.4}$$

$$\forall n \in N,\ n_{delay} = List_{delay_n}[n_{match}]\ avec\ List_{delay_n} = [m_{delay} \mid m \in M \wedge n \in m] \tag{2.5}$$

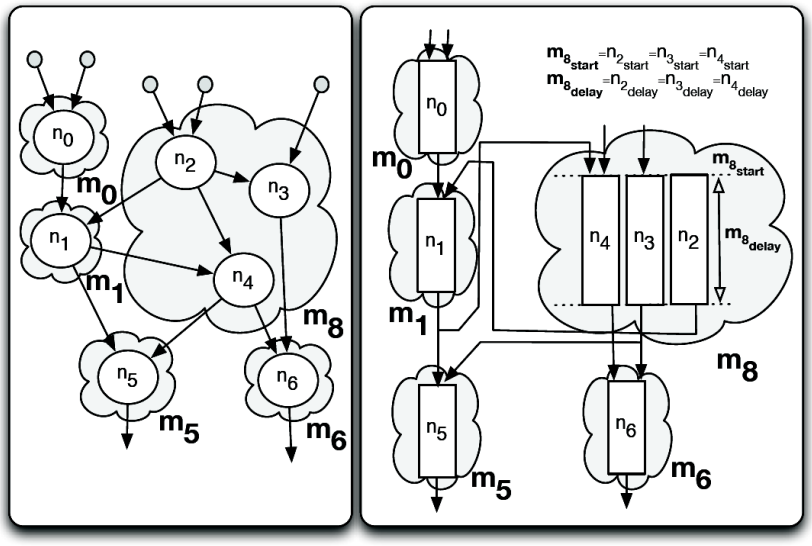


FIGURE 2.6: Exemple d'une occurrence non-convexe

Ces dernières contraintes imposent que toutes les entrées d'une même occurrence $m \in M$ soient disponibles au même moment m_{start} et que toutes ses sorties soient disponibles après la durée m_{delay} comme le montre le motif m_8 de la figure 2.6.

Enfin, il est indispensable d'ajouter la contrainte 2.6 qui garantit que les variables d'ordonnancement respectent le flot de données du programme. Cette contrainte est appliquée uniquement aux arcs qui ne sont pas contenus au sein d'une occurrence. Dans le cas où une occurrence est sélectionnée, ces arcs ne sont plus pris en compte par la contrainte.

$$\forall (s,d) \in E,\ if\ s_{match} \neq d_{match}\ then\ s_{start} + s_{delay} \leq d_{start} \tag{2.6}$$

Les occurrences identifiées précédemment peuvent être non-convexes, c'est-à-dire qu'il existe un chemin entre deux nœuds de l'occurrence qui passe par un nœud n'appartenant pas à cette occurrence. Cependant les contraintes temporelles 2.4, 2.5 et 2.6 n'autorisent pas ce cas de figure. Par exemple, si l'occurrence m_8 de la figure 2.6 est sélectionnée alors $m_{8start} = n_{4start} = n_{3start} = n_{2start} \wedge m_{1start} = n_{1start} \wedge n_{1start} + n_{1delay} \leq n_{2start} \wedge n_{4start} + n_{4delay} \leq n_{1start}$, ce qui est impossible car $n_{1delay} \neq 0$.

Chapitre 3

Exploration architecturale pour processeur VLIW

L'objectif des travaux menés dans le cadre de ce stage est de proposer une méthodologie et un outil permettant de définir les paramètres optimaux d'un processeur à instructions longues pour une application donnée. Avec cet outil, il sera alors possible de comparer les différentes structures de processeurs obtenues afin de choisir le meilleur compromis en termes de coût et de performance pour une application donnée. L'outil vise donc à permettre une exploration architecturale pour un modèle d'architecture de processeur s'appuyant sur le concept du VLIW. Dans sa version actuelle, notre modélisation est principalement axée sur des critères de performance ; cependant, il est tout à fait envisageable de l'orienter vers une considération telle que le coût matériel.

La recherche d'un processeur VLIW « idéal » pour une application donnée est un problème qui peut être résolu de deux manières : soit par la modification des paramètres architecturaux tels que le nombre d'unités fonctionnelles, leur organisation au sein de cluster ou encore le nombre de registres présents dans le processeur ; soit par l'extension du jeux d'instructions, c'est à dire l'ajout de nouvelles instructions dites instructions plus complexes (par opposition aux instructions classiques) et réalisant plusieurs opérations simultanément.

Des techniques existent déjà pour résoudre l'un ou l'autre des problèmes de manière plus ou moins efficace. Certaines, abordées dans la section 1.3, utilisent des algorithmes de *brute force*, génétiques ou encore utilisant la méthode de *branch-and-bound*. En revanche, aucune à notre connaissance ne permet encore de réaliser ces deux techniques simultanément pour obtenir un processeur encore plus performant.

Ces travaux se proposent d'utiliser la programmation par contraintes afin de les résoudre. En effet cette approche a déjà été utilisée avec succès dans l'équipe CAIRN pour résoudre le problème de l'extension de jeux d'instructions grâce à des techniques de recherche de motifs de calcul dans des graphes.

D'abord ce chapitre introduit le flot de compilation et le modèle de processeur à instructions longues sur lequel se base nos travaux. Il présente ensuite la modélisation de l'ordonnancement, c'est-à-dire les différentes contraintes issues du partage de ressources, des communications, de l'allocation de registres, etc. Enfin, il aborde les expérimentations et leurs résultats préliminaires obtenus.

3.1 Flot de compilation

Notre flot s'appuie sur GECOS (ou GEneric COmpiler Suite), l'infrastructure de compilation développée par l'équipe CAIRN comme base de travail à ses différents travaux. Le flot de compilation, illustré par la figure 3.1, est composée de plusieurs parties : d'abord l'interpréteur de code C transforme le code du programme vers une forme intermédiaire, le graphe de flot de données et de contrôle (CDFG) ; ensuite, quelques optimisations telles que la propagation de constantes ou le déroulage de boucles sont appliquées sur le graphe précédemment obtenu ; puis un générateur transforme le CDFG en HCDG. Cette représentation, décrite dans la section 2.1, est celle sur laquelle s'effectuera l'ensemble des traitements de la chaîne de compilation ; une fois le HCDG obtenu, la phase de génération des motifs est réalisée à partir de ce graphe et du modèle de l'architecture cible ; enfin, la dernière étape effectue la couverture de graphe à l'aide des motifs précédents dans le but de générer des instructions complexes, l'ordonnancement des instructions pour un processeur VLIW organisé en un ou plusieurs clusters ainsi que l'allocation de registres associée à cet ordonnancement.

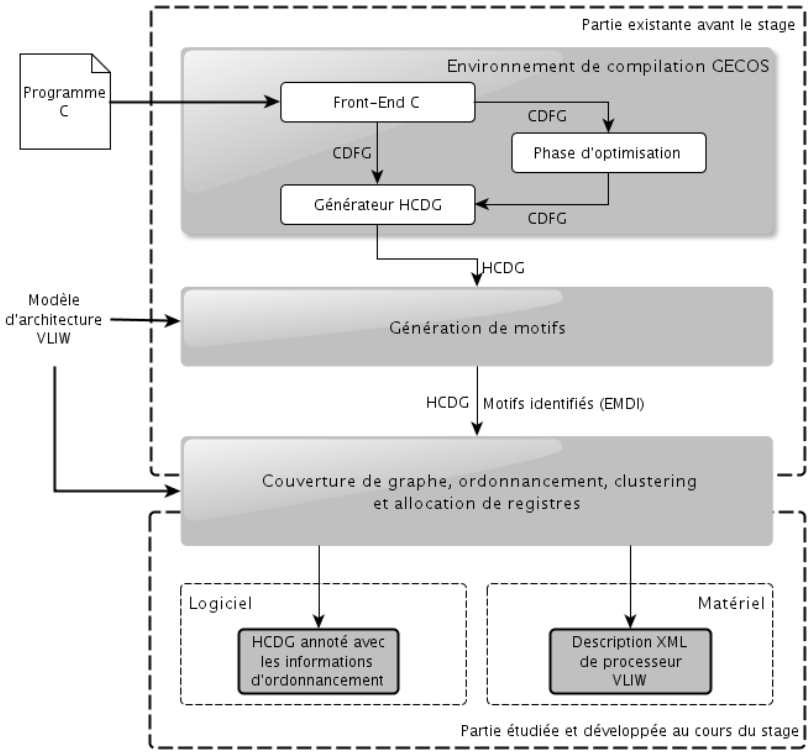


FIGURE 3.1: Flot de compilation

Au terme de la dernière étape, le compilateur fournit la description matérielle du processeur obtenu (nombre d'unités de calculs et de registres nécessaires, jeu d'instructions, organisation interne...) ainsi qu'un HCDG annoté par les informations calculées lors de l'ordonnancement et nécessaires à la génération d'un code exécutable.

3.2 Modèle de processeur VLIW

Dans cette section, nous allons définir le modèle d'architecture d'un processeur VLIW sur lequel notre outil réalisera l'exploration architecturale. Il sera ensuite spécifié à l'aide de

variables à domaines finis, sur lesquels seront exprimés des contraintes, dans le but de résoudre notre problème à l'aide de la programmation par contraintes.

3.2.1 Organisation architecturale

Ce modèle, illustré par la figure 3.2, est composé d'unités fonctionnelles réparties en deux catégories : les unités mémoires qui réalisent les interactions entre la mémoire de donnée et les registres ainsi que les unités logiques et arithmétiques (ALU ou *Arithmetic and Logical Unit*) qui exécutent le reste des opérations (multiplication, décalage, etc...). Ces dernières peuvent être spécialisées par l'utilisateur, dans ce cas elles ne réalisent que certaines opérations spécifiques. Par exemple il peut être intéressant d'utiliser des unités fonctionnelles qui réaliseraient uniquement les multiplications, en effet cette opération est particulièrement coûteuse d'un point de vue matériel et le fait de permettre à toutes les unités fonctionnelles de faire des multiplications peut augmenter considérablement la surface de silicium nécessaire à la réalisation du circuit.

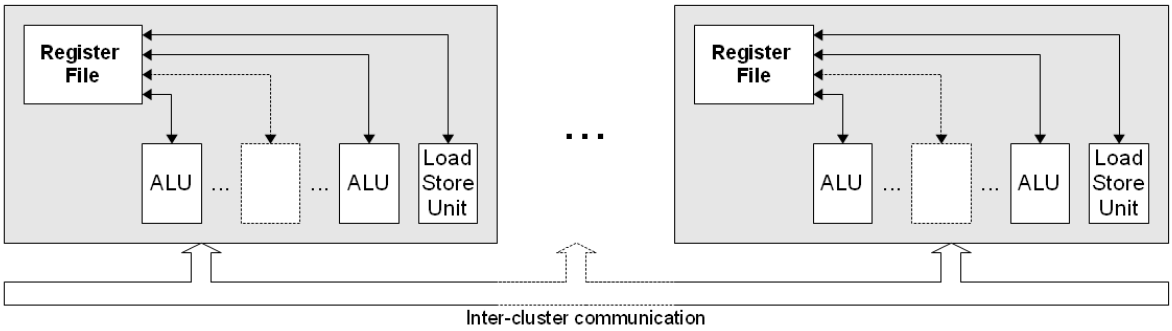


FIGURE 3.2: Modèle de processeur VLIW

Les unités fonctionnelles sont réparties dans plusieurs groupes distincts appelés clusters. Ces clusters sont tous strictement identiques, c'est-à-dire qu'ils contiennent le même nombre d'unités fonctionnelles et de registres. De plus une unité fonctionnelle ne peut appartenir à plusieurs clusters simultanément. Dans un soucis de logique, une unité mémoire est présente dans chaque cluster avec pour rôle de réaliser les interactions entre la mémoire du processeur et la file de registres spécifique au cluster.

Dans la plupart des cas, les processeurs à instructions longues utilisent des pipelines dans le but d'accélérer l'exécution des instructions. La profondeur du pipeline est un des éléments déterminant la vitesse d'exécution du processeur. Dans notre modèle, l'étage d'exécution est constitué de la phase de lecture des opérandes puis de la réalisation de l'opération en elle-même et enfin de l'écriture du résultat dans un registre. Celui-ci peut être cassé en plusieurs étapes afin d'équilibrer sa durée avec celle des autres étages tels que le *prefetch* ou le *decode*.

Ce pipeline est illustré par la figure 3.3 qui donne un exemple pour un processeur comprenant trois unités fonctionnelles. L'étage d'exécution est séparé en n parties distinctes, n étant défini selon la latence des opérations à réaliser. Les étapes de chargement et décodage des instructions se font par paquet, c'est-à-dire pour toutes les unités simultanément.

La contrepartie d'un pipeline plus complexe est l'ajout de circuiterie due au contrôle, et donc un coût matériel plus important. Cependant les opérations réalisées au sein d'un processeur VLIW peuvent avoir des latences très différentes les unes des autres. Par conséquent, la non-fragmentation de l'étage d'exécution entraîne un diminution importante des performances.

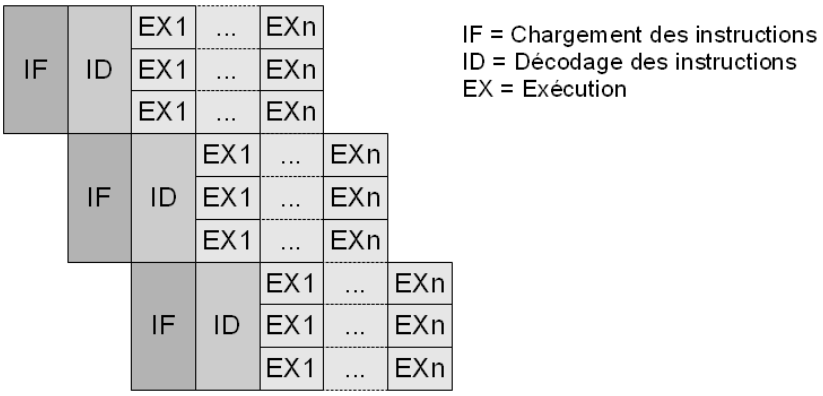


FIGURE 3.3: Pipeline utilisé

3.2.2 Communication

La communication entre les registres et les unités fonctionnelles est considérée comme étant non-limitante au sein d'un cluster. C'est-à-dire que les unités fonctionnelles sont connectées à tous les registres directement. La seule limite se situe au niveau du nombre d'accès simultanés à un même registre qu'il soit distant ou local, celui-ci est borné par la constante *MaxSimultaneousRegisterAccess*.

Par contre, les communications entre deux clusters distincts sont soumises à une restriction plus stricte. Le nombre de communications inter-clusters simultanées est limité, la constante *MaxInterClusterCommunications* définissant cette limite. De plus la mise-en-œuvre d'une telle communication nécessite un temps supplémentaire du au transfert de l'information. Cette latence est fixée par la constante *InterClusterCommunicationLatency*.

Lors de l'état de l'art sur les communications inter-cluster, deux types de communication ont été mises en évidence. Dans un premier temps, seules les techniques ne nécessitant pas l'ajout d'instructions supplémentaires seront modélisées. C'est pourquoi la latence de ces communications ne sera pas prise en compte lors de l'ordonnancement des opérations mais uniquement au moment de l'interprétation des résultats. Concrètement, si une telle communication est nécessaire avant l'exécution d'un instruction, le pipeline est gelé temporairement par une fonction matérielle jusqu'à ce que la donnée soit disponible.

3.2.3 Instruction spécialisée

Les processeurs basés sur le modèle VLIW utilisent des unités de calculs capables de réaliser plusieurs opérations différentes. C'est sur ce constat que l'idée de permettre aux différentes unités de calculs de réaliser des opérations plus complexe est apparue. En effet, comme il a été décrit dans la section 1.3.2, plusieurs avantages sont identifiables à l'utilisation de processeurs de type ASIP.

Cependant les instructions générées sont soumises à plusieurs contraintes architecturales afin de conserver un modèle viable et cohérent, les contraintes suivantes sont donc appliquées sur les motifs lors de leur génération : d'abord, le nombre d'entrées de tous les motifs est limité à 3 et le nombre de sortie à 1, dans le but de limiter le nombre de bits nécessaire pour coder les instructions et éviter une trop grande pression au niveau des registres ; enfin, la durée du chemin critique des motifs doit être inférieure à la latence maximum des opérations standards (qui sont généralement celles des interactions avec la mémoire) afin de ne pas augmenter la taille de l'étage d'exécution du pipeline.

3.3 Modélisation de l'ordonnancement

Comme il a été expliqué de nombreuses fois, l'objectif est de déterminer les paramètres optimaux d'un processeur VLIW pour une application donnée en termes de coût et de performance. Pour établir la performance, il est nécessaire de modéliser l'ordonnancement du programme sur le modèle de processeur introduit précédemment. Dans ce but, plusieurs critères sont à prendre en considération comme, par exemple, le partage de ressources et les contraintes de communications.

3.3.1 Flot de données

Les contraintes temporelles sont identiques à celles décrites dans la section 2.4, en résumé les contraintes qui expriment de flot de données et celles qui mettent en relation les variables des occurrences et celles de leurs nœuds.

La latence associée à chacun des nœuds, n_{delay}, est fixée initialement. Cette variable représente le temps nécessaire à l'opération modélisée par n pour être réalisée.

3.3.2 Partage de ressource

Lors de l'ordonnancement d'un programme pour un processeur de type VLIW, il est nécessaire d'assigner une ressource, ici une unité fonctionnelle, à chacune des opérations réalisées. Les variables $n_{resource}$ et $m_{resource}$ contiennent l'index de la ressource qui va exécuter le nœud ou l'occurrence respectivement.

La contrainte définie par l'équation 3.1 permet d'associer la variable de ressource d'un nœud $n \in N$ à la variable de ressource de l'occurrence sélectionnée qui contient n.

$$\forall n \in N,\ n_{resource} = List_{resource}[n_{match}]\ avec\ List_{resource} = [m_{resource} \mid m \in M \wedge n \in m] \tag{3.1}$$

Afin qu'une ressource ne soit pas allouée à deux opérations simultanées, il est nécessaire de modéliser les opérations et d'y imposer une contrainte. Ainsi, le temps d'exécution des différentes occurrences est modélisé par un rectangle à deux dimensions, dont la syntaxe est défini par $RecDiff2 = [x, y, \Delta x, \Delta y]$, et qui représente l'occupation d'une ressource par une opération au cours du temps. Si bien que chaque occurrence du graphe, qui correspond à une opération complexe ou non, est associée à un rectangle $RectOp = [m_{start}, m_{resource}, 1, m_{sel}]$. Ce rectangle représente plus précisément le premier étage d'exécution de l'opération en question. Enfin une contrainte de type *Diff*2 (équation 3.2) est alors appliquée à l'ensemble des rectangles *ListRectOp*. Cette contrainte permet d'interdire la superposition des rectangle dans l'espace caractérisé par le temps et les ressources du processeur, à savoir empêcher plusieurs opérations de s'exécuter simultanément sur la même ressource.

$$Diff2(ListRectOp) \tag{3.2}$$

La figure 3.4 illustre cette modélisation à l'aide de rectangles sur un exemple. Dans cette exemple, les opérations de type multiplication sont considérées avec une latence de deux cycles et tout les autres avec un unique cycle. Les rectangles pleins représentent les rectangles de la liste *ListRectOp* précédemment modélisés à l'aide de variables. Les rectangles hachurés sont juste ajoutés pour illustrer la durée totale de l'instruction et donc les dépendances de données. La contrainte empêche les rectangles pleins de se chevaucher les uns sur les autres. Commes il est possible de l'observer sur la figure, poser des contraintes sur le premier étage d'exécution et conserver les contraintes de flot de donnée classiques suffit à réaliser le partage des autres étages d'exécution.

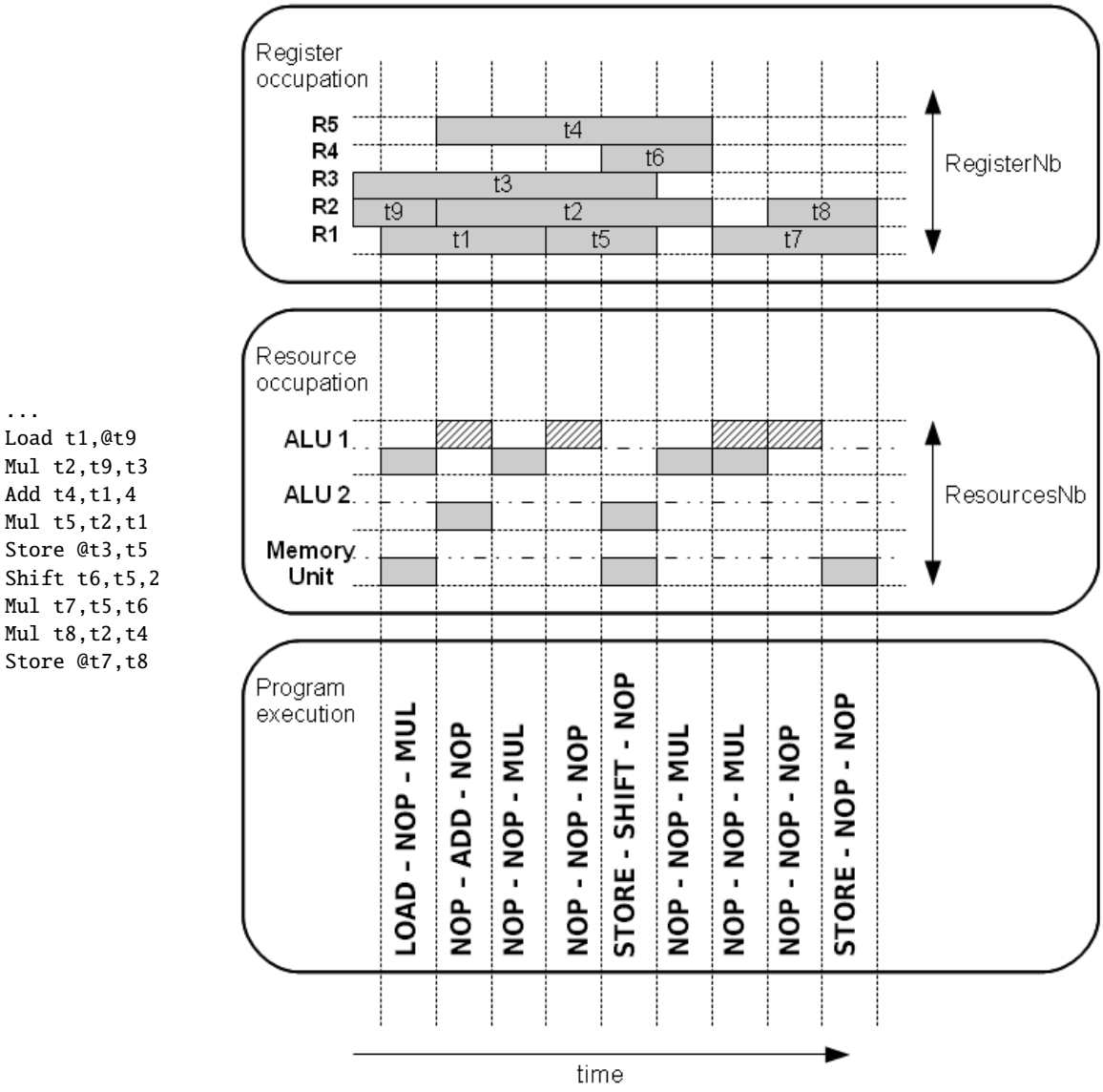


FIGURE 3.4: Exemple d'un morceau de programme. Illustration du partage de ressources et de l'allocation de registres appliqués à cet exemple

3.3.3 Clustering

Comme il a été expliqué précédemment, une unité fonctionnelle est associée à un unique cluster. Afin de le modéliser une table relationnelle entre les variables $n_{resource}$ et $n_{cluster}$ est utilisée, ces variables représentent respectivement l'index de l'unité fonctionnelle et du cluster où l'opération représentée par le nœud n sera exécutée. La contrainte définie par l'équation 3.3 illustre le cas où notre processeur contient deux clusters formés de deux unités fonctionnelles. Ainsi, sous l'hypothèse que le solveur choisit de fixer la variable $n_{resource}$ à 3 alors le domaine fini de la variable $n_{cluster}$ sera réduit à l'ensemble {2}. La table est initialisée au début de l'exécution selon les valeurs définies dans le modèle.

$$\forall n \in N, \quad \begin{array}{|c|c|} \hline n_{resource} & n_{cluster} \\ \hline 1 & 1 \\ 2 & 1 \\ 3 & 2 \\ 4 & 2 \\ \hline \end{array} \tag{3.3}$$

3.3.4 Communication

Selon le modèle, il est nécessaire de limiter le nombre d'accès simultanés à un même registre. Dans le graphe, ces accès sont représentés par les arcs sortant d'une occurrence. Et comme il a été défini précédemment dans la section 2.4, m_{start} représente le moment où l'instruction illustrée par m arrive à l'étage d'exécution du processeur. Or, les accès aux registres sont réalisés lors de la première étape de ce pipeline.

Ainsi, pour chaque occurrence $m \in M$, ces accès sont modélisés par des rectangles définis pour chaque arc $e_{(s,d)} \in E$ avec $s \in N \wedge s \in m \wedge d \in N \wedge d \notin m$ tels que $RectRead = [d_{start}, 1, m_{sel}]$. Chacune des occurrences est alors associée à une liste de rectangles $ListRectRead_m$ et la contrainte cumulative, définie par l'équation 3.4, est ajoutée pour chaque occurrence afin d'imposer le nombre maximum d'accès simultanés à chacun des registres.

$$\forall m \in M,\ Cumulative(ListRectRead_m, MaxSimultaneousRegisterAccess) \tag{3.4}$$

La figure 3.5 expose graphiquement la contrainte cumulative précédente. Chaque carré représente l'accès au registre courant et la ligne rouge impose la limite, dans notre cas elle correspond à la constante *MaxSimultaneousRegisterAccess* qui est égale à deux dans l'exemple.

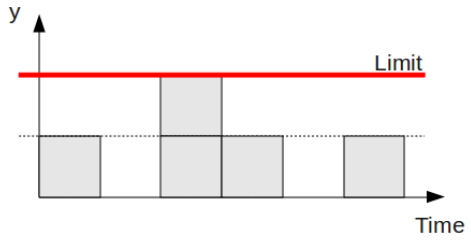


FIGURE 3.5: Illustration de la contrainte cumulative appliquée aux communications

Cependant, les communications entre clusters sont soumises à des contraintes supplémentaires. Aussi il est nécessaire de les identifier, c'est pourquoi la présence d'une communication entre deux clusters est caractérisée par la variable e_{comm} définie pour chaque arc du graphe. Cette variable est définie par la contrainte réifiée 3.5. Si la valeur de cette variable vaut 1 alors une communication inter-clusters est nécessaire, dans le cas contraire le résultat de l'opération précédente est déjà présent dans la file de registres du cluster courant : un accès direct est donc possible.

$$\forall e_{(s,d)} \in E,\ \ s_{cluster} \neq d_{cluster} \Leftrightarrow e_{comm} \tag{3.5}$$

Comme il a été décrit dans la section 1.2.2, la mise-en-œuvre matérielle de communication inter-clusters au sein du processeur n'est pas aisée . Elle est soumise à une contrainte de bande passante : il est donc nécessaire de limiter le nombre de communications simultanées de ce type.

La modélisation d'une limitation est réalisée par une contrainte *Cumulative*. Cette contrainte à une sémantique très proche de celle du *Diff*2 si ce n'est l'ajout d'une valeur limite, au sens quantitatif, à ne pas dépasser. Pour cette contrainte, la syntaxe des rectangles est légèrement différente : $RectCumulative = [x, \Delta x, \Delta y]$, la valeur de y n'est pas définie car le rectangle peut se placer à n'importe quel endroit sur cette axe.

Pour se faire, chacune de ces communications est modélisée à l'aide d'un rectangle défini par $\forall e_{(s,d)} \in E, RecComm = [d_{start}, 1, e_{comm}]$. L'ensemble de ces rectangles forment la liste *ListRecComm* utilisée par la contrainte définie par l'équation 3.6 qui limite ces communications.

$$Cumulative(ListRecComm, MaxInterClusterCommunications) \tag{3.6}$$

3.3.5 Allocation de registres

Toujours dans une logique de spécialisation de processeur, la réalisation de l'allocation de registres permet de déterminer le nombre de registres minimum nécessaire au processeur pour exécuter un programme sans ralentir son exécution.

Dans le but de la modéliser, les variables $n_{register}$ et $m_{register}$ sont ajoutées. Elles représentent l'index du registre au sein de sa banque de registres et dans lequel sera stocké le résultat de l'opération représentée respectivement par le nœud n et l'occurrence m. Ainsi, à l'issue de la résolution, la variable $m_{register}$ maximum correspondra au nombre de registres nécessaires par cluster.

Dans un premier temps et de la même manière que pour $n_{resource}$ et $m_{resource}$, la contrainte définie par l'équation 3.7 relie la valeur de la variable $m_{register}$ de l'occurrence m avec la valeur des variables $n_{register}$ pour toutes nœuds n qui composent m.

$$\forall n \in N,\ n_{register} = List_{register}[n_{match}]\ avec\ List_{register} = [m_{register} \mid m \in M \wedge n \in m] \tag{3.7}$$

La deuxième étape est la modélisation de la durée de vie de la variable temporaire qui contient le résultat d'une opération, c'est à dire la période entre la production de ce résultat et sa dernière utilisation. Dans ce but, quelques variables sont définies : m_{reg_start} qui correspond à la production du résultat, m_{reg_end} sa dernière utilisation, m_{reg_delay} la durée entre les deux et enfin m_{reg_number} qui représente l'index du registre au sein de l'ensemble du processeur (tous clusters confondus).

Dans notre modélisation, le résultat d'une opération est inscrit à la dernière étape de l'étage d'exécution, c'est pourquoi déterminer m_{reg_start} revient à calculer la fin de l'exécution de l'opération modélisée par m et d'y soustraire un comme l'illustre la contrainte 3.8.

$$\forall m \in M,\ m_{reg\_start} = (m_{start} + m_{delay}) - 1 \tag{3.8}$$

Inversement, les opérandes nécessaires à la réalisation d'une opération sont lus lors de la première étape de l'étage d'exécution. Pour calculer m_{reg_end}, il est nécessaire de déterminer l'arc sortant de m dont le nœud destination est le nœud exécuté le plus tard de toutes les possibilités. La contrainte définie par l'équation 3.9 permet le calcul de cette variable.

$$\begin{gathered}\forall m \in M,\ m_{reg\_end} = Max(List_{regUseStart}) + 1 \\ avec\ List_{regUseStart} = [d_{start} \mid s \in N \wedge d \in N \wedge e_{(s,d)} \in E \wedge n \in m]\end{gathered} \tag{3.9}$$

Ensuite la variable m_{reg_delay} est définie de manière triviale par la contrainte de l'équation 3.10.

$$\forall m \in M,\ m_{reg\_delay} = m_{reg\_end} - m_{reg\_start} \tag{3.10}$$

Enfin la contrainte 3.11 caractérise la variable m_{reg_number}, elle permet d'identifier de manière unique chacun des registres présents dans le processeur.

$$\forall m \in M,\ m_{reg\_number} = ClusterNb \times m_{register} + m_{cluster} \tag{3.11}$$

Un rectangle est alors associé à chaque occurrence, afin de représenter la vie de la variable contenant son résultat, défini par $RectReg = [m_{reg\_start}, m_{reg\_number}, m_{reg\_delay}, m_{sel}]$. Tous ces rectangles forment la liste ListRectReg sur laquelle est appliquée la contrainte 3.12 pour rendre mutuellement exclusive l'utilisation d'un registre par deux variables différentes.

$$diff2(ListRectReg) \tag{3.12}$$

La figure 3.4 illustre cette allocation de registre sous la forme d'un exemple. Comme il est possible de le remarquer, cinq registres sont nécessaires au minimum pour exécuter ce programme sans le ralentir.

3.3.6 Optimisations à partir des informations de contrôle

Un programme peut suivre différentes traces d'exécution selon les valeurs de ses variables. De plus certaines de ces traces peuvent être en exclusion mutuelle l'une avec l'autre, dans le cas d'une instruction de type `if condition then bloc1 else bloc2` par exemple. Ainsi, il est possible que certaines instructions ne s'exécutent jamais au cours de la même exécution.

A partir de ce constat, il est possible d'améliorer les différentes modélisations qui caractérisent notre problème. Pour se faire, nous utilisons les informations de contrôle présentes au sein des HCDG ainsi que le graphe d'exclusion mutuelle (*Mutual Exclusiveness Graph* ou MEG), défini en [8] et généré à partir du HCDG.

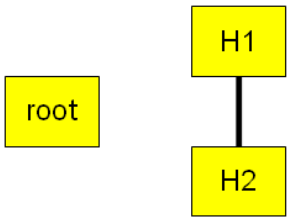


FIGURE 3.6: MEG généré à partir du HCDG de l'exemple de la figure 2.1

Un MEG est un graphe non dirigé dont la sémantique correspond au graphe de compatibilité utilisé dans les algorithmes de synthèse d'architecture de haut niveau. Chaque arc représente l'exclusion mutuelle des gardes représentées par les nœuds source et destination de l'arc. Cette représentation permet de tester facilement l'exclusivité de deux gardes. De plus, les groupes de gardes mutuellement exclusives correspondent aux cliques dans le MEG, c'est-à-dire aux ensembles de sommets deux-à-deux adjacents. La figure 3.6 donne un exemple trivial de MEG.

A partir de l'ensemble des cliques du MEG et du HCDG initial, il est possible de déterminer l'ensemble des exclusivités d'exécution entre les différentes occurrences. L'algorithme décrit par la figure 3.7 permet de calculer cet ensemble. Seules les cliques dont la taille est strictement supérieure à un sont conservées. Ensuite, l'ensemble des combinaisons de deux gardes au sein de chaque clique est traitée séparément : l'ensemble des occurrences dépendant de chacune des gardes est obtenue par la méthode $TrouverToutesLesOccurrencesDependantDe(HCDG, G)$, il suffit ensuite de créer des exceptions deux-à-deux entre les occurrences des deux ensembles.

Pour finir, il faut ajouter aux contraintes de type *Diff2* et *Cumulative* l'ensemble des exceptions obtenues.

```
// Entrée : HCDG - HCDG correspondant à l'application
// Entrée : EC - Ensemble des cliques d'exclusion mutuelle
// CLIQUE - Ensemble de gardes exclusives deux-à-deux
// G1,G2 - Gardes appartenant au HCDG
// EM1,EM2 - Ensembles d'occurrences
// M1,M2 - Occurrences
// Sorties : EE - Ensemble des exceptions entre les occurrences s'exécutant en
exclusion mutuelle
EE ← ∅
pour chaque CLIQUE ∈ EC
  if | CLIQUE | > 1
    pour chaque G1 ∈ CLIQUE
      CLIQUE ← CLIQUE \ G1
      EN1 ← TrouverToutesLesOccurrencesDependantDe(HCDG, G1)
      pour chaque G2 ∈ CLIQUE
        EN2 ← TrouverToutesLesOccurrenceDependantsDe(HCDG, G2)
        pour chaque M1 ∈ EM1
          pour chaque M2 ∈ EM2
            EE ← EE ∪ {creerException(M1, M2)}
          end pour
        end pour
      end pour
    end pour
  end if
end pour
```

FIGURE 3.7: Algorithme d'identification des exceptions

3.4 Expérimentations

Des expérimentations ont été réalisées dans le but de déterminer la qualité des différents structures de processeurs obtenues à l'aide de notre outil.

3.4.1 Contexte

Toutes les expérimentations ont été réalisées à partir du code d'applications, écrits en langage C, issus des suites de benchmarks *MediaBench* [11], *MiBench* [7], et *Cryptographic Library*.

La latence des différents opérateurs est définie par tableau 3.1. Ces valeurs sont issues de la documentation technique du processeur Texas Instruments TMS320C6X. Dans notre modélisation un cycle correspond à la durée d'exécution d'un étage du pipeline.

Les expérimentations sont réalisées d'après le protocole suivant : dans un premier temps la simulation est réalisée pour le mode séquentiel, c'est à dire que toutes les opérations (logiques, arithmétiques ou encore interactions avec la mémoire) sont exécutées sur la même unité fonctionnelle ; ensuite une seconde simulation est exécutée en suivant le modèle VLIW mais avec un unique cluster, ce qui permet de calculer le nombre maximum d'unités logiques et arithmétiques, c'est-à-dire le niveau de parallélisme du programme ; enfin des simulations sont exécutées en organisant le processeur en 2, 3 ou 4 clusters selon le nombre d'unités de calculs obtenues précédemment.

Opérateurs	Latence (en cycle)
Load	5
Store	4
Mul/Div	2
Arithmétiques (Add,Sub,etc...)	1
Shift	1
Logiques (And,Or,etc...)	1
Mov	1

TABLE 3.1: Latence des différents opérateurs

3.4.2 Résultats préliminaires

Les résultats obtenus pour l'ordonnancement sont résumés dans le tableau 3.2 pour l'ensemble des benchmarks en utilisant le mode séquentiel (avec et sans instructions complexes) et parallèle. Il indique le nombre de nœuds contenus dans le HCDG (sans ses informations de contrôle) généré à partir du code de l'application, le nombre de cycles nécessaires à l'exécution, l'accélération de ce temps d'exécution par rapport à la version séquentielle qui n'utilise pas d'instruction complexe, le nombre de motifs sélectionnés et la durée de résolution du problème. Pour la catégorie correspondant à la simulation d'un processeur contenant un seul cluster, le nombre maximum d'unités de calculs utilisées en parallèle est ajouté. Lorsque le nombre de cluster est supérieur à un, le nombre de cycles supplémentaires nécessaires à la communication entre les clusters, c'est-à-dire les moments où le pipeline est gelé, est aussi indiqué.

		Séquentiel					VLIW																	
Nombre de clusters		-					1					2						3/4						
Applications	Nœuds	Cycles (sans motif)	Sélectionnés	Cycles	Accélération	Résolution (en s)	Sélectionnés	Cycles	Accélération	ALU	Résolution (en s)	Sélectionnés	Cycles	Gels du pipeline	Total	Accélération	Résolution (en s)	Clusters	Sélectionnés	Cycles	Gels du pipeline	Total	Accélération	Résolution (en s)
JPEG IDCT col	120	132	9	120	1,1	12,2	9	59	2,24	6	19,9	9	56	25	81	1,63	1,8	3	9	55	28	83	1,59	1,9
JPEG IDCT row	101	112	9	106	1,06	8,3	9	53	2,11	6	2	9	49	19	68	1,64	1,2	3	9	48	19	67	1,67	1
POLARSSL des	51	51	12	48	1,06	1	11	33	1,55	4	0,7	11	33	9	42	1,21	0,7	4	11	33	18	51	1	0,4
MCRYPT Cast128	475	546	26	484	1,12	562	32	358	1,52	6	179	28	299	113	412	1,33	174	3	28	287	108	395	1,38	115
MESA invert matrix	301	374	10	361	1,04	499	10	223	1,67	9	77	-	-	-	-	-	-	3	10	151	67	218	1,72	50,9

TABLE 3.2: Résultats obtenus pour l'ordonnancement

L'utilisation des registres par les différentes applications est résumée dans le tableau 3.3. Ce dernier indique le nombre de registres utilisés lors d'un ordonnancement séquentiel puis avec le modèle de processeur VLIW, l'utilisation par cluster y est détaillé.

La tableau 3.4 résume les caractéristiques principales des différentes applications utilisés telles que le niveau de parallélisme ou encore la quantité d'accès mémoires.

Parallélisme

Les résultats justifient l'intérêt des processeurs VLIW sur des applications ayant un ILP important. Même si le gain n'est pas énorme pour certains benchmarks, ces programmes sont généralement exécutés de nombreuses fois au sein d'une application complète : un léger gain peut alors devenir non négligeable sur le temps d'exécution global.

	Séquentiel	VLIW						
Nombre de clusters	-	1	2		3		4	
	Nombre de registres utlisés							
Applications			par cluster	total	par cluster	total	par cluster	total
JPEG IDCT col	31	30	22	44	28	83	-	-
JPEG IDCT row	30	30	21	42	12	36	-	-
POLARSSL des	15	15	10	20	-	-	7	28
MCRYPT Cast128	86	79	54	108	36	108	-	-
MESA invert matrix	117	83	-	-	42	126	-	-

TABLE 3.3: Résultats obtenus pour l'allocation de registres

Applications	**Régularité**	**Parallélisme**	**Chemin critique**	**Connectivité**	**Accès mémoire**
JPEG IDCT col	faible	important	court	importante	moyen
JPEG IDCT row	faible	important	court	importante	moyen
POLARSSL des	faible	moyen	moyen	moyenne	faible
MCRYPT Cast128	moyenne	moyen	long	moyenne	important
MESA invert matrix	forte	important	moyen	moyenne	important

TABLE 3.4: Type d'applications

Clustering

Si on ne tient pas compte de la latence imposée par les communications entre clusters, les résultats expose une accélération de l'exécution lorsque le processeur est réparti en plusieurs clusters. Ce résultat est entièrement dû à notre modélisation : en effet celle-ci instaure qu'une unité mémoire soit présente dans chacun des clusters, de sorte que les interactions avec la mémoire en sont accélérées.

Le tableau montre que légèrement plus de registres sont nécessaires lorsque le processeur est organisés en plusieurs clusters. Cette augmentation est principalement due à l'augmentation de la complexité : le solveur résout alors moins efficacement le problème.

Communication inter-clusters

Les résultats montrent que les latences engendrées par les communications entre clusters diminuent fortement les performances du processeur pour chacune des applications dès qu'il y a plusieurs clusters. La principale raison à ce problème est que ces communications ne sont pas prises en considération par la fonction de coût qui oriente la résolution du problème.

Actuellement, notre modélisation est celle choisit par le processeur TMS320C6X, à savoir que le pipeline du processeur est entière gelé dès qu'une opération nécessite une telle communication.

Une solution évidente serait d'ajouter le terme $\sum_{e \in E} e_{comm} \times InterClustersCommunicationLatency$ à la fonction de coût, cependant cette modélisation est incorrecte car elle ne considère pas que deux communications inter-clusters peuvent être réalisées au cours du même cycle et donc que le pipeline ne sera gelé qu'une seul fois.

Aucune solution est entrevue pour ajuster la fonction de coût avec notre modélisation actuelle. Cependant la modélisation de ces communications peut être modifiée. Ainsi, deux alternatives sont envisagées :

– Au lieu de réaliser des opérations dont les opérandes peuvent provenir d'une banque de registres distante sous la forme `op1 r1[1],r2→r3`, le résultat de l'opération serait di-

rectement envoyé vers un registre extérieur en utilisant une instruction de la forme `op1 r1,r2→r3[2]`. En [18], les auteurs nomment ce type de modélisation *extended results*. Il suffirait alors d'ajouter une variable de délai sur les arcs qui serait instanciée à 1 dans le cas d'une communication inter-clusters et 0 sinon puis de modifier la modélisation du flot de donnée pour prendre en compte ce délai.

– La deuxième possibilité consisterait à geler uniquement le pipeline de l'unité fonctionnelle incriminée. Au niveau de notre modélisation, il faudrait définir le délai de la manière suivante : $m_{delay} = \delta in_m + \delta_m$ et faire varier δin_m entre la latence des communications inter-clusters et 0 selon la présence ou non d'une telle communication.

Ces deux alternatives sont encore à l'étude pour déterminer laquelle est la plus satisfaisante. Il sera alors nécessaire d'implémenter la solution choisie afin de comparer les résultats obtenus pour déterminer si les performances sont réellement améliorées.

Conclusion

Actuellement, les processeurs basés sur une architecture VLIW sont principalement utilisés dans le cadre des systèmes embarqués. En effet, ces processeurs sont une des solutions possibles aux besoins d'accroissement des performances et de réduction de la taille des puces, liés à la démocratisation des systèmes multimédias. Dans cette optique, le problème de spécialisation de processeur est la solution permettant d'obtenir le ratio performance/dimensionnement le plus intéressant.

Dans ce rapport, nous avons présenté une nouvelle approche de l'exploration architecturale d'un processeur à instructions longues pour une application donnée, principalement basée sur la programmation par contraintes. Contrairement aux techniques précédemment étudiées, elle permet de déterminer les paramètres optimaux de l'architecture : aussi bien en termes d'organisation et de dimensionnement que dans la définition d'un jeu d'instructions spécialisés. De plus, notre outil fournit toutes les informations résultant de l'ordonnancement et de l'allocation de registres pour la production du code exécutable.

L'avantage irrémédiable de notre méthode par rapport aux précédentes est la concentration des efforts non pas sur le résolution du problème, mais sur sa modélisation. En effet, les techniques de résolution sont implémentées de manière transparente au sein du solveur. Elles sont développées parallèlement à notre outil et profitent des dernières évolutions dans le domaine.

Afin de modéliser notre problématique, des nouveautés ont été apportés aux précédents travaux de l'équipe sur la couverture de graphe : l'utilisation des informations de contrôle dans le but d'affiner les différentes contraintes, ainsi que la résolution de l'allocation de registres au cours de la couverture.

Plusieurs améliorations peuvent être apportées à cet outil : dans un premier temps, la gestion des boucles avec, par exemple, l'implémentation d'un algorithme réalisant un pipeline logiciel à l'aide de la programmation par contraintes ; ensuite, l'ajout au deuxième type de communications inter-clusters au modèle ; établir plus nettement des contraintes prenant en compte le coût matériel, et ainsi permettre une résolution en termes de coût et non pas de performance.

De plus, il est important de préciser que les travaux décrits dans le dernier chapitre de ce rapport ont été implémentés à l'aide d'un développement logiciel. Les résultats préliminaires obtenus sont encourageant et permettent une première validation du raisonnement.

Bibliographie

[1] Giuseppe Ascia, Vincenzo Catania, Alessandro G. Di Nuovo, Maurizio Palesi, and Davide Patti. Efficient design space exploration for application specific systems-on-a-chip. *J. Syst. Archit.*, 53(10) :733–750, 2007.

[2] Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher, Giuseppe Desoli, and Fred Homewood. Lx : A Technology Platform for Customizable VLIW Embedded Processing. In *ISCA '00 : Proceedings of the 27th annual international symposium on Computer architecture*, pages 203–213, New York, NY, USA, 2000. ACM.

[3] Joseph A. Fisher. Trace Scheduling : A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, 30(7) :478–490, 1981.

[4] Joseph A. Fisher. Very Long Instruction Word architectures and the ELI-512. In *ISCA '83 : Proceedings of the 10th annual international symposium on Computer architecture*, pages 140–150, New York, NY, USA, 1983. ACM.

[5] Antoine Floch, Christophe Wolinski, and Krzysztof Kuchcinski. Combined Scheduling and Instruction Selection for Processors with Reconfigurable Cell Fabric. In *21th IEEE International Conference on Application-specific Systems, Architectures and Processors, (ASAP 2010)*, Rennes France, July 2010. IEEE.

[6] J. P. Grossman. Compiler and Architectural Techniques for Improving the Effectiveness of VLIW Compilation. 2007.

[7] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench : A free, commercially representative embedded benchmark suite. In *WWC '01 : Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[8] Apostolos A. Kountouris and Christophe Wolinski. Hierarchical Conditional Dependency Graphs for Conditional Resource Sharing. In *EUROMICRO '98 : Proceedings of the 24th Conference on EUROMICRO*, page 10313, Washington, DC, USA, 1998. IEEE Computer Society.

[9] Krzysztof Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Trans. Des. Autom. Electron. Syst.*, 8(3) :355–383, 2003.

[10] Krzysztof Kuchcinski and Christophe Wolinski. Global approach to assignment and scheduling of complex behaviors based on hcdg and constraint programming. *J. Syst. Archit.*, 49(12-15) :489–503, 2003.

[11] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench : a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO 30 : Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society.

[12] Yung-Chia Lin, Chia Han Lu, Chung-Ju Wu, Chung-Lin Tang, Yi-Ping You, Ya-Chaio Moo, and Jenq-Kuen Lee. Effective Code Generation for Distributed and Ping-Pong Register Files : A Case Study on PAC VLIW DSP Cores. *J. Signal Process. Syst.*, 51(3) :269–288, 2008.

[13] Y.-S. Lu, L. Shen, L.-B. Huang, Z.-Y. Wang, and N. Xiao. Optimal subgraph covering for customisable VLIW processors. *Computers Digital Techniques, IET*, 3(1) :14 –23, january 2009.

[14] Kevin Martin, Christophe Wolinski, Krzysztof Kuchcinski, Antoine Floch, and François Charot. Constraint-Driven Identification of Application Specific Instructions in the DURASE System. In *SAMOS '09 : Proceedings of the 9th International Workshop on Embedded Computer Systems : Architectures, Modeling, and Simulation*, pages 194–203, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] Bhuvan Middha, Anup Gangwar, Anshul Kumar, M. Balakrishnan, and Paolo Ienne. A Trimaran Based Framework for Exploring the Design Space of VLIW ASIPs with Coarse Grain Functional Units. In *ISSS '02 : Proceedings of the 15th international symposium on System Synthesis*, pages 2–7, New York, NY, USA, 2002. ACM.

[16] T. Miyamori. Venezia : a scalable multicore subsystem for multimedia applications. In *Proceedings of the 8th International Forum on Application-Specific Multi-Processor SoC*, 2008.

[17] Debyo Saptono, Vincent Brost, Fan Yang, and Eri Prasetyo. Design Space Exploration for a Custom VLIW Architecture : Direct Photo Printer Hardware Setting Using VEX Compiler. In *SITIS '08 : Proceedings of the 2008 IEEE International Conference on Signal Image Technology and Internet Based Systems*, pages 416–421, Washington, DC, USA, 2008. IEEE Computer Society.

[18] A. S. Terechko and H. Corporaal. Inter-cluster communication in VLIW architectures. *ACM Trans. Archit. Code Optim.*, 4(2) :11, 2007.

[19] Jan-Willem van de Waerdt, Stamatis Vassiliadis, Sanjeev Das, Sebastian Mirolo, Chris Yen, Bill Zhong, Carlos Basto, Jean-Paul van Itegem, Dinesh Amirtharaj, Kulbhushan Kalra, Pedro Rodriguez, and Hans van Antwerpen. The TM3270 Media-Processor. In *MICRO 38 : Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 331–342, Washington, DC, USA, 2005. IEEE Computer Society.