Sélection automatique d'instructions et ordonnancement d'applications basés sur la programmation par contraintes

Kevin Martin, Christophe Wolinski, Krzysztof Kuchcinski, Antoine Floch, François Charot

RenPar'19 / SympA'13 / CFSE'7
Toulouse, France, du 9 au 11 septembre 2009

Sélection automatique d'instructions et ordonnancement d'applications basés sur la programmation par contraintes

Kevin Martin[1], Christophe Wolinski[1,2], Krzysztof Kuchcinski[3], Antoine Floch[1], François Charot[1]

[1]INRIA Rennes - Bretagne Atlantique, France
[2]Université de Rennes I, Irisa, France
[3]Dept. of Computer Science, Lund University, Sweden

Résumé
Ce papier présente une nouvelle méthode, basée sur la programmation par contraintes, pour la sélection de motifs de calcul, le placement et l'ordonnancement d'applications sur des extensions de processeurs configurables. Cette méthode est intégrée dans l'environnement DURASE (*Generic Environment for Design and Utilization of Reconfigurable Application-Specific Processors Extensions*). Les extensions du processeur, qui mettent en œuvre les motifs de calcul et qui sont accessibles via des instructions spécialisées, sont fortement couplées au chemin de données du processeur. Ces instructions spécialisées sont générées et sélectionnées à partir du graphe de l'application. Notre méthode supporte un ordonnancement sous contrainte de ressources ou sous contrainte de temps. Les résultats expérimentaux obtenus sur les benchmarks *MediaBench* et *MiBench* montrent une accélération de l'exécution des applications d'un facteur de 2,3 en moyenne.

Mots-clés : ASIP, extension de jeu d'instructions, Altera NIOSII, couverture de graphe, programmation par contraintes

1. Introduction

Les processeurs de type ASIP (*Application Specific Instruction-set Processor*) constituent un excellent compromis car ils offrent la flexibilité d'un processeur à usage général et la performance d'un matériel dédié pour faire face à la contrainte de temps de mise sur le marché. Cependant, la spécialisation de tels processeurs nécessite encore des efforts humains importants et l'automatisation de ce processus à partir de spécifications de haut niveau est un véritable enjeu. Notre environnement DURASE [16] permet l'extension automatique du jeu d'instructions de processeurs RISC par la génération d'instructions spécialisées. Ces instructions spécialisées sont modélisées dans notre environnement par des motifs de calcul. Un motif de calcul est un ensemble d'instructions qui sera remplacé par une instruction spécialisée mise en œuvre matériellement dans une extension. Cette extension est fortement couplée au chemin de données du processeur. L'environnement permet également de répercuter les modifications sur le code source pour la prise en compte des instructions nouvellement créées. La chaîne de compilation adoptée dans DURASE est illustrée par la figure 1. L'environnement prend en entrée le code source de l'application écrit en langage C, le jeu d'instructions du processeur et le modèle d'architecture. Il génère en sortie une extension pour le processeur cible et le code source transformé qui exploite cette extension. L'extension du processeur est construite autour d'un motif fusionné qui met en œuvre tous les motifs sélectionnés.

Notre flot de conception consiste en la génération de motifs puis la sélection des motifs spécifiques qui accélèrent l'exécution de l'application. La génération et la sélection de motifs sont réalisées en deux phases. Dans la première phase, nous générons les motifs candidats et identifions les plus utiles pour une application donnée. La deuxième phase est la phase de sélection de motifs et d'ordonnancement qui conserve un sous-ensemble des motifs candidats. Dans ce papier, nous présentons notre méthode pour la sélection de motifs et l'ordonnancement d'applications basées sur la programmation par contraintes.

L'environnement DURASE utilise des technologies avancées comme les algorithmes d'isomorphisme de (sous-)graphe et de fusion de chemin de données (récemment développés dans notre équipe) ainsi que les méthodes de programmation par contraintes. DURASE utilise notre plate-forme générique de

compilation Gecos [5] récemment étendue pour supporter des transformations polyédriques. La représentation interne de l'environnement DURASE est le Graphe Hiérarchisé aux Dépendances Conditionnées (GHDC), qui permet de réunir le flot de contrôle et le flot de donnée de l'application au sein de la même représentation. Cette représentation a déjà été utilisée auparavant pour la synthèse de haut niveau [11]. Cette représentation supporte des transformations formelles de graphe, et contient les informations concernant l'exclusivité mutuelle, utile pour le partage de ressource. Les boucles peuvent être partiellement ou complètement déroulées. Le GHDC est utilisé en entrée du générateur de motifs.

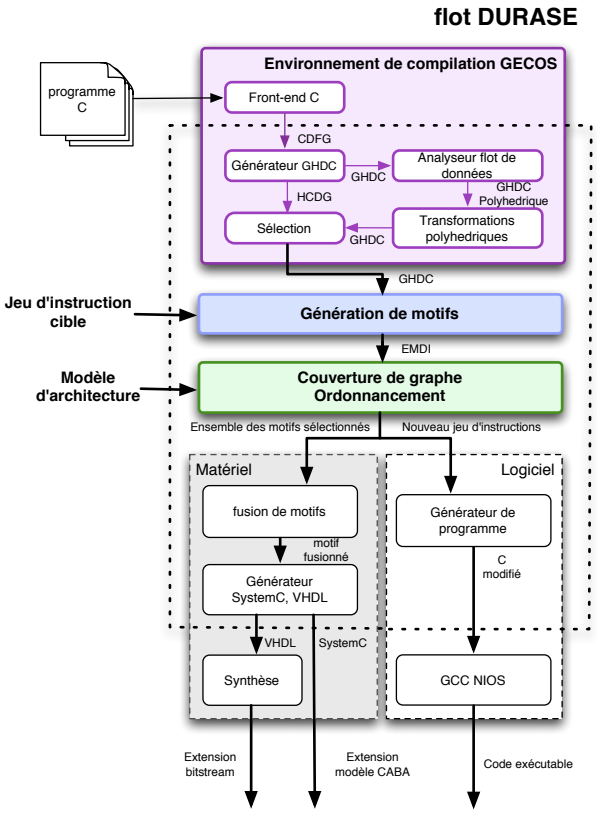


FIG. 1 – Chaîne de compilation générique pour la génération matérielle et logicielle d'extensions de processeur

Dans ce papier, nous considérons le modèle d'architecture d'un processeur ASIP avec un jeu d'instructions étendu. Les instructions étendues mettent en œuvre les motifs de calcul identifiés et sélectionnés et peuvent être exécutées séquentiellement aux instructions de base du processeur. La figure 2 présente l'exemple de l'architecture d'un processeur NIOSII d'Altera. L'architecture cible est composée d'une cellule fonctionnelle reconfigurable qui met en œuvre les motifs sélectionnés. Ces motifs sélectionnés sont fusionnés par notre procédure de fusion de chemins de données [24]. La cellule possède également des registres pour le cas où le nombre d'entrées et sorties des motifs générés dépassent les limitations imposées par l'architecture du processeur (par exemple, 2 entrées et 1 sortie pour le cas du NIOSII). Le nombre de registres et la structure d'interconnexion sont dépendants de l'application.

Le papier est organisé de la façon suivante. La section 2 concerne la programmation par contraintes utilisée dans notre approche. La génération de motifs est présentée section 3. La section 4 décrit la modélisation pour la sélection de motifs et l'ordonnancement. La section 5 donne des résultats expérimentaux. La section 6 présente les travaux en rapport avec notre problématique. Enfin, la conclusion en section 7.

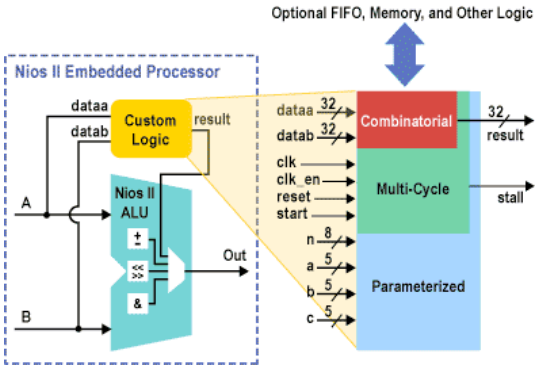


FIG. 2 – Extension du jeu d'instructions pour un NIOSII

2. La programmation par contraintes

Nos travaux reposent sur l'utilisation de méthodes de satisfaction de contraintes mises en œuvre dans le solveur de contraintes JaCoP [9].

Un *problème de satisfaction de contraintes* (ou *CSP* pour *Constraint Satisfaction Problem*) est défini par un 3-tuple $\mathcal{S} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ où $\mathcal{V} = \{x_1, x_2, \ldots, x_n\}$ est un *ensemble de variables*, $\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$ un ensemble de *domaines finis* (FD pour *Finite Domains*) de ces variables, et $(\mathcal{C})$ un ensemble de *contraintes*. Les variables à domaines finis (FDV pour *Finite Domain Variables*) sont définies par leur domaines, c'est-à-dire les valeurs que peuvent prendre ces variables. Un domaine fini est couramment exprimé à l'aide d'entiers, par exemple $x :: 1..7$. Une contrainte $c(x_1, x_2, \ldots, x_n) \in \mathcal{C}$ sur des variables de $\mathcal{V}$ est un sous-ensemble de $D_1 \times D_2 \times \ldots \times D_n$ qui restreint chaque combinaisons de valeurs que les variables peuvent prendre simultanément. Des équations, des inégalités et même des programmes peuvent définir des contraintes.

Nous utilisons en particulier une contrainte de correspondance de graphes appelée `GraphMatch`. Cette contrainte définit les conditions pour l'isomorphisme de (sous-)graphe entre un graphe cible et un motif (un motif peut être défini par un ensemble de sous-graphes séparés). Cette contrainte a été implémentée par un algorithme d'élagage développé pour cet objectif précis et spécialement implémentée dans notre système UPaK.

Une *solution à un CSP* est une affectation à chaque variable d'une valeur définie dans son domaine, de telle manière que toutes les contraintes soient satisfaites. Le problème spécifique à modéliser va déterminer si nous avons besoin de *juste une solution*, *toutes les solutions* ou une *solution optimale* étant donnée une fonction de coût définie par le biais des variables.

Le solveur est construit en utilisant des méthodes propres de consistance des contraintes et des procédures de recherche systématique. Les méthodes de consistance essaient 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. La plupart des techniques de consistance ne sont pas complètes et pour trouver une solution le solveur a besoin d'explorer les domaines restants par une recherche.

Les solutions à un CSP sont souvent trouvées par une affectation systématique à une variable de chaque valeur de son domaine. Ceci consiste en un parcours en profondeur (*depth-first-search*). La méthode de consistance est appelée dès que les domaines des variables pour une contrainte donnée sont réduits. Si une solution partielle enfreint n'importe laquelle des contraintes, le *backtracking* se met en place, réduisant ainsi la taille de l'espace de recherche restant.

3. La génération de motifs

La génération de motifs est appliquée à un graphe d'application acyclique $G = (N, E)$ où N est un ensemble de nœuds et E un ensemble de liens. Un motif est un sous-graphe $P = (N_p, E_p)$ du graphe G

où $N_p \subseteq N$ et $E_p \subseteq E$. Le motif P est également sous-graphe isomorphe au graphe G. Cet isomorphisme de sous-graphe est trouvé en définissant un ensemble de contraintes et en trouvant des solutions aux contraintes.

```
// Entrées: G(N,E)-- graphe d'application,
//          N-- ensemble des noeuds,
//          E-- ensemble des liens,
//          EMDI-- Ensemble des Motifs Définitivement
//                 Identifiés,
//          EMC-- Ensemble des Motifs Courants,
//          EMT-- Ensemble des Motifs Temporaires
//          n_s-- noeud ''seed'' du motif
EMDI ← ∅
pour chaque n_s ∈ N
  EMT ← ∅
  EMC ← TrouverTousLesMotifs(G, n_s)
  pour chaque p ∈ EMC
      si ∀_{pattern∈EMT} p ≢ pattern
          EMT ← EMT ∪ {p},
          NMP_p ← TrouverToutesLesOccurrences(G, p)
  NMP_{n_s} ← TrouverToutesLesOccurrences(G, n_s)
  pour chaque p ∈ EMT
      si coef · NMP_{n_s} ≤ NMP_p
          EMDI ← EMDI ∪ {p}
return EMDI
```

FIG. 3 – Algorithme d'identification des motifs

```
// Entrées: G=(N,E)-- graphe d'application,
//          EMDI-- Ensemble des Motifs Définitivement
//                 Identifiés
//          M_p-- ensemble des occurrences pour un
//                 motif p,
//          M-- ensemble de toutes les occurrences,
//          matches_n-- ensemble des occurrences qui
//                 peuvent couvrir un noeud n,

M ← ∅
pour chaque p ∈ EMDI
  M_p ← TrouverToutesLesOccurrences(G, p)
  M ← M ∪ M_p
pour chaque m ∈ M
  pour chaque n ∈ m
    matches_n ← matches_n ∪ {m}
```

FIG. 4 – Algorithme trouvant toutes les occurrences qui couvrent un nœud dans un graphe

L'algorithme de génération de motifs est illustré par la figure 3. Dans la première étape, l'algorithme identifie tous les motifs de calcul formés autour de chaque nœud appelé "seed node" $n_s \in N$ satisfaisant toutes les contraintes architecturales et technologiques. Il est également possible d'identifier les motifs pour seulement quelques seed nodes bien choisis selon des heuristiques mais cette approche n'est pas considérée ici. L'identification de tous les motifs est réalisée par la méthode *TrouverTousLesMotifs*(G, n) qui est implémentée en utilisant la programmation par contraintes. Dans l'étape suivante, l'ensemble des motifs temporaires (EMT) est successivement élargi par les motifs non-isomorphes venant de l'ensemble des motifs courants (EMC). Finalement, les motifs ajoutés à l'ensemble des motifs définitivement identifiés (EMDI) sont les motifs dont le nombre d'occurrences dans le graphe de l'application est suffisamment grand comparé au nombre d'occurrences du seed node seul (modulo un coefficient de filtrage *coef* où $0 \leq coef \leq 1$). Le nombre d'occurrences d'un motif donné dans le graphe de l'application est aussi obtenu en utilisant les méthodes de programmation par contraintes implémentées par la fonction *TrouverToutesLesOccurrences*(G, n). La contrainte `GraphMatch` évoquée dans le paragraphe 2 est utilisée. La formulation complète de cette phase de génération de motifs est décrite dans [15].

4. La sélection de motifs et l'ordonnancement

La sélection de motifs de calcul et l'ordonnancement de l'application sont réalisés à partir du graphe d'application G et de l'ensemble des motifs définitivement identifiés (EMDI) produit par la phase de génération de motifs. Pour modéliser les problèmes de sélection de motifs et d'ordonnancement, il est nécessaire de savoir quel nœud du motif peut couvrir quel nœud du graphe. Cette information est obtenue par la procédure de la figure 4. Après exécution de cette procédure, chaque nœud $n \in N$ possède un ensemble d'occurrences ($matches_n$) qui peuvent le couvrir.
Évidemment, dans la couverture finale du graphe G, chaque nœud $n \in N$ ne peut être couvert que par une et une seule occurrence. Par exemple, la figure 5 montre les occurrences possibles et sélectionnées pour l'exemple de la figure 6. Les étoiles grises représentent les occurrences possibles alors que les étoiles noires indiquent les occurrences sélectionnées. Nous modélisons la sélection d'une occurrence donnée dans un ordonnancement final en utilisant une variable à domaine fini m_{sel} associée à chaque occurrence $m \in M$, où M est l'ensemble des occurrences. La valeur de la variable m_{sel} vaut 1 si l'occurrence m est sélectionnée, 0 sinon. La contrainte (1) impose que chaque nœud ne peut être couvert que par une occurrence.

$$\forall n \in N : \sum_{m \in matches_n} m_{sel} = 1 \quad (1)$$

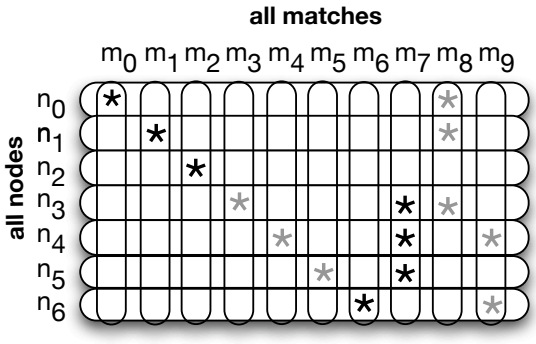


FIG. 5 – Occurrences identifiées et sélectionnées pour l'exemple de la figure 6.

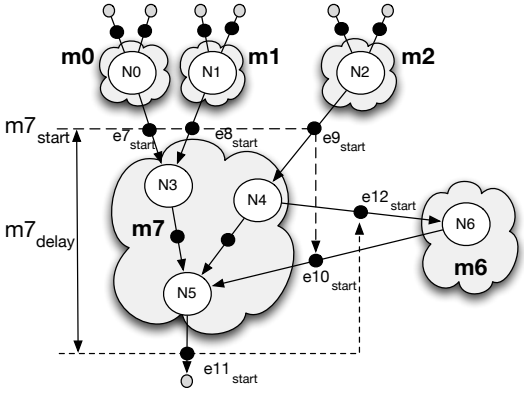


FIG. 6 – Exemple d'une occurrence non-convexe

4.1. La sélection de motifs

Dans ce papier, nous nous intéressons dans un premier temps à la sélection de motifs en vue de l'optimisation de l'ordonnancement séquentiel d'une application sur un processeur donné. Cet objectif peut être obtenu en minimisant la fonction de coût définie par l'équation (2) où le délai introduit par l'exécution d'une occurrence (m_{delay}) dépend du modèle de l'architecture et est présenté en détail dans la section 4.2. Dans ce cas précis, le problème de la sélection de motifs correspond au problème de couverture de graphe où tous les nœuds doivent être couverts et la somme des délais des occurrences qui couvrent ce graphe doit être minimale. Une fois les occurrences sélectionnées, l'ordonnancement peut être réalisé en utilisant par exemple un algorithme d'ordonnancement par liste.

$$\text{ExecutionTime} = \sum_{m \in M} m_{sel} \cdot m_{delay} \tag{2}$$

Nous pourrions également sélectionner les motifs de calcul pour un ordonnancement sous contraintes de ressources. Dans ce cas, $ExecutionTime$ est limité par une constante $ExecutionTime_{max}$ et le nombre de motifs différents sélectionnés est minimisé. Ce nombre est calculé par la contrainte (4) où la variable p_s vaut 1 si le motif p a été sélectionné par au moins une de ses occurrences m pour couvrir le graphe (3).

$$\forall p \in \text{EMDI} : \sum_{m \in M_p} m_{sel} > 0 \Leftrightarrow p_s = 1 \tag{3}$$

$$\text{NumberOfPatterns} = \sum_{p \in DISP} p_s \tag{4}$$

Les occurrences identifiées par notre méthode peuvent être non-convexes. Une occurence est dite non convexe lorsqu'il existe un chemin entre deux nœuds de l'occurence qui passe par un nœud n'appartenant pas à cette occurence (par exemple, $m7$ de la figure 6). Ces occurrences ne peuvent pas être sélectionnées car elles ne respectent pas les conditions temporelles d'exécution. Pour écarter automatiquement les occurrences non convexes d'un motif, nous imposons des contraintes sur les entrées et les sorties. Ces contraintes sont imposées d'après les dépendances de données de l'occurrence. Pour cela, nous déclarons tout d'abord une variable e_{start} pour chaque lien $e \in E$ du graphe d'application. Cette variable définit l'instant de départ de l'exécution d'une occurrence. Puis, nous imposons la contrainte (5) pour chaque occurrence.

$$m_{sel} = 1 \Rightarrow \forall_{e \in m_{in}} e_{start} = m_{start} \wedge \forall_{e \in m_{out}} m_{start} + m_{delay} \leq e_{start} \tag{5}$$

où la variable m_{start} définit l'instant de départ de l'exécution de l'occurrence m et la variable m_{delay} spécifie son temps d'exécution. L'ensemble m_{in} est l'ensemble des liens entrants de l'occurrence, c'est-à-dire les liens qui sont soit des entrées de l'application, soit des liens dont le nœud source n'appartient pas au motif. L'ensemble m_{out} est l'ensemble des liens sortants de l'occurrence, c'est-à-dire les liens qui sont

soit des sorties de l'application, soit des liens dont le nœud destination n'appartient pas au motif. Par exemple, si m_7 dans la figure 6 est sélectionné, alors $m7_{start} = e7_{start} = e8_{start} = e9_{start} = e10_{start}$ et $m7_{start} + m7_{delay} \leq e11_{start} \wedge m7_{start} + m7_{delay} \leq e12_{start}$. Cette solution n'est pas valide car au même moment, l'occurrence $m6$ doit être sélectionnée et cela conduit à la contradiction $e10_{start} \leq e12_{start}$ indiquant que la sortie doit être disponible avant l'entrée. Par conséquent, l'occurrence $m7$ ne peut pas être sélectionnée pour le graphe d'application de la figure 5.

4.2. Modélisation du temps d'exécution d'une occurrence

Notre environnement supporte deux modèles d'architecture (A et B). Dans le modèle A, l'extension du processeur, qui met en œuvre tous les motifs sélectionnés, ne possède pas de registre. Tous les opérandes nécessaires à l'exécution du motif sont sauvegardés dans la file de registre du processeur. Les opérandes sont lus depuis les registres du processeur et tous les résultats sont écrits dans ces registres. Dans le modèle B, l'extension du processeur possède ses propres registres pour sauvegarder les données. Les instructions de base du processeur, représentées par des motifs composés de un seul nœud, utilisent la file de registre du processeur comme dans le modèle A mais l'extension peut sauvegarder ses données dans ses propres registres. Les résultats produits par l'exécution d'une occurrence et utilisés par le processeur sont sauvegardés dans les registres du processeur. Dans ce modèle B, le nombre de cycles additionnels nécessaires au transfert des données est réduit mais sa complexité est plus élevée puisque le nombre de registres externes et de connexions externes est plus grand.

Le délai de l'occurrence m, m_{delay}, exprimé en nombre de cycles du processeur, est la somme de trois composantes, comme définit par l'équation (6).

$$m_{delay} = \delta_{in_m} + \delta_m + \delta_{out_m} \tag{6}$$

où δ_m est le temps d'exécution de l'occurrence m, δ_{in_m} représente le temps de lecture des opérandes d'entrée pour l'occurrence m et δ_{out_m} le temps d'écriture des résultats.

Le temps d'exécution d'une occurrence (δ_m) est le même pour les deux architectures mais le temps de transfert est différent. Pour le modèle A, le temps de transfert des données en lecture et en écriture est défini par les équations (7)-(8).

$$\delta_{in_m} = \lceil |pred(m)|/in\_PerCycle \rceil - 1 \tag{7}$$

$$\delta_{out_m} = \lceil |last(m)|/out\_PerCycle \rceil - 1 \tag{8}$$

où $pred(m)$ définit l'ensemble des nœuds prédécesseurs directs de l'occurrence m dans le graphe G, $in\_PerCycle$ est le nombre de registres lus par cycle du processeur, $last(m)$ est l'ensemble des nœuds terminaux de l'occurrence m, $out\_PerCycle$ est le nombre de registres écrits par cycle du processeur. Par exemple, pour le processeur NIOSII, $in\_PerCycle = 2$ et $out\_PerCycle = 1$. Si le temps d'exécution d'une occurrence est $\delta_m = 1$, $pred(m) = 2$ et $last(m) = 1$, alors le délai de l'occurrence est $m_{delay} = 1$.

Pour le modèle B, le temps de transfert des données est variable. Le temps de transfert en lecture est défini par les équations (9)-(10) et le temps de transfert en écriture est défini par les équations (11) à (13).

$$IN = \sum_{n \in pred1(m)} n_{sel} \tag{9}$$

$$\delta_{in_m} = \lceil IN/in\_PerCycle \rceil - 1 \tag{10}$$

$$\forall n \in last(m) : \sum_{m \in succ1(n)} m_{sel} > 0 \Leftrightarrow B_n = 1 \tag{11}$$

$$OUT = \sum_{n \in last(m)} B_n \tag{12}$$

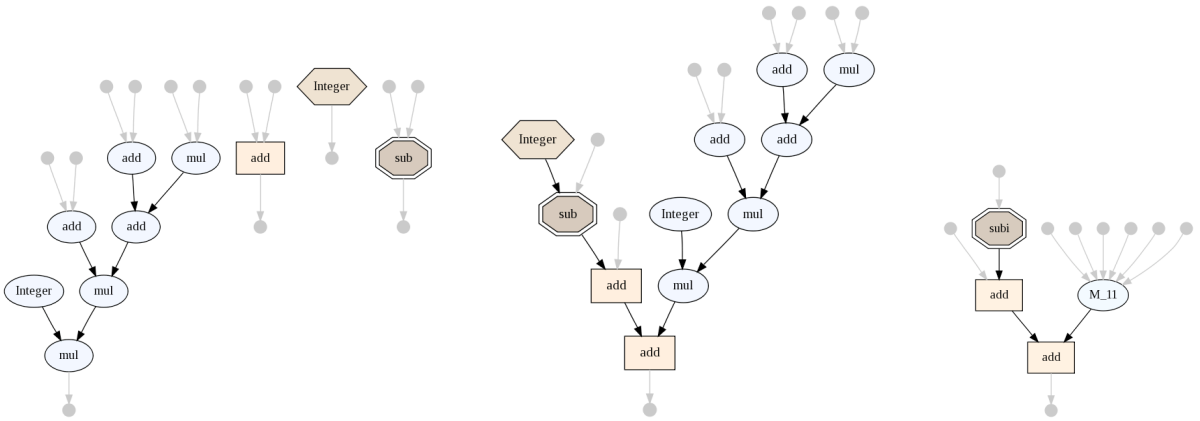
$$\delta_{out_m} = \lceil OUT/out\_PerCycle \rceil - 1 \tag{13}$$

où $pred1(m)$ représente l'ensemble des occurrences exécutées par le processeur qui sont des prédécesseurs de l'occurrence m dans le graphe G, n_{sel} est une variable qui vaut 1 si l'occurence n est sélectionnée et 0 sinon, $succ1(m)$ représente l'ensemble des occurrences de nœud successeurs du nœud n

dans le graphe G. La valeur de la variable IN est égale au nombre d'opérations de lecture dans la file de registres du processeur. La valeur de la variable OUT est égale au nombre d'opérations d'écriture dans la file de registres du processeur. Ces accès sont nécessaires car les données seront utilisées par les instructions exécutées sur le processeur.

4.3. Amélioration par recalage d'instructions

Lorsque le temps d'exécution d'une occurrence (δ_m) sur l'extension est suffisamment long[1], il est possible d'exécuter en parallèle une instruction sur le processeur en attendant la fin de l'exécution de l'occurrence. Le lancement des instructions est exclusif mais leur exécution peut être parallèle. Nous proposons d'améliorer l'ordonnancement obtenu à l'étape précédente en ajoutant cette opportunité, toujours grâce à la programmation par contraintes.



(a) Les motifs sélectionnés (b) Le graphe couvert (c) Le graphe réduit

FIG. 7 – Motifs sélectionnés, graphe couvert et graphe réduit

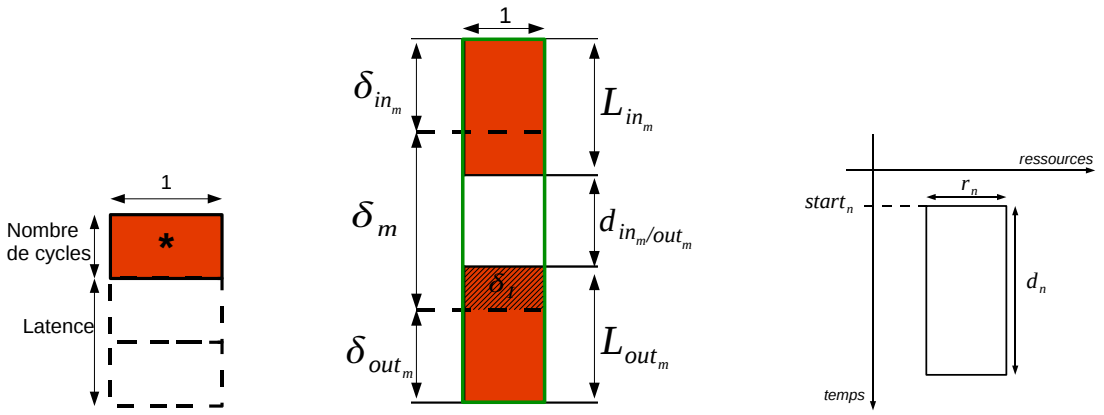
Après notre premier ordonnancement, nous obtenons le graphe d'application couvert par un ensemble de motifs. Nous pouvons alors construire un nouveau graphe, appelé graphe réduit, dans lequel toutes les occurrences sont modélisées par un seul nœud. Soit ce nouveau graphe $G' = (N', E')$. La figure 7(c) illustre le graphe réduit du graphe de la figure 7(b).

Tout d'abord, nous modélisons chaque occurrence sélectionnée sous forme de rectangle (figure 8(c)), de largeur égale au nombre de ressources utilisées et de longueur égale au temps d'occupation des ressources. Une occurrence exécutée sur le processeur est modélisée par un rectangle de largeur 1 (car l'instruction occupe une ressource, le processeur) et de longueur égale au nombre de cycles nécessaires pour exécuter cette instruction (typiquement 1). La figure 8(a) montre l'exemple pour l'instruction *mul* d'un NIOSII (version rapide) dont le temps d'exécution est de 1 cycle et le temps de latence est de 2 cycles. Le rectangle est donc de largeur 1 et de longueur 1. Une occurrence exécutée sur l'extension est modélisée par un double rectangle de largeur 1. Le premier rectangle correspond au nombre de cycles nécessaires à la communication pour les entrées plus le premier cycle toujours nécessaire pour le lancement de l'instruction ($L_{in_m} = \delta_{in_m} + 1$). Le second rectangle correspond au nombre de cycles nécessaires à la communication pour les sorties ($L_{out_m} = \delta_{out_m} + \delta_I$), où δ_I est défini par l'équation (14). Ces deux rectangles sont séparés d'une distance constante($d_{in_m/out_m} = \delta_m - 1 - \delta_I$). L'équation (15) montre bien l'équivalence avec l'équation (6), illustrée par la figure 8(b).

$$\delta_I = \begin{cases} 1 & \text{si } \delta_m > 1 \\ 0 & \text{sinon} \end{cases} \tag{14}$$

$$m_{delay} = \delta_{in_m} + \delta_m + \delta_{out_m} = L_{in_m} + d_{in_m/out_m} + L_{out_m} \tag{15}$$

[1] strictement supérieur à 2 cycles pour le modèle d'architecture A, supérieur à 1 cycle pour le modèle B dans le cas où le processeur ne lit pas le résultat de l'instruction



(a) Instruction *mul* sur un NIOSII

(b) Instruction spécialisée

(c) Un rectangle pour la contrainte cumulative

FIG. 8 – Modélisation des instructions sous forme de rectangles

Dans notre architecture cible, nous disposons d'une seule ressource, le processeur. Le processeur ne peut lancer qu'une seule instruction à la fois (soit un lancement, soit une communication), chaque instruction étant modélisée par un rectangle. Pour garantir l'exclusivité d'utilisation du processeur, il convient de s'assurer que les rectangles ne se chevauchent pas dans une colonne de largeur 1. Nous voulons donc obtenir un placement optimal de tous les rectangles, sans chevauchement, sur une colonne de largeur égale à 1 tout en respectant les dépendances de données. Pour cela, nous utilisons une contrainte cumulative. La contrainte cumulative permet d'assurer qu'à chaque instant, le total des ressources utilisées ne dépasse pas une certaine limite. La contrainte cumulative prend en paramètre un ensemble de rectangle, et une limite sur le nombre de ressources disponibles. Tous les rectangles sont ajoutés à cette contrainte cumulative, et nous fixons le nombre de ressources à 1, comme défini dans l'équation 16. Enfin, nous définissons la fonction de coût à minimiser pour cet ordonnancement (19) et nous imposons les contraintes sur les dépendances de données (17) et (18).

$$\forall n \in N' : Cumulative(\{\ldots\{start_n^i\},\{r_n^i\},\{d_n^i\}\ldots\},1)\ i \in R_n \tag{16}$$

où R_n est l'ensemble des rectangles modélisant un nœud n.

$$\forall n \in N' : start_n + delay_n = end_n \tag{17}$$

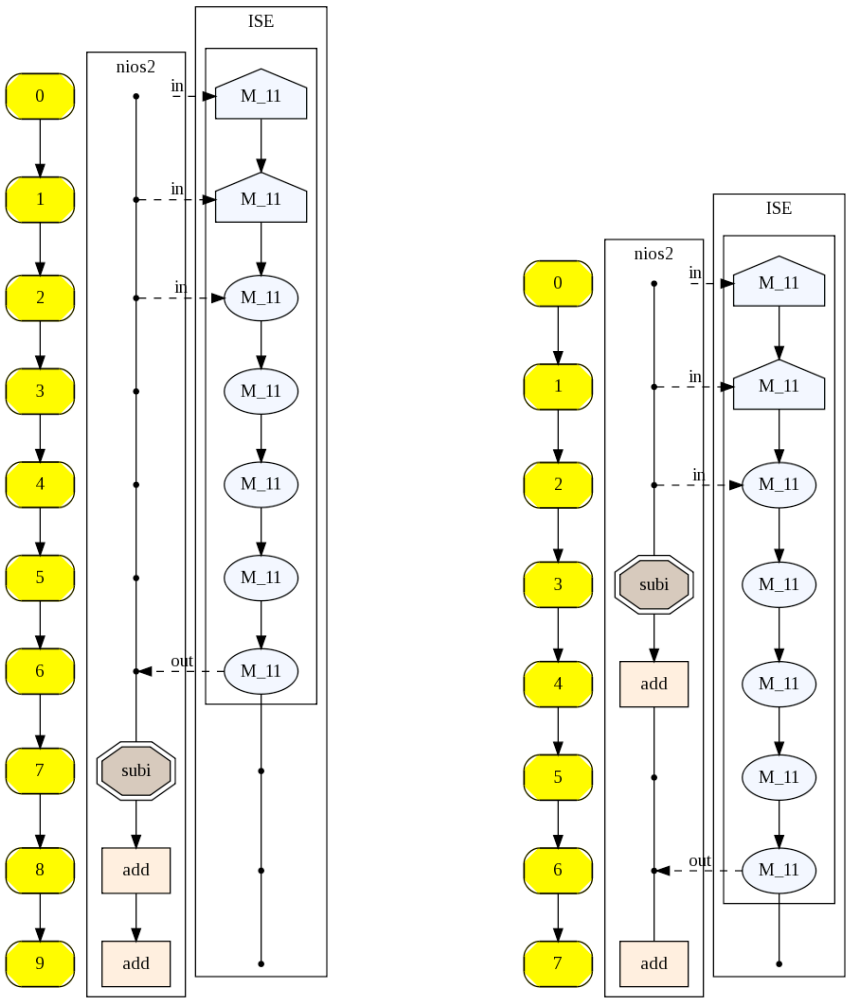
$$\forall (n,m) \in E' : end_n \leq start_m \tag{18}$$

$$\text{ExecutionTime } = max(end_n \forall n \in N') \tag{19}$$

La figure 9 montre un exemple d'ordonnancement sans recalage 9(a) et avec recalage d'instructions 9(b).

5. Résultats expérimentaux

Nous avons réalisé de nombreuses expérimentations pour évaluer à la fois la qualité des motifs générés et l'accélération possible en les implémentant en tant qu'instructions spécialisées. Nous avons utilisé les applications issues de *MediaBench* [13] et *MiBench* [8], écrites en langage C et compilées par l'environnement DURASE pour le processeur NIOSII d'Altera. Les résultats ont été comparés à ceux obtenus par UPaK qui ne prend pas en compte les contraintes sur le nombre d'entrées et de sorties. Les expérimentations ont été réalisées pour les modèles d'architecture A et B et pour des motifs générés sous différentes contraintes technologiques et architecturales, montrant ainsi la flexibilité et les capacités d'exploration offertes par DURASE. Le tableau 1 montre les résultats obtenus pour l'ensemble des benchmarks. Il montre le nombre de motifs identifiés et ceux sélectionnés pour obtenir l'accélération optimale du temps d'exécution de l'application, les couvertures du graphe correspondantes et l'accélération du temps d'exécution de l'application. Ces résultats ont été obtenus sous les contraintes suivantes :



(a) Sans le recalage d'instruction

(b) Avec le recalage d'instruction

FIG. 9 – Exemple d'ordonnancement du graphe de la figure 7(c) et de placement sur le modèle d'architecture B

le nombre de nœuds d'un motif ne dépasse pas 10, et le chemin critique du motif vaut au maximum 15 ns. Le chemin critique correspond à 3 cycles d'un processeur NIOSII cadencé à 200MHz sur un FPGA Stratix2 d'Altera. Pour chaque expérimentation, le coefficient *coef* (algorithme de la figure 3) a été choisi selon les propriétés du graphe de l'application. Ce coefficient est grand s'il y a beaucoup de motifs répétitifs et petit sinon. Le signe "-" dans le tableau indique que seul des motifs composés de 1 nœud ont été identifiés et sélectionnés, dans ce cas l'application est entièrement exécutée par les instructions de base du NIOSII. Deux types de motifs ont été utilisés pour l'expérimentation. Le premier type est limité à 2 entrées et 1 sortie, et le deuxième type contient les motifs limités à 4 entrées et 2 sorties. Pour le premier type, les cycles additionnels pour le transfert de données entre le NIOSII et son extension ne sont pas nécessaires. Pour le deuxième type, les transfert de données sont parfois nécessaires mais le temps d'exécution global de l'application est réduit.

Le tableau 2 montre les résultats obtenus par le système UPaK qui contraint seulement le nombre de nœuds dans les motifs générés (limité à 7 dans nos expérimentations). On note que notre nouvelle méthode génère d'excellents motifs. Le nombre de motifs sélectionnés est petit et la couverture est grande. Quand le nombre d'entrées autorisées est de 4 et le nombre de sortie est de 2, la couverture moyenne des graphes atteint 84% et est bien meilleur que celle obtenue avec le système UPaK qui ne limitait même

Benchmarks	noeuds	cycles	2 entrées / 1 sortie										4 entrées / 2 sorties									
			modèle A						modèle B				modèle A						modèle B			
			coef	identifiés	sélectionnés	couverture	cycles	accélération	sélectionnés	couverture	cycles	accélération	coef	identifiés	sélectionnés	couverture	cycles	accélération	sélectionnés	couverture	cycles	accélération
JPEG Write BMP Header	34	34	0	6	2	82%	14	2,42	2	82%	14	2,42	0	66	2	88%	12	2,83	3	88%	12	2,83
JPEG Smooth Downsample	66	78	0	5	2	19%	68	1,14	2	19%	68	1,14	0	49	4	95%	44	1,77	4	100%	35	2,22
JPEG IDCT	250	302	0,5	28	10	76%	214	1,41	10	76%	134	2,25	0,5	254	13	83%	141	2,36	15	89%	112	2,69
EPIC Collapse	274	287	0	11	8	68%	165	1,74	8	68%	165	1,74	0	111	11	71%	156	1,83	14	71%	159	1,8
BLOWFISH encrypt	201	169	0,5	11	3	74%	90	1,87	3	74%	90	1,87	0	153	8	90%	81	2,08	7	92%	73	2,31
SHA transform	53	57	0	5	3	64%	28	2,03	3	64%	28	2,03	0	48	8	98%	22	2,59	6	95%	17	3,35
MESA invert matrix	152	334	0,5	2	2	10%	320	1,04	2	10%	320	1,04	0,5	53	9	65%	262	1,27	9	65%	243	1,37
FIR unrolled	67	131	0	3	2	9%	126	1,04	2	9%	126	1,04	1	10	2	94%	98	1,30	2	97%	67	1,95
FFT	10	18	0	0	-	-	-	-	-	-	-	-	0	12	2	60%	10	1,80	2	60%	10	1,80
Moyenne						50%		**1,5**		50%		**1,7**				83%		**2**		84%		**2,3**

TAB. 1 – Résultats obtenus pour les benchmarks MediaBench et MiBench compilés pour un processeur NIOSII avec l'environnement DURASE.

Benchmarks	noeuds	cycles	Système UPaK									
			modèle A						modèle B			
			identifiés	sélectionnés	in/out	couverture	cycles	accélération	sélectionnés	in/out	couverture	cycles	accélération
JPEG Write BMP Header	34	34	4	2	2/2	79%	15	2,26	2	2/2	79%	15	2,26
JPEG Smooth Downsample	66	78	10	5	8/1	68%	71	1,10	4	8/1	71%	66	1,20
JPEG IDCT	250	302	7	4	4/2	46%	215	1,40	4	4/2	48%	212	1,42
EPIC Collapse	274	287	5	2	4/3	36%	220	1,30	2	4/3	36%	220	1,30
BLOWFISH encrypt	201	169	4	3	6/2	65%	135	1,25	3	6/3	65%	130	1,30
SHA transform	53	57	16	4	8/1	53%	35	1,60	4	8/1	53%	35	1,60
MESA invert matrix	152	334	5	3	8/3	33%	292	1,15	3	8/3	33%	289	1,15
FIR unrolled	67	131	3	1	8/1	83%	75	1,74	1	8/1	83%	75	1,74
FFT	10	18	3	1	4/1	60%	10	1,80	1	4/1	60%	10	1,80
Moyenne						58%		**1,51**			59%		**1,53**

TAB. 2 – Résultats obtenus pour les benchmarks MediaBench et MiBench avec le système UPaK.

pas les motifs par des contraintes technologiques et architecturales. Les temps d'exécution moyen des applications est amélioré de 2,3 (3,35 pour SHA transform). Ces résultats sont excellents pour un processeur tournant à 200MHz sur une carte FPGA (le nombre d'operateurs sur le chemin critique doit être petit pour pouvoir être exécuté en un cycle du processeur) et des applications présentant peu de parallélisme inhérent.

Cette partie ne présente pas les résultats obtenus après amélioration par recalage d'instruction. En effet, comme cité dans la partie 4.3, pour qu'il y ait opportunité d'optimisation, le delai d'exécution d'une instruction spécialisée doit être strictement supérieur à un seuil. Dans ces expérimentations, le chemin critique est limité à 15 ns (3 cycles du processeur). Les opportunités de recalage d'instruction sont peu nombreuses et l'amélioration ne peut pas réduire le temps d'exécution des applications. La seule amélioration observée est pour l'application JPEG IDCT. Cependant, en augmentant le chemin critique des motifs, on peut obtenir une augmentation des performances.

6. Travaux liés

Deux domaines sous-jacents aux travaux présentés dans ce papier sont brièvement introduits : la génération de motifs pour des graphes flot de contrôle et flot de données, et l'isomorphisme de graphe.

Les travaux concernant l'extraction et la sélection de motifs présentés dans [1, 4, 10] sont caractérisés par la similitude entre la génération et la correspondance de motifs, ceux-ci visent les processeurs de type ASIP. Dans [10], la recherche de motifs est effectuée par agglomération de nœuds en fonction de

la fréquence du type des nœuds successeurs. Dans [1, 4], les auteurs utilisent une agglomération incrémentale qui s'appuie sur des heuristiques dont l'objectif commun est d'identifier les motifs fréquents.
Une autre méthode présentée dans [2] concerne un algorithme de recherche de motifs qui les identifie en utilisant les contraintes de convexité et d'entrées/sorties. Des améliorations de cette méthode ont été proposées dans [3]. La recherche de motifs sous contraintes d'entrées/sorties est aussi utilisée dans [19]. L'algorithme classique a pour point de départ un nœud de sortie du bloc de base, il construit un sous-graphe en essayant d'ajouter récursivement les nœuds parents. Le sous-graphe assemblé est considéré comme étant une nouvelle instruction potentielle. Dans [6], un ensemble de sous-graphes appelés MaxMISO (*Multiple Input Single Output*) est d'abord identifié. Aucun MaxMISO n'est contenu dans un autre MISO. Ensuite, un ensemble candidat est sélectionné parmi les MISO à 2 entrées et 1 sortie trouvés dans le MaxMISO. Enfin, à partir des candidats sélectionnés, le graphe de l'application est partitionné par une méthode de recherche quasi-exhaustive en utilisant un algorithme de *branch and bound* (séparation et évaluation). Un flot complet de spécialisation d'un processeur est présenté dans [14], où les motifs sont agglomérés, l'un après l'autre, suivant des décisions locales. Dans la méthode présentée dans [7], les motifs sont assemblés de manière incrémentale en ajoutant les nœuds voisins aux occurrences des motifs non-isomorphes formés à l'itération précédente.
Notre approche est radicalement différente. Nous générons tous les motifs non-isomorphes satisfaisant toutes les contraintes architecturales et technologiques à partir de chaque nœud du graphe. La sélection de motifs dans l'ensemble des motifs générés est contrôlée par un processus de filtrage similaire à celui présenté dans [15]. Ce filtrage utilise les informations d'occurrence d'un motif fournies par une méthode basée sur l'isomorphisme de sous-graphe et résolu par la programmation par contraintes.
La sélection de motifs, le placement et l'ordonnancement sont des problèmes difficiles, leurs résolutions reposent généralement sur l'utilisation d'approches heuristiques : algorithmes gloutons, recuit simulé, algorithmes génétiques, recherche tabou, etc. Récemment, plusieurs approches intéressantes ont été proposées. Wang et.al. [22] utilisent l'algorithme *ACO (Ant Colony Optimization)*, ou optimisation par colonie de fourmis, et Guo et.al. [7] une heuristique basée sur l'ensemble maximum indépendant dans un graphe de conflit.
Notre approche pour la sélection de motifs, le placement et l'ordonnancement est entièrement définie par un modèle de contraintes. Nous définissons les contraintes de sélection de motifs en même temps que les contraintes de placement et d'ordonnancement, ce qui nous permet de résoudre le problème par des méthodes complètes ou des heuristiques. Dans notre système UPaK présenté précédemment [23], nous utilisions déjà la programmation par contraintes mais l'approche actuelle, basée sur une nouvelle modélisation du problème permet de supprimer les inconvénients précédents. Précisément, il n'est plus nécessaire de définir les *dummy nodes* qui surchargeaient considérablement les graphes d'applications utilisés lors de l'ordonnancement. De plus, notre nouvelle approche supprime les apparitions d'*occurrences fantômes* qui survenaient dans des cas bien précis à cause de la difficulté d'interprétation des résultats fournis par les contraintes d'isomorphisme de sous-graphe.
Différents types de morphismes entre des graphes et des sous-graphes ont été largement étudiés et beaucoup d'algorithmes et de méthodes ont été proposés. L'isomorphisme de graphe peut être résolu en affectant des étiquettes spécialement conçues, qui reflètent la structure du graphe, aux nœuds de ce graphe. Cette approche est utilisée par le programme appelé *nauty* [17] tout comme l'approche de programmation par contraintes qui utilise une méthode similaire [20]. Cette méthode est efficace et permet de trouver l'isomorphisme rapidement pour beaucoup de graphes mais ne peut pas être utilisée pour résoudre le problème d'isomorphisme de sous-graphe.
Le premier algorithme d'isomorphisme de sous-graphe a été développé par Ullmann dans [21]. Larossa et Valiente [12] ont étudié le problème d'isomorphisme de sous-graphe et des méthodes pour le résoudre en utilisant la satisfaction des contraintes. L'algorithme `VF2`, qui peut être utilisé à la fois pour l'isomorphisme de graphe et de sous-graphe, est présenté dans [18]. Cet algorithme peut être décrit par le moyen de *State Space Representation (SSR)*. Dans chaque état, une solution de correspondance partielle est maintenue et seuls les états cohérents sont conservés. Ces états sont générés en utilisant des *règles de faisabilité* qui suppriment des paires de nœuds qui ne peuvent pas être isomorphes. La comparaison entre `VF2` et nauty montre qu'aucun des deux n'est supérieur à l'autre, le résultat de la comparaison dépend des caractéristiques du graphe.
Nous utilisons une méthode très similaire à celle utilisée par `VF2` mais nous élargissons l'ensemble des

règles pour être capable de traiter les graphes directionnels et non-directionnels, et différents types de ports qui connectent les liens dans le graphe. Ceci est nécessaire dans les systèmes électroniques quand des opérations non commutatives, telles que la soustraction sont utilisées.

7. Conclusion

Nous avons présenté une nouvelle approche, principalement basée sur la programmation par contraintes, qui permet d'étendre automatiquement le jeu d'instructions d'un processeur. Nous avons décrit comment les applications sont placées et ordonnancées sur les nouvelles architectures. Nous avons également montré comment exploiter le parallélisme inhérent pour exécuter deux instructions en même temps à travers le recalage d'instructions. L'environnement est donc capable d'ordonnancer finement une application sur la nouvelle architecture. Nos expérimentations confirment l'efficacité de cette approche par rapport au système UPaK. La couverture moyenne des graphes a été augmentée de 69% à 84% et l'accélération moyenne des applications a été remontée de 1,53 à 2,3 (pour un NIOSII cadencé à 200MHz sur un FPGA Stratix2 d'Altera). Concernant les améliorations, une intégration de l'optimisation par recalage d'instruction directement dans la phase de sélection de motifs et d'ordonnancement est à l'étude.

Bibliographie

1. M. ARNOLD et H. CORPORAAL : Designing domain-specific processors. *Hardware/Software Codesign, 2001. CODES 2001. Proceedings of the Ninth International Symposium on*, p. 61–66, 2001.
2. K. ATASU, L. POZZI et P. IENNE : Automatic application-specific instruction-set extensions under microarchitectural constraints. *In DAC '03 : Proceedings of the 40th conference on Design automation*. ACM Press, 2003.
3. P. BISWAS, S. BANERJEE, N. DUTT, L. POZZI et P. IENNE : Isegen : Generation of high-quality instruction set extensions by iterative improvement. *42nd Design Automation Conference (DAC)*, p. 1246–1251, 2005.
4. N. CLARK, H. ZHONG et S. MAHLKE : Processor acceleration through automated instruction set customization. *In In MICRO*, p. 129–140, 2003.
5. GECOS : Generic compiler suite, http ://gecos.gforge.inria.fr/.
6. Y. GUO : *Mapping Applications to a coarse-grained Reconfigurable Architecture*. Thèse de doctorat, University of Twent, Eindhoven, Netherland, sept. 8, 2006.
7. Y. GUO, G. J. SMIT, H. BROERSMA et P. M. HEYSTERS : A graph covering algorithm for a coarse grain reconfigurable system. *In LCTES '03 : Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, 2003.
8. M. R. GUTHAUS, J. S. RINGENBERG, D. ERNST, T. M. AUSTIN, T. MUDGE et 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*, p. 3–14, Washington, DC, USA, 2001.
9. JACOP : Java constraint programming library, http ://jacop.cs.lth.se.
10. R. KASTNER, S. OGRENCI-MEMIK, E. BOZORGZADEH et M. SARRAFZADEH : Instruction generation for hybrid reconfigurable systems. *In ICCAD*, p. 127, 2001.
11. K. KUCHCINSKI et C. 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.
12. J. LARROSA et G. VALIENTE : Constraint satisfaction algorithms for graph pattern matching. *Mathematical Structures in Computer Science*, 12:403–422, 2002.
13. C. LEE, M. POTKONJAK et W. H. MANGIONE-SMITH : Mediabench : A tool for evaluating and synthesizing multimedia and communicatons systems. *In International Symposium on Microarchitecture*, p. 330–335, 1997.
14. R. LEUPERS, K. KARURI, S. KRAEMER et M. PANDEY : A design flow for configurable embedded processors based on optimized instruction set extension synthesis. *DATE*, p. 581–586, 2006.
15. K. MARTIN, C. WOLINKSI, K. KUCHCINSKI, F. CHAROT et A. FLOC'H : Constraint-driven identification of application specific instructions in the durase system. *SAMOS*, 2009.
16. K. MARTIN, C. WOLINKSI, K. KUCHCINSKI, F. CHAROT et A. FLOC'H : Durase : Generic environment for design and utilization of reconfigurable application-specific processors extensions. *DATE U-Booth*, 2009.
17. B. D. MCKAY : The nauty page. http ://cs.anu.edu.au/ bdm/nauty/, 2004.
18. L. P. CORDELLA, P. FOGGIA, C. SANSONE et M. VENTO : A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.
19. A. PEYMANDOUST, L. POZZI, P. IENNE et G. D. MICHELI : Automatic instruction set extension and utilization for embedded processors. *Application-Specific Systems, Architectures and Processors, IEEE International Conference on*, 0:108, 2003.
20. S. SORLIN et C. SOLNON : A global constraint for graph isomorphism problems. *Lecture notes in computer science ISSN 0302-9743*, p. 287–301, avr. 2004.
21. J. R. ULLMANN : An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.
22. G. WANG, W. GONG et R. KASTNER : System level partitioning for programmable platforms using the ant colony optimization. *In International Workshop on Logic & Synthesis (IWLS'04)*, Temecula, California, 2004.
23. C. WOLINSKI et K. KUCHCINSKI : Automatic selection of application-specific reconfigurable processor extensions. *DATE*, mars 10-14, 2008.
24. C. WOLINSKI, K. KUCHCINSKI, E. RAFFIN et F. CHAROT : Architecture-driven synthesis of reconfigurable cells. DSD, 2009.