Selection d'instructions et ordonnancement parallèle simultanes pour la conception de processeurs specialises

Antoine Floch, François Charot, Steven Derrien, Kevin Martin, Antoine Morvan, Christophe Wolinski

RenPar'20 / **SympA'14** / CFSE 8
Saint-Malo, France, du 10 au 13 mai 2011

Sélection d'instructions et ordonnancement parallèle simultanés pour la conception de processeurs spécialisés

Antoine Floch[1], François Charot[1], Steven Derrien[2], Kevin Martin[2], Antoine Morvan[1], Christophe Wolinski[2]

[1]INRIA Rennes - Bretagne Atlantique, France
[2]Université de Rennes I, Irisa, France

Résumé
Ce papier présente une méthode basée sur la programmation par contraintes qui résout de manière simultanée la sélection d'instructions et l'ordonnancement parallèle pour des processeurs spécialisés. Cette phase prend en entrée une bibliothèque de motifs, soit générée, soit décrite manuellement par le concepteur. Les résultats expérimentaux montrent une amélioration du temps d'exécution des applications allant jusqu'à un facteur 2 pour un surcoût matériel de 5,45%. Cette méthode est intégrée à un flot de conception mis en œuvre grâce aux techniques avancées du génie logiciel. Ce flot offre au concepteur la possibilité de décrire à haut niveau les instructions spécialisées et s'appuie sur les outils de synthèse pour en déterminer la latence.

Mots-clés : ASIP, Extensions de jeu d'instructions, Synthèse d'architecture, Programmation par contraintes

1. Introduction

Ces dernières années, les besoins en performance des systèmes embarqués n'ont cessé de croître, rendant leur conception de plus en plus complexe, et de fait plus risquée. Cette tendance qui ne va vraisemblablement pas s'inverser dans les années à venir, va de pair avec une pression de plus en plus forte sur la durée du cycle de conception de ces systèmes, dans le but de réduire au maximum le délai de mise sur le marché. Dans ce contexte, les processeurs ASIP offrent un compromis intéressant entre les performances d'un circuit dédié (ASIC) et la souplesse d'utilisation d'un processeur généraliste. La mise en œuvre complète d'un système à base d'ASIP reste cependant délicate, notamment parce qu'elle implique la génération d'une description matérielle complète de la micro-architecture du processeur, laquelle doit souvent se faire à partir d'une spécification de très haut niveau de la machine, qui peut parfois se limiter à la description de son jeu d'instructions.

Pour ces raisons, il est bien moins risqué de s'appuyer sur des solutions dans lesquelles le cœur de la micro-architecture (et une grande partie de son jeu d'instructions) est fixé, mais où il est possible d'ajouter des fonctionnalités au processeur au travers d'extensions de son jeu d'instructions [12]. Ces extensions, très finement couplées au processeur, permettent d'améliorer de manière significative ses performances pour certaines parties critiques de l'application. Un exemple commercial de ce type de processeur est le processeur *soft-core* NIOSII de la société Altera [1]. Dans ce processeur, l'extension prend la forme d'un module matériel qui vient se greffer en parallèle de l'UAL du processeur et qui peut donc accéder en lecture et en écriture à sa file de registres.

Une instruction spécialisée peut se définir comme un regroupement d'opérations élémentaires (opérations arithmétiques ou logiques) qui sera entièrement réalisé par l'extension. Si l'approche permet de s'affranchir de nombreux problèmes, l'identification des regroupements[1] pertinents pour une exécution sur l'extension, tout comme la prise en compte de ces extensions dans la phase de sélection d'instructions sont des problèmes complexes. Cette difficulté est de deux ordres.

– Tout d'abord la plupart des algorithmes mis en jeu dans ces problèmes d'identification et de sélection souffrent d'un risque important d'explosion combinatoire. Par exemple l'isomorphisme de sous-graphes et la couverture de sommets d'un graphe sont des problèmes connus pour être NP-complets.
– Ensuite, la multiplicité des cibles technologiques et la difficulté d'évaluer des gains ou coûts en surface précis empêchent de guider de manière pertinente le choix des extensions.

[1] Dans la suite de cet article nous parlerons plutôt de motifs

Dans ce travail, nous présentons un flot de synthèse d'extensions de jeu d'instructions original. Les contributions de ce travail sont les suivantes.

– Nous proposons un flot complet qui intègre directement les outils de synthèse RTL et de placement routage dans la boucle de conception. Les mesures précises de temps et de surface ainsi obtenues guident ensuite plus finement le choix et l'exploitation des motifs de calculs déportés vers l'extension du processeur.
– En plus de permettre l'identification automatique de motifs de calculs bon candidats à une accélération matérielle, notre flot permet aux utilisateurs experts de spécifier dans le code source de l'application (sous la forme de fonctions annotées) les motifs jugés pertinents.
– À l'issue de cette étape d'identification, la phase de sélection et d'ordonnancement simultanés de ces instructions spécialisées, ainsi que la phase de synthèse matérielle de l'extension sont entièrement automatisées. En particulier, nous proposons un algorithme original[2] qui résout conjointement les problèmes de sélection et d'ordonnancement. Cette résolution exploite les techniques de programmation par contraintes [26].
– Enfin, ce flot a également la particularité de reposer sur une utilisation systématique d'outils et de méthodes issus de l'Ingénierie Dirigée par les Modèles (IDM), et notamment de l'infrastructure Eclipse Modeling Framework [10].

L'article est organisé comme suit. Nous commençons par présenter dans la section 2 le flot de conception dans son ensemble, en détaillant nos contributions par rapport à l'existant. La section 3 s'intéresse quant à elle plus précisément au problème de l'identification des motifs et positionne notre contribution par rapport à l'existant. La section 4 détaille la principale contribution de l'article, à savoir l'approche abordant conjointement le problème de la sélection et de l'ordonnancement des motifs sur l'extension. Les résultats expérimentaux qui démontrent la viabilité de l'approche sont donnés en section 5.

2. Présentation du flot de conception

Notre infrastructure de compilation (figure 1) pour la synthèse d'extensions de jeux d'instructions aborde le problème à la fois sous l'angle de la compilation mais aussi de la synthèse matérielle. En conséquence, celle-ci manipule diverses représentations, utilisées à des niveaux d'abstraction très variés.

– Une représentation intermédiaire d'un programme C qui mêle l'arbre de syntaxe abstrait du programme et un graphe représentant son flot de contrôle.
– Une représentation intermédiaire de type graphe flot de données pour représenter à la fois l'organisation des calculs au sein d'un bloc de base, mais aussi pour décrire les motifs de calculs qui seront utilisés comme instructions.
– Une représentation bas-niveau (RTL) de composants matériels, servant à la modélisation de la structure de l'extension.

Chacune de ces représentations est conçue dans l'optique d'aider à la résolution d'un sous-problème particulier. La mise en œuvre de l'infrastructure au complet implique de nombreuses étapes qui réaliseront le passage d'une représentation vers une autre. La multiplicité de ces représentations au sein de ce type de flot pose dès lors de nombreux problèmes de conception, qui relèvent souvent plus du génie logiciel que de la compilation. Ce constat a donc motivé l'utilisation des techniques issues de la communauté du génie logiciel (IDM).

2.1. Description du flot

Dans ce paragraphe, nous présentons chacun des quatre principaux blocs du flot de conception de la figure 1.

Infrastructure de compilation GeCoS

GeCoS [14] est une infrastructure de compilation pour le langage C, dont la particularité est d'être complètement intégrée à l'environnement Eclipse. Dans sa version actuelle, GeCoS peut-être utilisée soit dans un contexte de transformation source à source (C vers C) soit comme une infrastructure de compilation reciblable pour la synthèse d'ASIP et/ou l'extension de processeurs programmables [19, 24]. Dans ce travail, nous utilisons deux éléments de GeCoS.

1. Le *front-end* C, sert à construire une représentation intermédiaire, dans laquelle les opérations des blocs de base et leurs dépendances (contrôle et données) sont représentées sous la forme d'un graphe flot de données

[2] Méthode faisant partie d'une soumission en cours pour le journal "ACM Transactions on Reconfigurable Technology and Systems", http://trets.cse.sc.edu/index.html

acyclique. Cette représentation est ensuite utilisée pour extraire des motifs spécifiés par l'utilisateur (à l'aide de fonctions annotées par la directive `GCS_PATTERN`) qui représentent les motifs à sélectionner, et qui seront utilisés pendant l'opération de couverture.

2. Le *back-end* source à source est utilisé lors la phase de régénération d'un code C exploitant les extensions du jeu d'instructions. Le code sera ensuite compilé avec le compilateur croisé du NIOSII. Dans ce code C, l'appel aux extensions définies par l'utilisateur se fait par l'utilisation d'instructions assembleur *en-lignes*.

La mise en œuvre de ce regénérateur de code exploite un des points forts de l'infrastructure GeCoS, à savoir son extensibilité. En effet, il est possible d'étendre la représentation intermédiaire de base utilisée par GeCoS, en y ajoutant de nouvelles constructions, ou en spécialisant des constructions existantes. Les passes d'analyse et d'optimisation existantes peuvent alors être étendues pour supporter ces extensions (sans nécessiter de modifications de leur code source) à l'aide d'un système de *points d'extension* qui se charge de réaliser, à l'exécution, l'édition de liens entre le cœur de l'infrastructure et les greffons qui lui sont associés.

EMF4RTL : Méta-modèle pour la manipulation de circuits

Le but d'EMF4RTL est de fournir une boîte à outils permettant de spécifier, d'analyser, et de transformer des représentations de circuits au niveau RTL. EMF4RTL peut-être considéré comme un héritier des langages de description de circuits de type structurel tels JHDL [5], Jazz [6], etc. Sa spécificité est d'être orienté vers la modélisation et l'optimisation de circuits synchrones, et d'offrir une grande palette d'outils (langage spécifique pour la spécification de flots de contrôle complexes, générateur de code VHDL, SystemC, prototype de flot source à source pour VHDL). Parmi les analyses et transformations possibles, EMF4RTL offre, entre autres, des fonctionnalités de *retiming* [17] et de fusion de chemin de données [23].

Compilation NIOSII

La sélection et l'ordonnancement d'occurrences de motifs (cf. section 4) issus d'une bibliothèque (extraite ou générée, cf. section 3) est un problème difficile qui requiert un haut niveau d'expertise de la part du concepteur. L'objectif de notre flot est d'automatiser ce processus. Une fois ces informations extraites, chaque occurrence sélectionnée est remplacée par une instruction spécialisée dont l'opérateur unique code le chemin de données utilisé dans l'extension.

Synthèse de l'extension

L'extension à synthétiser est composée d'un ensemble de blocs de calcul correspondant aux motifs sélectionnés. Ces blocs de calcul sont connnectés avec les registres du processeur ou des registres internes à l'extension. Nous disposons d'une phase d'allocation des registres internes à l'extension[3] pour obtenir une description précise de l'architecture de l'extension à synthétiser. Un langage dédié permet de décrire simplement les blocs de l'extension et les différents contextes de configuration supportés. Chaque contexte correspond à une instruction spécialisée indiquant les correspondances entre les ports d'entrées/sorties des motifs et l'interface ou des registres de l'extension. Ce fichier de description de l'extension est généré à partir des informations obtenues à la fin de l'étape précédente et est utilisé pour générer du code VHDL synthétisable.

2.2. Une approche basée sur l'ingénierie dirigée par les modèles

Les objectifs de l'IDM sont de fournir des outils et des méthodes pour des infrastructures logicielles manipulant de multiples niveaux de représentations intermédiaires. Par exemple l'infrastructure EMF (*Eclipse Modeling Framework*) propose un ensemble d'outils tels que la génération de code à partir d'un diagramme de classe, le génération de texte à partir d'une représentation intermédiaire (M2T, *Model to Text*) ou encore la possibilité de définir très facilement des langages dédiés ainsi que tout leur environnement (analyse syntaxique, éditeur intégré à Eclipse, etc. (T2M, *Text to Model*)).

L'outil présenté dans ce papier utilise ce paradigme de programmation à chaque étape du flot pour en accélérer le développement et en faciliter l'intégration. Il est à noter que l'application des principes de l'IDM pour la spécification, la conception et la vérification de systèmes embarqués n'est pas nouvelle, et de nombreux autres outils (basés notamment sur le profil UML Marte) existent déjà. Parmi ceux-ci on peut en particulier noter des travaux [16] effectués autour de la synthèse d'accélérateurs matériels pour des chaînes de traitement flot de données multi-dimensionnels.

[3] non décrite dans cet article

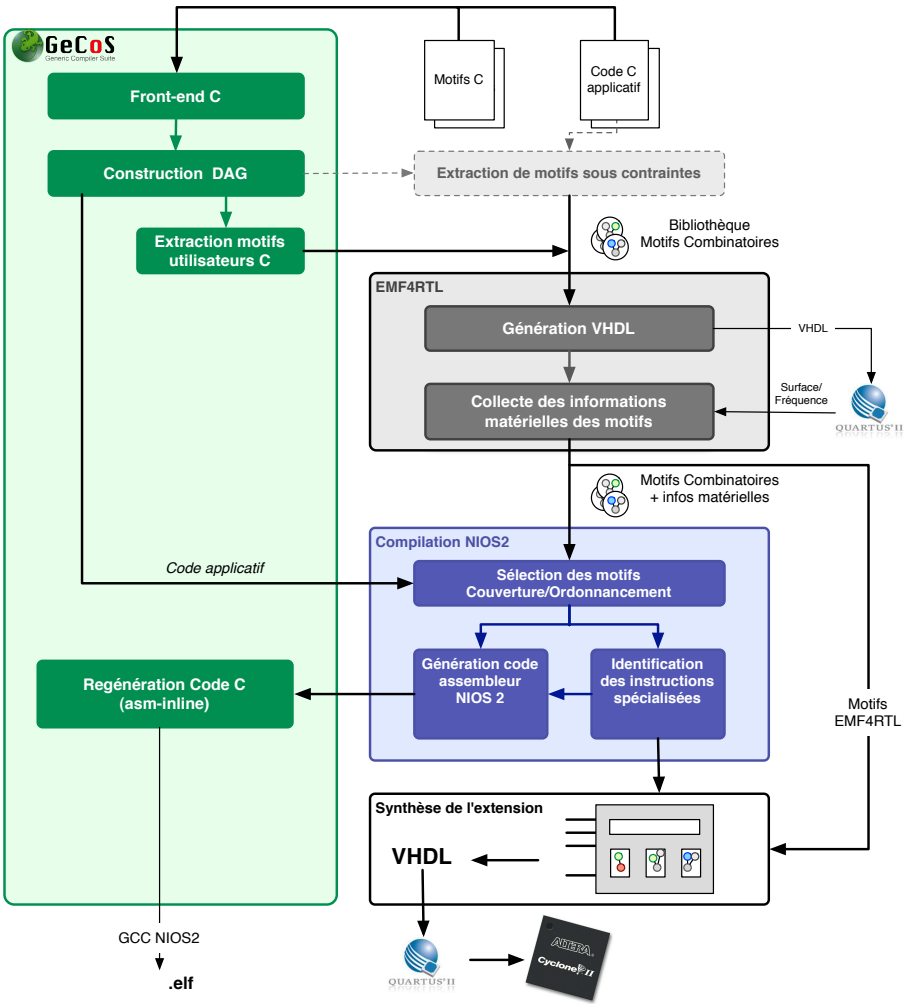


FIG. 1 – Infrastructure pour la synthèse d'extensions de jeux d'instructions de processeurs

Ces outils ne ressemblent pas vraiment à des outils de compilation au sens strict du terme. Ils se basent en effet sur une spécification d'entrée exprimée dans un domaine métier (ex : modèle flot de données multi-dimensionnel) au travers d'un langage dédié, dont la syntaxe concrète peut être graphique ou textuelle. Par ailleurs, les flots existants dans la littérature ne semblent pas nécessiter la mise en œuvre de phases d'optimisations complexes. Ils relèvent donc plus d'outils de transformations dirigées par la syntaxe (ou en l'occurrence par les modèles).
À notre connaissance, notre flot est le premier à mêler l'IDM à une infrastructure de compilation classique opérant à partir d'un langage de programmation généraliste de bas niveau tel que C.

3. Identification des motifs

L'étape d'identification des motifs dont l'exécution sera déportée sur l'extension est un élément critique du flot. Ce problème est délicat, dans la mesure où les motifs sélectionnés doivent satisfaire de nombreuses contraintes, tant topologiques (nombre d'opérandes, nombre de nœuds sur le chemin critique) que technologiques (coût en surface et délai combinatoire du motif sur la cible technologique) ou architecturales (contraintes sur le nombre d'entrées et de sorties).
Dans cette section, nous présentons les deux contributions de ce travail, à savoir :

– une estimation précise des caractéristiques des motifs en permettant de coupler l'outil d'identification des motifs avec les outils de *back-end* associés à la technologie cible ;
– la possibilité offerte au concepteur de spécifier directement dans son programme les motifs qu'il souhaite pouvoir utiliser lors de la phase de sélection/ordonnancement.

3.1. Estimation fine des caractéristiques des motifs

Une des difficultés technologiques est d'évaluer précisément le délai d'exécution d'un motif de calcul sur l'extension. Une estimation trop optimiste risque d'avoir un impact sur le chemin critique du processeur et ainsi limiter sa fréquence de fonctionnement, ralentissant du coup l'ensemble du système. À l'inverse, une estimation conservatrice peut faire perdre de nombreuses opportunités d'accélération.

Notre approche proposée dans [21] évalue la durée d'exécution d'un motif (ainsi que son coût en surface) en se basant sur le coût et le délai de chaque opérateur considéré individuellement. Dans le cas de cibles technologiques de type FPGA, cette estimation peut s'avérer être très imprécise car elle ignore alors à la fois l'impact du *mapping technologique* (les cellules logiques des FGPA sont devenues très complexes et offrent de nombreuses opportunités d'optimisation aux outils de synthèse) et le fait que le chemin critique d'un circuit combinatoire résultant de la fusion/concaténation d'un ou plusieurs opérateurs arithmétiques (ou logiques) est souvent très inférieur à la somme des chemins critiques de chaque opérateur considéré séparément. À titre d'exemple, le délai (pour une FPGA Cyclone II) d'une multiplication 32 bits est de 6,6 ns et celui d'une addition est de 4,2 ns, ce qui fait une somme de 10,8 ns, alors que le délai de la multiplication directement suivie de l'addition est de 7,8 ns (soit seulement 72% du délai estimé).

Dans un tel contexte, l'estimation sur la base d'une bibliothèque d'opérateurs va clairement amener l'outil à un choix de solution sous optimale. De fait, de par la complexité des technologies ciblées (FPGA) il semble difficile de pouvoir offrir une estimation à la fois précise et robuste (sur un grand nombre de familles FPGA, avec différents speed-grade, etc.). Il nous semble donc que la meilleure solution à ce problème consiste à intégrer directement les outils de synthèse RTL (ou de placement routage) directement dans la boucle de conception, en les utilisant au cours d'une phase de *calibrage*, ce dans le but d'obtenir une information précise à la fois sur le comportement temporel et sur le coût en ressource d'un motif candidat.

De nombreux flots de conception se basent sur la synthèse logique des instructions spécialisées potentielles pour déterminer précisément leur coût et leur latence [25, 27, 3]. Si ils permettent d'obtenir des chiffres précis pour estimer les performances du processeur spécialisé, ces flots souffrent d'un surcoût en temps de traitement. Par ailleurs, les résultats de synthèse passent par un test vérifiant que la fréquence des instructions candidates est compatible avec celle du processeur cible. Si tel n'est pas le cas, l'instruction est simplement rejetée [27]. Dans nos travaux, la latence d'une instruction spécialisée est utilisée pour déterminer le nombre de cycles nécessaires pour être exécutée par un processeur dont la cadence est fixée. Nous n'adressons pas le problème d'insertion automatique de registres dans le chemin de données de l'instruction spécialisée.

3.2. Spécification manuelle des motifs d'extension

Dans le contexte où le concepteur a une idée précise du type d'extension qu'il souhaite ajouter à son cœur de processeur, une piste intéressante est de pouvoir spécifier, directement dans le code source de l'application, les motifs de calculs susceptibles d'être utilisés pour la mise en œuvre de l'extension de jeu d'instructions. À notre connaissance, il n'existe pas de flot disponible offrant cette possibilité.

Le flot présenté dans ce papier permet de spécifier directement dans le code source de l'application le motif de calcul dont on souhaite déporter l'exécution sur l'extension. Cette fonctionnalité se fait très simplement en créant une fonction C qui réalise le calcul demandé, puis en y associant la directive de compilation `#pragma GCS_PATTERN` comme illustré par la figure 2.

Seul un sous ensemble de C peut-être utilisé pour décrire des motifs, en particulier, il n'est pas possible d'utiliser des structures de contrôle (boucles, conditionnelles), ni d'accéder à des tableaux ou de manipuler des pointeurs. Les paramètres passés à la fonction correspondent aux entrées/sorties du motif, les types d'entrée autorisés se limitent à des scalaires, les types de sortie sont nécessairement des pointeurs sur des scalaires (on suppose que les valeurs des données pointées sont seulement écrites et jamais lues).

À la fin de cette étape d'identification de motifs, nous disposons d'une bibliothèque de motifs, générés ou extraits. L'étape suivante est de sélectionner les motifs qui offrent la meilleure accélération, tout en tenant compte de l'ordonnancement des instructions.

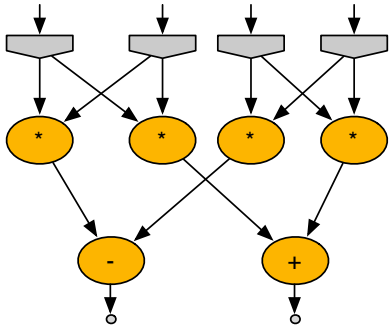
4. Sélection et ordonnancement des instructions spécialisées

La sélection des motifs à exécuter sur une extension est un problème d'optimisation pouvant tenir compte de plusieurs critères : durée d'exécution, nombre de motifs sélectionnés, nombre de nœuds, surface matérielle utilisée, etc. La pertinence de cette sélection est fortement liée aux contraintes matérielles de l'architecture (partage des

```
#pragma GCS_PATTERN
void butterfly(int a, int b,int c,
      int d,int *out1, int *out2){

    *out1 = a*b + c*d;
    *out2 = a*b - c*d;
}
```

(a) Description C du motif



(b) Motif extrait

FIG. 2 – Exemple de motif directement spécifié dans le code source de l'application, et sa représentation sous forme de graphe flot de données

ressources de calcul, de mémoire et de communication) limitant les possibilités d'ordonnancement des motifs.
La contribution présentée dans cette section consiste à spécialiser notre précédente technique [11] de sélection et d'ordonnancement conjoints au cas d'un processeur NIOSII couplé à une extension simple disposant de registres internes. L'approche proposée permet de résoudre de manière simultanée, en une seule étape, les problèmes de sélection et d'ordonnancement, alors que notre approche présentée dans [22] le fait en deux étapes successives.

4.1. Approches existantes pour la sélection de motifs

Certains algorithmes déterminent directement dans le graphe cible les sous-ensembles de nœuds qui seront exécutés sur l'extension. Le problème est vu comme un problème de partitionnement des nœuds [4], [2], [18], [13] souvent exprimé dans un problème de programmation linéaire en nombres entiers[4]. Atasu et.al [2] déterminent à chaque itération de leur algorithme l'occurrence maximisant l'accélération du chemin critique du reste du graphe à analyser. Galuzzi et .al [13] décomposent le graphe en MaxMISO. La résolution d'un problème ILP permet ensuite de déterminer les meilleures instructions MIMO combinaisons des MaxMISO.
Générer et sélectionner conjointement les motifs dans un ensemble de graphes risque de produire un nombre important d'instructions spécialisées. Dans un contexte ou ce nombre est limité, il est souvent plus pertinent d'identifier et de sélectionner dans le graphe cible des instances de motifs issus d'une bibliothèque. Guo et.al. [15] proposent un algorithme itératif s'appuyant sur un graphe de conflit. Bonzini et.al. [7] proposent trois algorithmes. Le premier est un algorithme glouton qui détermine à chaque itération le meilleur motif dans le graphe restant à couvrir en tenant compte du nombre d'occurrences de chaque motif sélectionné. Le deuxième est également incrémental mais néanmoins optimal. Le dernier est une heuristique analysant un sous-ensemble des solutions explorées dans l'algorithme optimal. Cong [9] exprime le problème sous forme de couverture binaire qui a la particularité d'autoriser le recouvrement des occurrences sélectionnées. Clark et.al. [8] définissent une méthode basée sur la programmation dynamique pour sélectionner les meilleures occurrences sous contraintes de surface. L' algorithme de « branch and bound » décrit dans [20] a l'originalité de chercher une solution au problème de sélection d'occurrences pouvant s'exécuter en parallèle sur une extension VLIW. Néanmoins, cet algorithme ne traite pas les difficultés associées au partage du processeur. Celui-ci exécute les instructions non accélérées, il est en plus chargé du lancement des instructions spécialisées et de leur alimentation en données. Les travaux présentés dans [22] tiennent compte des contraintes d'un NIOSII couplé à une extension pour déterminer une sélection minimisant la durée d'exécution globale. L'ordonnancement entre le processeur et l'extension est alors effectué par une étape succédant à celle de sélection. L'approche décrite dans [11] modélise quant à elle l'ordonnancement et la sélection dans un unique problème mais, contrairement à ce papier, cible un processeur connecté à plusieurs extensions parallèles.

4.2. Sélection et ordonnancement conjoints

Soient P une bibliothèque de motifs et $G(N, E)$ un graphe d'application, où N et E correspondent respectivement à l'ensemble des nœuds et des liens du graphe. Si M représente l'ensemble des occurrences (isomorphisme de sous-graphes) de P dans G, l'objectif de la sélection et de l'ordonnancement est de déterminer pour chaque nœud $n \in N$:
– o_n : l'occurrence couvrant n.

[4] ILP : Integer Linear Programming

– t_n, d_n : le début et la durée de l'exécution de n.
L'ensemble de ces informations constitue une couverture du graphe G par la bibliothèque de motifs P.
La suite de ce paragraphe explique les bases de l'approche [11] qui modélise dans un unique problème d'optimisation la sélection des occurrences et leur ordonnancement. L'énoncé de ce problème repose sur la programmation par contraintes [26] qui permet notamment d'exprimer des contraintes globales non-linéaires particulièrement adaptées aux problèmes de partage de ressources.

Sélection des occurrences

Chaque occurrence $m_i \in M$ est identifiée par un indice unique i. Un nœud n est associé à un ensemble $occurrences_n$ référençant les identifiants des occurrences de motifs contenant n. Cet ensemble détermine les valeurs du domaine fini de la variable $o_n :: \{i \mid m_i \in occurrences_n\}$ qui identifie l'occurrence sélectionnée après couverture. Ainsi, le nombre de variables de décision est proportionnel au nombre de nœuds, contrairement à notre approche présentée dans [21] où ce nombre était proportionnel au nombre d'occurrences[5]. Le tableau 1 donne un exemple des domaines des variables de sélection o_n pour chaque nœud à l'initialisation du problème et après résolution (couverture de la figure 3).

variables	domaine après initialisation	domaine après sélection
o_{n_1}	$\{0, 7\}$	$\{0\}$
o_{n_2}	$\{1, 7\}$	$\{1\}$
o_{n_3}	$\{2, 6, 8\}$	$\{2\}$
o_{n_4}	$\{4, 8\}$	$\{4\}$
o_{n_5}	$\{3\}$	$\{3\}$
o_{n_6}	$\{2, 5, 7\}$	$\{2\}$

TAB. 1 – Domaines des variables o_n pour tous les nœuds après initialisation et sélection (Figure 3).

La contrainte globale `Count` utilisée dans (1) permet de déterminer si une occurrence est sélectionnée ou non. Cette contrainte maintient un vecteur $list_m = [o_n | n \in m]$ contenant les variables des occurrences pour le nœud n et $count_m :: \{0, size(m)\}$ est utilisé pour compter le nombre de nœuds couverts par l'occurrence m. Cette contrainte spécifie que pour chaque nœud $n \in m_i$, soit $o_n = i$ soit $o_n \neq i$. Puisque le domaine de la variable o_n est composé des indices des occurrences contenant n, pour toute solution valide, chaque nœud est couvert par une et une seule occurrence. La contrainte (2) définit la variable sel_{m_i} en utilisant une contrainte « réifiée » (`reified`). Cette variable vaut un si l'occurrence est sélectionnée, et zéro sinon.

$$\forall m_i \in M : \text{Count}(i, list_{m_i}, count_{m_i}) \quad (1)$$

$$\forall m_i \in M : count_{m_i} > 0 \Leftrightarrow sel_{m_i} \quad (2)$$

Modèle temporel

Pour appliquer les contraintes d'ordonnancement, nous définissons les variables correspondant au début et au délai d'une occurrence m. Ces variables sont appelées $start_m$ et $delay_m$. Nous définissons également, pour chaque nœud $n \in N$, les variables t_n et d_n associées au début et au délai de l'occurrence qui le couvre. Tout nœud $n \in m_i$ voit ses variables respectivement égales aux variables de l'occurrence : $t_n = start_{m_i}$ et $d_n = delay_{m_i}$. Ceci est assuré par la contrainte (3) et par le fait que pour chaque occurrence sélectionnée m_i, les variable $o_n = i$ des nœuds $n \in m_i$ sont égales. La contrainte `Element` impose une relation entre I et les variables V, en appliquant la formule $V = List[I]$ où List est un vecteur de variables à domaine fini.

[5] Le nombre d'occurrence est souvent bien supérieur au nombre de nœuds

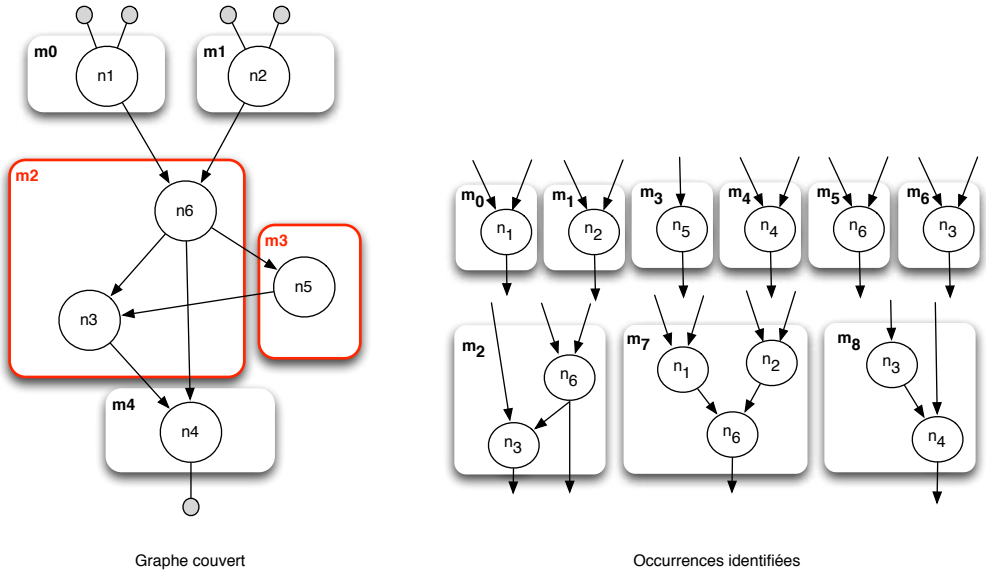


FIG. 3 – Exemple de couverture avec des occurrences non-convexes (m_2,m_3) : invalide dans le cadre d'un ordonnancement.

$$\forall_{n \in N} \text{ Element}(o_n, \text{List}_{start}, t_n), \text{ Element}(o_n, \text{List}_{delay}, d_n) \quad (3)$$
$$\text{avec } \text{List}_{start} = [start_m \mid m \in M] \text{ et } \text{List}_{delay} = [delay_m \mid m \in M]$$

Nous imposons donc que toutes les entrées de l'occurrence sélectionnée m soient disponibles en même temps et que toutes ses sorties soient disponibles après $delay_m$ (cf. l'occurrence m_2 de la figure 3). Dans le reste du papier, $delay_m$ est considéré comme étant la somme du temps d'exécution de l'occurrence m sur l'extension et du temps de communication nécessaire au transfert de données entre le processeur et l'extension.
Le respect des dépendances de données est assuré par la contrainte (4) où s et d sont respectivement les nœuds source et destination du lien $e \in E$. Cette contrainte conditionnelle IF-THEN n'est appliquée que pour les liens du graphe connectant deux occurrences. Cela permet de modéliser la création dynamique du graphe couvert pendant le processus de résolution.

$$\forall_{(s,d) \in E} \quad \text{IF } o_s \neq o_d \text{ THEN } t_d \geq t_s + d_s \quad (4)$$

Dans le cas général, une occurrence, qu'elle soit convexe ou non-convexe, est détectée par l'isomorphisme de sous-graphes. Or, la sélection d'une occurrence non-convexe introduit un cycle dans le graphe couvert et rend celui-ci impossible à ordonnancer. Les contraintes temporelles (3-4) suffisent à garantir la légalité de l'ordonnancement et donc la convexité des occurrences sélectionnées. Par exemple, si l'occurrence m_2 de la figure 3 est sélectionnée, alors $(start_{m_2}{=}t_{n_6}{=}t_{n_3}) \wedge (start_{m_3}{=}t_{n_5}) \wedge (t_{n_6} + d_{n_6} \leq t_{n_5}) \wedge (t_{n_5} + d_{n_5} \leq t_{n_3})$ ce qui est impossible car $(d_{n_5} = delay_{m_3} \neq 0) \wedge (d_{n_6} = d_{n_3} = delay_{m_2} \neq 0)$.

Minimisation de la durée d'exécution

L'ordonnancement recherché minimise la durée d'exécution de la totalité du graphe d'application. Il s'agit donc de minimiser la fin du dernier nœud exécuté.

$$C(G) = \text{Max}([t_n + d_n | \forall n \in N]) \quad (5)$$

Résoudre le problème dans l'état actuel correspond en fait à identifier la couverture de l'application qui minimise la durée d'exécution parallèle sans contraintes de ressources. Le résultat obtenu est évidemment loin d'être le plus

intéressant dans le cas d'une cible matérielle réelle et donc restreinte au niveau des ressources d'exécution, de mémoire et de communication.

4.3. Modèle d'exécution des motifs sur l'architecture

Le modèle de contraintes précédent peut être raffiné pour tenir compte du partage des ressources de l'architecture cible. Dans la suite de ce paragraphe on s'intéressera à un modèle de contraintes[6] dédié à une exécution sur un processeur (NIOSII par exemple) couplé à une extension séquentielle (qui ne peut exécuter qu'une seule instruction spécialisée à la fois).

Caractéristiques de l'extension

Dans le contexte du flot proposé, on considère que l'extension synthétisée ne peut exécuter qu'un seul motif à la fois. Néanmoins, durant l'exécution d'un motif multicycles, le processeur sera disponible pour exécuter une séquence d'opérations issues de son jeu d'instructions standard. Par défaut, on considère que toutes les occurrences sélectionnées de motifs à un seul nœud sont exécutées sur le processeur. Les autres occurrences sont exécutées sur l'extension. La disponibilité du processeur (et donc le parallélisme potentiel avec l'extension) est réduite du fait qu'il soit responsable de l'alimentation en données et du lancement de l'exécution de chaque motif exécuté sur l'extension.

Partage des ressources

L'exécution d'une occurrence est modélisée par des rectangles placés dans un espace spatio-temporel (cycles sur l'axe des abscisses et ressources d'exécution sur celui des ordonnées). Seulement deux ressources sont disponibles, (1) le processeur (valeur $y = 0$), et (2) l'extension (valeur $y = 1$). Tous les rectangles sont définis par $Rect = [x, y, \Delta x, \Delta y]$, où x et y sont les coordonnées du coin en bas à gauche du rectangle, et Δx et Δy correspondent respectivement à la longueur et la hauteur du rectangle.
L'objectif est de partager les ressources d'exécution et de communication en répartissant des tâches dans l'espace spatio-temporel disponible. Les occurrences à un nœud, exécutées sur le processeur, sont modélisées par un rectangle $Rect_m = [start_m, 0, delay_m, sel_m]$. Les occurrences à nœuds multiples, exécutées sur l'extension, nécessitent trois tâches :

1. Occupation du processeur lors de l'alimentation en données (partage des deux entrées disponibles) et du lancement de l'exécution de l'occurrence sur l'extension :

$$Rect1_m = [start_m, 0, ERT_m + 1, sel_m] \tag{6}$$

   avec ERT_m : nombre de cycles nécessaires pour fournir plus de deux données à l'extension.

2. Occupation du processeur lors de la récupération des résultats (partage de l'unique sortie de l'extension) :

$$Rect2_m = [swr_m, 0, (EWT_m + MultiCycle) \cdot IsWriteTransfer_m, sel_m] \tag{7}$$

   avec :
   - swr_m : début de l'écriture des données produites par l'extension vers le processeur.
   - EWT_m : nombre de cycles nécessaires pour récupérer plus d'une donnée venant de l'extension.
   - $MultiCycle$: vaut 1 si la durée d'exécution du motif est supérieure à un cycle, 0 sinon.
   - $IsWriteTransfer_m$: vaut 1 si l'occurrence produit au moins une donnée vers le processeur, 0 sinon.

3. Occupation de l'extension pour l'occurrence m :

$$Rect3_m = [start_m, 1, delay_m, sel_m] \tag{8}$$

Les rectangles $Rect1_m$ et $Rect2_m$ permettent d'éviter l'exécution d'une instruction sur le processeur pendant le transfert de données du processeur vers l'extension ou inversement, ou pendant le lancement d'une instruction spécialisée. Le rectangle $Rect3_m$ modélise le temps d'occupation de l'extension pendant tout le temps d'exécution de l'instruction spécialisée, composée de trois phases, lecture, exécution, et écriture. Finalement, la contrainte globale (9) empêche le recouvrement des rectangles et assure ainsi une exécution correcte des instructions sur le

[6] Soumis pour le journal "ACM Transactions on Reconfigurable Technology and Systems", http://trets.cse.sc.edu/index.html

processeur et son extension. Par manque de place, les contraintes initialisant les valeurs des variables utilisées dans les rectangles ne sont détaillées ici.

$$\begin{aligned} & \texttt{Diff2}(\mathsf{ListRect}) \\ \text{avec } \mathsf{ListRect} = & [\mathsf{Rect}_m \mid m \in M \wedge \mathsf{size}(m) = 1] \ +\!\!+ \\ & [\mathsf{Rect1}_m \mid m \in M \wedge \mathsf{size}(m) \neq 1] \ +\!\!+ \\ & [\mathsf{Rect2}_m \mid m \in M \wedge \mathsf{size}(m) \neq 1] \ +\!\!+ \\ & [\mathsf{Rect3}_m \mid m \in M \wedge \mathsf{size}(m) \neq 1] \end{aligned} \tag{9}$$

où $+\!\!+$ symbolise la concaténation de vecteurs.

5. Résultats expérimentaux

Nous avons réalisé de nombreuses expérimentations pour évaluer les améliorations obtenues pour un processeur NIOSII (version *fast*), cadencé à 150 MHz sur un FPGA CycloneII EP2C35F672C6 d'Altera, avec une extension. Nous avons généré automatiquement des motifs grâce à notre technique présentée dans [21]. Les contraintes sur les motifs générés sont les suivantes : au maximum 4 entrées, 2 sorties, et 10 nœuds. Les motifs répondant à ces contraintes permettent une large couverture du graphe, comme nous l'avons montré dans nos travaux [21].
Le tableau 2 présente les résultats obtenus pour des algorithmes issus de *MediaBench*, *MiBench* et de bibliothèques cryptographiques, écrits en langage C. Dans ce tableau, |N|, |P| et |M| indiquent respectivement le nombre de nœuds dans le graphe, le nombre de motifs identifiés, et le nombre d'occurrences de ces motifs. |Psel| et |Msel| indiquent le nombre de motifs sélectionnés et le nombre d'occurrences sélectionnées.

Algorithmes	|N|	|P|	|M|	|Psel|	|Msel|	Amélioration	Nombre de *Logic Element*	Surcoût matériel (%)
JPEG IDCT	268	138	496	44	105	1,41	2 447	95,88
MiBench BF encrypt	632	46	620	36	283	1,50	1 118	43,80
MiBench BF decrypt	632	46	620	36	283	1,50	939	36,79
Mediabench Sha transform	194	67	233	22	50	2,05	700	27,43
Mediabench arf coef	64	954	1266	17	18	1,75	1 004	39,34
MCrypt Cast 128	279	129	366	pas de solutions en 30s			335	13,12
MiBench fft	15	28	60	8	31	1,15	472	18,5
MCrypt gost	33	137	168	6	6	2,29	139	5,45

TAB. 2 – Résultats d'amélioration du temps d'exécution des algorithmes après sélection et ordonnancement, et surcoût matériel induit par l'extension.

Les résultats d'amélioration du temps d'exécution ont été obtenus en utilisant une méthode de recherche standard fournie par le solveur JaCoP. Cette méthode est une méthode complète de parcours en profondeur de l'arbre de recherche (*Depth-first search*). L'ordre d'évaluation des variables a un impact important sur l'efficacité de la recherche. Trouver le bon ordre des variables n'est pas simple, et est bien souvent dépendant du problème à résoudre. Pour notre problème de sélection et d'ordonnancement simultanés, une manière efficace de choisir les variables est d'évaluer en premier lieu les variables liées au temps de début et à la couverture des occurrences présentes sur le chemin critique de l'application. Cette technique permet de prouver l'optimalité des solutions mais peine à trouver des solutions pour des graphes à chemin critique assez court mais à fort parallélisme. Pour des graphes de grande taille (plus de 50 nœuds), nous avons appliqué une méthode de recherche itérative qui consiste à grouper les variables et à résoudre le problème pour chaque groupe. La solution obtenue pour le premier groupe est réinjectée dans le problème, puis le deuxième groupe est résolu et ainsi de suite. Cette technique permet d'obtenir des résultats intéressants. Toutefois, il n'est pas toujours possible d'aboutir à une solution. C'est le cas par exemple pour l'algorithme *MCrypt Cast 128*, où aucune solution n'est trouvée dans les 30 secondes imparties.
Le tableau montre également les résultats obtenus après synthèse de l'extension. La métrique choisie pour mesurer la surface de l'extension est le *Logic Element*, l'élément logique de base du FPGA cible. Un élément logique est

Motif	Nombre de nœuds / sur le chemin critique	composition du chemin critique	chemin critique estimé (ns)	chemin critique mesuré (ns)	erreur (%)	Nombre de cycles NIOSII (150 MHz)	
						estimé	réel
p1	4/1	and (1 LUT)	2,38	2,38	0	1	1
p2	4/1	and (1 LUT)	2,38	2,38	0	1	1
p3	3/3	xor-sub-add (2 LUT)	10,41	5,57	186	2	1
p4	9/3	3 or (1 LUT)	7,14	2,38	333	2	1

TAB. 3 – Comparaison des chemins critiques estimés et mesurés pour des motifs de l'algorithme *MCrypt Cast 128*

constitué d'une LUT à quatre entrées et d'une bascule D. Les résultats obtenus pour l'algorithme IDCT montrent que l'extension peut nécessiter autant d'éléments logiques que le processeur. Ceci s'explique par les nombreuses instructions spécialisées composées de une voir plusieurs opérations de multiplication. Par ailleurs, nous n'appliquons pas les techniques de fusion de chemin de données des motifs. Ces techniques permettraient d'améliorer grandement les résultats. Pour les algorithmes du domaine cryptographique, les extensions générées sont moins gourmandes en ressources. Ainsi, pour l'algorithme GOST par exemple, il est possible d'améliorer d'un facteur 2 le temps d'exécution de la fonction d'encryptage pour un surcoût matériel de 5,45% seulement.

Pour l'algorithme *MCrypt Cast 128*, nous avons appliqué notre flot avec description manuelle des motifs et appel aux outils de synthèse pour en mesurer le chemin critique. Quatre motifs qui semblent pertinents pour le concepteur ont été décrits. Le tableau 3 présente un bref résumé de ces quatre motifs. La première colonne indique le nom du motif. La deuxième colonne donne le nombre de nœuds qui composent ces motifs (les valeurs immédiates sont stockées dans un nœud) pour donner un aperçu de leur complexité, ainsi que le nombre de nœuds qui composent son chemin critique, sachant qu'un décalage avec une valeur immédiate n'induit aucune latence et n'est donc pas compté s'il se trouve sur le chemin critique. La quatrième colonne indique la latence du chemin critique estimée par notre technique présentée dans [21], qui consiste à simplement faire la somme des opérations prises individuellement. La colonne suivante montre la latence mesurée, c'est-à-dire la latence fournie par les outils de synthèse (QuartusII v10.0). La colonne erreur indique le pourcentage d'erreur de notre technique d'estimation, qui a un impact direct sur le calcul du nombre de cycles nécessaires à un NIOSII cadencé à 150 MHz pour exécuter le motif.

Pour des motifs très simples, comme les motifs p1 et p2, les résultats montrent que le pourcentage d'erreur est nul. En toute logique, dès que le nombre de nœuds sur le chemin critique s'allonge, le pourcentage d'erreur augmente. C'est le cas pour le motif p4 où le pourcentage d'erreur grimpe jusqu'à 333%. À l'aide de ces quatre motifs, notre technique de sélection et d'ordonnancement est capable de trouver une solution et d'améliorer de 22% le temps d'exécution de l'algorithme *MCrypt Cast 128* pour un surcoût matériel d'environ 13%.

6. Conclusion

Dans cet article, nous avons présenté une méthode qui résout de manière simultanée la sélection et l'ordonnancement des calculs sur un processeur et son extension. Les résultats présentés dans ce travail permettent d'obtenir des améliorations en performance allant jusqu'à un facteur 2 pour seulement 5,45% de ressources matérielles en plus. Cette méthode fait partie intégrante d'un flot complet pour l'extension de jeu d'instructions ciblant les processeurs NIOSII de la société Altera. Ce flot, qui est entièrement basé sur une approche à base d'ingénierie des modèles, permet de combiner des techniques d'identification automatiques et manuelles des motifs mis en œuvre sur l'extension. Par ailleurs, nous avons montré que l'évaluation de la durée d'exécution des motifs est souvent faussée par les optimisations réalisées par l'outil de synthèse. Pour répondre à ce problème notre flot intègre une étape automatisée de synthèse et donne une connaissance précise de la durée d'exécution réelle d'un motif sur le FPGA cible. Une perspective envisagée est d'appliquer des techniques de fusion de chemin de données pour réduire la surface occupée par les motifs de l'extension.

Bibliographie

1. ALTERA, http://www.altera.com.
2. K. ATASU, G. DÜNDAR et C. ÖZTURAN : An integer linear programming approach for identifying instruction-set ex-

tensions. *In CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, p. 172–177, New York, NY, USA, 2005. ACM.

3. K. ATASU, W. LUK, O. MENCER, C. OZTURAN et G. DUNDAR : Fish: Fast instruction synthesis for custom processors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, PP(99):1, 2010.
4. 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.
5. P. BELLOWS et B. HUTCHINGS : Jhdl - an hdl for reconfigurable systems. *In Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, FCCM '98, p. 175–, Washington, DC, USA, 1998. IEEE Computer Society.
6. G. BERRY : Esterel and jazz: Two synchronous languages for circuit design (abstract). *In Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, CHARME '99, p. 1–, London, UK, 1999. Springer-Verlag.
7. P. BONZINI et L. POZZI : Recurrence-aware instruction set selection for extensible embedded processors. *IEEE Trans. Very Large Scale Integr. Syst.*, 16(10):1259–1267, 2008.
8. N. T. CLARK, H. ZHONG et S. A. MAHLKE : Automated Custom Instruction Generation for Domain-Specific Processor Acceleration. *IEEE Trans. Comput.*, 54(10):1258–1270, 2005.
9. J. CONG, Y. FAN, G. HAN et Z. ZHANG : Application-specific instruction generation for configurable processor architectures. *In FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, p. 183–189, New York, NY, USA, 2004. ACM Press.
10. EMF : Eclipse modeling framework, http://www.eclipse.org/modeling/emf/.
11. A. FLOCH, C. WOLINSKI et K. KUCHCINSKI : Combined scheduling and instruction selection for processors with reconfigurable cell fabric. *In Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, p. 167 –174, 2010.
12. C. GALUZZI et K. BERTELS : The Instruction-Set Extension Problem: A Survey. *In ARC '08: Proceedings of the 4th international workshop on Reconfigurable Computing*, p. 209–220. Springer-Verlag, 2008.
13. C. GALUZZI, E. M. PANAINTE, Y. YANKOVA, K. BERTELS et S. VASSILIADIS : Automatic selection of application-specific instruction-set extensions. *In CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/-software codesign and system synthesis*, p. 160–165, New York, NY, USA, 2006. ACM.
14. GECOS : Generic compiler suite, http://gecos.gforge.inria.fr/.
15. 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.
16. S. LE BEUX, P. MARQUET et J.-L. DEKEYSER : A design flow to map parallel applications onto fpgas. *In Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, p. 605 –608, 2007.
17. C. E. LEISERSON et J. B. SAXE : Retiming Synchronous Circuitry. *Algorithmica*, 6(1):5–35, 1991.
18. 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.
19. L. L'HOURS : Generating Efficient Custom FPGA Soft-Cores for Control-Dominated Applications. *In Application-Specific Systems, Architectures and Processors, IEEE International Conference on*, vol. 0, p. 127–133, Los Alamitos, CA, USA, 2005. IEEE Computer Society.
20. Y.-S. LU, L. SHEN, L.-B. HUANG, Z.-Y. WANG et N. XIAO : Optimal subgraph covering for customisable vliw processors. *IET Computers and Digital Techniques*, 3(1):14–23, 2009.
21. K. MARTIN, C. WOLINSKI, K. KUCHCINSKI, F. CHAROT et A. FLOC'H : 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*, p. 194–203, Samos, Greece, 2009. Springer-Verlag.
22. K. MARTIN, C. WOLINSKI, K. KUCHCINSKI, A. FLOC'H et F. CHAROT : Sélection automatique d'instructions et ordonnancement d'applications basés sur la programmation par contraintes. *In 13ème Symposium en Architecture de machines (SympA'13)*, Toulouse, France, 2009.
23. N. MOREANO, E. BORIN, C. de SOUZA et G. ARAUJO : Efficient datapath merging for partially reconfigurable architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(7):969–980, July 2005.
24. M. PASHA, S. DERRIEN et O. SENTIEYS : System level synthesis for ultra low-power wireless sensor nodes. *In Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, p. 493 –500, 2010.
25. 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.
26. F. ROSSI, P. v. BEEK et T. WALSH : *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.
27. F. SUN, S. RAVI, A. RAGHUNATHAN et N. JHA : Custom-instruction synthesis for extensible-processor platforms. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(2):216 – 228, 2004.