RENPAR'15 / CFSE'3 / SympAAA'2003
La Colle sur Loup, France, 15 au 17 octobre 2003

Compilation d'un langage orienté objet pour une exécution répartie dans un composant reconfigurable : le parallélisme de classe

Frédéric Raimbault* et Dominique Lavenier

VALORIA, Université de Bretagne Sud
Campus de Tohannic
56000 Vannes - France
frederic.raimbault@univ-ubs.fr

IRISA, Université de Rennes I
Campus de Beaulieu
35042 Rennes - France
dominique.lavenier@irisa.fr

Résumé
Les travaux décrits dans cet article visent à faciliter l'utilisation des accélérateurs de calculs construits avec des composants *FPGA*. L'approche que nous prônons est de combler une partie du fossé entre algorithme et matériel reconfigurable en concevant un ensemble de briques de base, génériques, directement exploitables par un compilateur. L'assemblage de ces briques de base forme un multiprocesseur à mémoire distribuée, implantable dans seul un composant *FPGA*. Nous présentons le parallélisme de classe, un moyen d'exprimer et de réaliser l'exécution répartie d'un langage à objets séquentiel sur une machine multiprocesseur. Nous montrons les techniques de compilation utilisées pour exploiter de manière transparente et efficace le parallélisme de classe.

Mots-clés : parallélisme, langage orienté objet, compilation, reconfigurable

1. Introduction

Les travaux décrits dans cet article visent à faciliter l'utilisation des accélérateurs de calcul construits avec des composants *FPGA*. Les composants *FPGA* sont issus de la technologie du reconfigurable [13] ; ils sont principalement composés de blocs logiques (une fonction booléenne à quelques entrées) et d'interconnexions reprogrammables. Cette capacité à être reconfiguré rend le matériel à base de composants *FPGA* potentiellement adaptable à toute application. C'est pourquoi cette technologie s'est rapidement imposée dans la conception de machines spécialisées. Grâce à la technologie reconfigurable un architecte de machine peut prétendre exploiter tout le parallélisme potentiel d'une application : il dispose des ressources matérielles nécessaires à l'implantation d'opérateurs spécialisés en nombre suffisant et à leur interconnexion ; il peut optimiser les chemins de données et favoriser globalement l'utilisation concurrente des fonctions de calcul, d'entrées-sorties et de mémorisation. La seule limitation physique réside dans la capacité totale du ou des composants employés. Cependant, si la capacité des composants *FPGA* n'est plus un problème pour y implanter la totalité d'une application[2], cette technologie est actuellement réservée aux architectes de machine et reste hors de portée du programmeur d'application.

Exemple introductif

L'exemple suivant sera repris tout au long de l'article pour illustrer notre démarche. Supposons que nous cherchions à réaliser un accélérateur pour la recherche par le contenu dans des banques de données

* En délégation CNRS dans le projet Symbiose à l'IRISA.

[2] Par exemple, le Virtex-II, composant *FPGA* commercialisé par la société Xilinx pour le prix d'un processeur généraliste, contient 100 000 CLB (blocs logiques), soit l'équivalent de 8 millions de portes logiques.

génomiques. De nombreuses machines spécialisées à base de *FPGA* [6, 8, 4, 9] ont été proposées pour répondre précisément à ce type de traitement, fondamental en bio-informatique.
La solution générique est de connecter une carte à base de *FPGA* à une machine hôte : la machine hôte contient la banque de données ou permet d'y accéder ; le *FPGA* joue le rôle d'un co-processeur pour accélérer l'algorithme de recherche par le contenu. L'algorithme de référence dans le domaine de la bio-informatique est celui de Smith et Waterman [14]. Il consiste à calculer l'indice de similarité entre une séquence de test et une séquence contenue dans la base de données.
À partir de cette spécification minimaliste, il existe un grand nombre d'implantations matérielles possibles. Pour prétendre accélérer le traitement par rapport à une implantation à base de microprocesseur standard, il importe d'exploiter tout le parallélisme présent dans cette application.

- Au niveau de l'opérateur de comparaison : l'algorithme de comparaison entre deux séquences est implantable sous la forme d'un opérateur pipeliné ; il a déjà fait l'objet d'une mise en oeuvre systolique, par exemple dans [7]. Un tel opérateur est capable de délivrer un résultat partiel à chaque cycle.
- Au niveau des entrées-sorties : pour alimenter rapidement l'opérateur de comparaison il faut prévoir un recouvrement des tâches de calcul et d'entrées-sorties avec la base de données.
- Au niveau des données à traiter : les comparaisons de la séquence test avec chaque séquence de la base de données sont totalement indépendantes. Il est donc possible de lancer en parallèle autant de calculs que l'on dispose d'opérateurs de comparaison.

Parvenu à ce stade un programmeur d'application recherchera les outils à sa disposition pour décrire l'algorithme de comparaison, le dispositif d'alimentation en données, le mécanisme de parallélisation des comparaisons, etc afin de produire automatiquement l'architecture répondant à son objectif.
Le problème est qu'il n'existe pas à l'heure actuelle d'outils de synthèse d'architecture équivalent à ceux utilisés en développement de logiciel et qui permettraient de synthétiser rapidement et automatiquement une architecture dédiée à partir d'une spécification ou d'un programme exprimé dans un langage de haut niveau [15].

L'approche ROOM

L'approche que nous prônons est de combler une partie du fossé entre algorithme et matériel reconfigurable en concevant un ensemble de briques de base, génériques, directement exploitables par un compilateur. L'assemblage de ces briques de base forme un support d'exécution appelé une ROOM [11] – acronyme de *Reconfigurable Object Oriented Machine* – implantable dans un seul composant *FPGA*. Les briques de base sont soit des processeurs génériques, soit des processeurs spécialisés au niveau de leurs opérateurs pour un domaine d'application particulier. L'ensemble constitue un multiprocesseur à mémoire distribuée pour lequel nous fournissons un environnement de programmation objet. Le nombre et le choix des processeurs est déterminé par le programmeur lors de la décomposition de l'application en classes ; un processeur est associé à chaque classe de l'application : c'est le principe du parallélisme de classe. La synthèse des processeurs et des opérateurs spécialisés n'est pas abordée dans cet article. Elle est actuellement effectuée à l'aide des outils de CAO standards et des bibliothèques de composants (IP) pour la plate-forme reconfigurable choisie. On notera que, dans notre approche, cette synthèse n'est réalisée qu'une seule fois. Après conception des opérateurs pour un domaine d'application, ceux-ci peuvent être exploités par programmation pour toute une gamme d'algorithmes. Les motivations et les apports de notre approche dans le domaine de la conception d'architecture sont détaillés dans [12]. Nous nous focalisons dans cet article sur les aspects langage et compilation du parallélisme de classe.

Les langages concurrents à objets

La méthodologie objet possède plusieurs atouts pour la programmation d'un système distribué [2], en particulier dans le cas d'une machine parallèle intégrée dans un composant *FPGA* : d'une part, l'encapsulation des données et des traitements dans les classes répond au principe de localité prôné dans la réalisation de circuit ; d'autre part la hiérarchisation permet de maîtriser la complexité d'un design important. Pour autant, les différentes approches de l'utilisation des objets dans le contexte de la programmation parallèle et répartie ne sont pas directement utilisables dans le contexte d'un système synthétisé dans un circuit. Dans [5], l'auteur réalise une synthèse des langages concurrents à objets et propose une classification basée sur l'expression de la concurrence d'une part et sur le contrôle de la concurrence d'autre part. Concernant l'expression de la concurrence, tous les langages étudiés associent objet et processus ;

or les objets étant créés à l'exécution il est difficilement concevable (dans l'état actuel de la technologie reconfigurable) de leur associer du matériel qui serait créé dynamiquement (à l'exécution) et viendrait s'ajouter au système existant. Par contre il est possible avec le parallélisme de classe, en associant statiquement (à la compilation) du matériel à une classe, d'obtenir une concurrence inter-objet entre objet de classes distinctes. Concernant le contrôle de cette concurrence inter-objet, notre choix a été de le rendre totalement implicite. Ce choix nous distingue des travaux présentés dans [15] où le langage proposé pour la programmation d'un *FPGA* associe également un processus à chaque classe mais laisse à la charge du programmeur la synchronisation et les échanges inter-processus.
Le reste de cet article est décomposé de la manière suivante. Dans le paragraphe 2 nous présentons le langage de programmation objet utilisé et la notion de parallélisme de classe. Dans le paragraphe 3 nous exposons les éléments essentiels d'une ROOM pour supporter le parallélisme de classe. Dans le paragraphe 4 nous détaillons les techniques de compilation qui permettent de passer du modèle de programmation d'un langage à objets séquentiel au modèle d'exécution réparti d'une ROOM. Enfin nous donnons en conclusion l'état de nos travaux et les perspectives envisagées.

2. Le modèle de programmation

Dans notre approche, la programmation d'une application sur un *FPGA* s'effectue dans un langage de programmation orienté objet, dénommé SOL.

2.1. Le langage SOL

Le langage SOL est très proche syntaxiquement du langage JAVA ; la majeure partie des concepts sémantiques liés aux objets sont également repris.
Un programme SOL est structuré en classes à partir desquelles sont créés (instanciés) des objets à l'exécution. Une classe contient à la fois l'ensemble des informations (attributs) propres à chaque objet (attributs d'instance), ou communes à tous les objets de la classe (attributs de classe), et les fonctions (méthodes) applicables aux objets de cette classe (méthodes d'instance) ou à des données communes aux objets (méthodes de classe). Une classe peut être construite à partir d'une (seule) autre classe (la classe mère) par héritage simple : elle contient alors en plus tous les attributs et méthodes (publiques) de la classe mère. Toutes ces notions présentes dans le langage SOL servent à guider implicitement la structuration matérielle du support d'exécution sous-jacent (cf. paragraphe 2.2). À noter que, comme dans le langage JAVA, la dichotomie entre références (objets) et valeurs (types primitifs) est conservée pour des raisons de performances.
À l'intérieur des classes de l'utilisateur, les algorithmes des méthodes sont codés de manière séquentielle, à l'aide des structures de contrôle conditionnelles et itératives classiques. Une méthode principale (`main`) présente dans une seule classe de l'utilisateur sert comme en JAVA de lanceur du flot d'exécution.
Une différence notable avec le langage JAVA est l'absence du constructeur de type tableau et de ses opérateurs associés. Dans le langage SOL les tableaux sont explicitement manipulés comme des instances de classes[3] : il existe dans la bibliothèque de classes prédéfinies (l'API SOL) une classe générique tableau à une dimension (`Array`) dont on déclare une instance en passant en paramètre (à la C++) le type des éléments (par exemple : `Array<char>`). Et les opérateurs d'indexation ([ ]) sont remplacés par deux méthodes d'accès (`putAt` et `getAt`). La raison de ce choix est liée à deux facteurs déterminant sur le plan des performances :
- un facteur technologique : il est difficilement concevable de doter chaque processeur d'une ROOM d'une mémoire principale de taille importante ; par ailleurs une mémoire globale partagée serait un goulot d'étranglement trop pénalisant. L'usage des tableaux est donc réservé à quelques classes prédéfinies dont les processeurs associés sont connectés à une mémoire externe.
- un facteur applicatif : rappelons que nous visons des applications où les processeurs généralistes sont mal adaptés, en particulier quand il s'agit de réaliser des calculs intensifs sur des flux de données à débit élevé. Dans ce contexte l'utilisation d'une structure de données parcourue séquentiellement au moyen d'un itérateur est encouragée par rapport à une structure de données adressable de manière aléatoire.

L'API SOL contient des classes générales – comme les tableaux, les entrées-sorties, ...– et des classes

[3] En JAVA une classe est implicitement construite par le compilateur à la rencontre d'une déclaration de tableau.

spécialisées pour un domaine d'application – par exemple HWsequence pour la comparaison de séquences (cf. paragraphe 2.2). Certaines méthodes de ces classes prédéfinies exploitent directement des opérateurs matériels dédiés.

2.2. Le parallélisme de classe

Le modèle de programmation du langage SOL est séquentiel mais implicitement parallèle. Le grain du parallélisme sous contrôle du programmeur est assez élevé ; il se situe au niveau tâche (ou processus). À chaque classe de l'application correspond un flot d'exécution (séquentiel) indépendant qui s'exécute sur un processeur de la ROOM. Une classe doit être considérée comme une entité active et indépendante des autres classes, y compris de sa classe mère.

Une classe ne mémorise que l'état de ses objets. Mais elle peut contenir des références vers des objets d'autres classes ou des variables d'un type primitif. Pratiquement, la sémantique opérationnelle d'une déclaration de variable prévoit les trois cas suivants.

- La variable est d'un type primitif (int , char ,...). Elle sera mémorisée dans le processeur associé à la classe où elle est déclarée. Toutes les opérations sur cette variable auront lieu dans le processeur associé à cette classe.
- La variable contient une instance de la classe où elle est déclarée. Elle est gérée (mémorisation de l'état et opérations) entièrement par le processeur de cette classe.
- La variable contient une instance d'une classe différente de la classe où elle est déclarée. Le processeur associé à la classe qui contient la déclaration mémorise uniquement une référence vers l'instance ; il communique cette référence au processeur associé à la classe de l'instance qui mémorise l'état de l'objet et effectue toutes les opérations relatives à cet objet.

Les opérations appliquées aux types primitifs et aux objets de chaque classe étant effectuées dans des processeurs différents, on obtient à l'exécution un parallélisme au niveau classe, que nous appelons parallélisme de classe. Le découpage en classe étant de la responsabilité du programmeur, celui ci dispose d'un moyen simple pour exprimer le recouvrement entre plusieurs tâches sans se soucier de leur synchronisation.

Exemple

La figure 1 contient une modélisation simplifiée de notre exemple de comparaison de séquences. L'application est structurée en cinq classes, dont trois sont liées par une relation d'héritage : la classe utilisateur Sequence contenant l'algorithme de Smith et Waterman (méthode compare) est une sous-classe (hérite de) de la classe prédéfinie HWsequence. La classe HWsequence contient un opérateur matériel de comparaison (méthodes fill et compute) ; c'est une sous-classe de la classe prédéfinie Array<char>. La classe Array<char> contient toutes les opérations relatives à des tableaux de caractères, en particulier les méthodes de parcours séquentiel (méthodes hasNext et getNext).

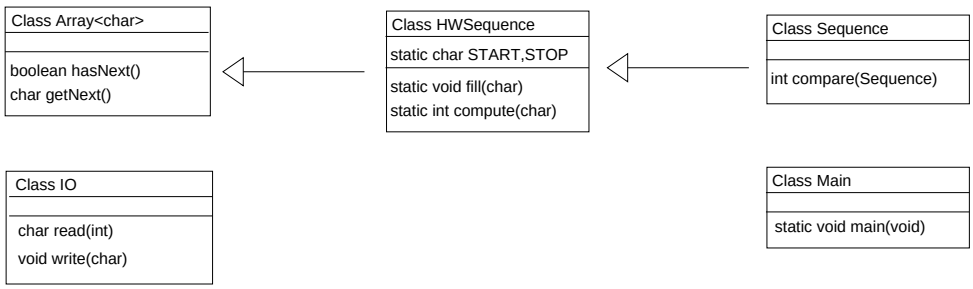

FIG. 1 – Diagramme UML de l'application de comparaison de séquences

Ce découpage permet de réaliser en recouvrement les tâches de calcul (dans la classe HWsequence) et de parcours de la structure de données (dans la classe Array<char>) et d'entrées-sorties (dans la classe IO). Il est généralisable à de nombreuses applications de calculs intensifs sur des flux de données.

3. Le modèle d'exécution

La cible du compilateur du langage SOL est une machine ROOM composée de processeurs d'objet (*PO*) et d'un réseau de communication. L'ensemble est un système parallèle qui supporte le parallélisme de classe. La figure reffig :archi-room contient le schéma d'une ROOM pour l'application de comparaison de séquences. On distingue dans ce schéma les cinq *PO* correspondants aux cinq classes de l'application et un réseau de communication spécifique.

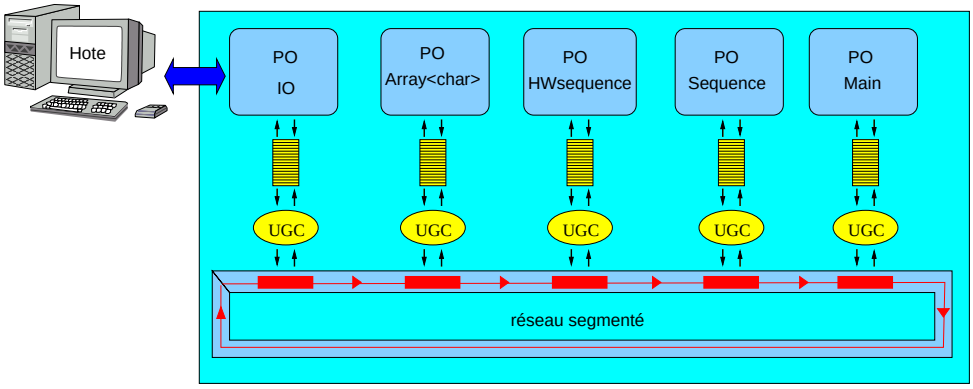

FIG. 2 – Une ROOM pour la comparaison de séquences

Les *PO* sont des processeurs de type RISC dotés de quelques instructions particulières :

Des instructions de gestion d'une mémoire d'objet pour l'allocation d'objets, la lecture et l'écriture d'attributs. La gestion de cette mémoire est simple du fait de la taille fixe et unique des objets alloués dans chaque *PO*[4]. Les accès aux attributs sont réalisés par un adressage basé standard. La taille des objets de chaque classe et la taille des champs sont des paramètres déterminés à la compilation et communiqués au début de l'exécution aux unités de gestion mémoire de chaque *PO*.

Des instructions de calcul paramétrées pour activer les opérateurs spécialisés des *PO* associés aux classes prédéfinies. Ces *PO* contiennent soit des opérateurs matériels généraux (par exemple pour les entrées-sorties avec l'hôte dans le *PO* IO, ou pour la mémorisation et le parcours de tableau dans Array, soit des opérateurs dédiés à un domaine d'application (par exemple pour les comparaisons de séquence en bio-informatique). A noter que les *PO* associés aux classes utilisateurs disposent pour leur part des opérateurs disponibles dans une unité arithmétique et logique standard (pour des opérations sur des types primitifs).

Des instructions de communication pour que les échanges de données(valeurs ou références) entre *PO*. Une instruction de lecture, resp. une instruction d'écriture, opère sur une file d'attente en entrée, resp. en sortie. Chaque *PO* possède une adresse. L'unité d'échange est le mot, précédé de l'adresse de l'émetteur et de celle du destinataire. L'écriture d'un mot n'est pas bloquante si la file de sortie n'est pas pleine ; la lecture d'un mot en provenance d'un autre *PO* est bloquante si la file est vide ou si elle ne contient aucun mot en provenance du *PO* voulu.

Les communications inter-*PO* sont supportées par un réseau segmenté en anneau qui fonctionne de manière synchrone – à la manière d'un train de wagons – à son propre rythme et inter-agit avec chaque *PO* par l'intermédiaire de leur unité de gestion des communications (UGC sur le schéma).

[4] Le caractère fixe de la taille des objets évite tous les problèmes classiques de fragmentation que l'on rencontre habituellement dans la gestion d'un tas d'objet.

4. La compilation

Le processus de compilation, depuis le programme source écrit dans le langage SOL, jusqu'au code interprétable par une ROOM se décompose dans les quatre phases suivantes.

1. La phase d'analyse : le programme source est analysé syntaxiquement et sémantiquement ; un code intermédiaire (séquentiel) équivalent est produit (si le programme source est correct).
2. La phase de répartition : le code intermédiaire est réparti en n codes intermédiaires décrivant n processus séquentiels communiquants ; n étant le nombre de classes présentes dans le programme source.
3. La phase d'optimisation : le code intermédiaire de chaque processus communiquant est optimisé localement (séparément) et globalement.
4. La phase de production : le code cible de chaque PO d'une ROOM est généré à partir du code intermédiaire de chaque processus communiquant.

Seules les phases numérotées 2 et 3 ont nécessité le développement ou l'adaptation de nouvelles techniques de compilation. L'algorithme utilisé dans la phase de répartition est donné dans le paragraphe 4.1. L'optimisation globale des processus communiquants est décrite dans le paragraphe 4.2.

4.1. La phase de répartition

À l'issue de la phase d'analyse le code source est traduit dans un code intermédiaire dont le niveau d'expressivité est comparable à celui de la machine virtuelle Java (JVM). Chaque instruction est un quadruple composé d'un code opération, de deux opérandes sources et d'une opérande destination. Les codes opérations reflètent les instructions élémentaires d'un *PO*. Par ailleurs les informations sur la classe d'appartenance et le type associées aux opérandes sont conservées pour la phase de répartition du code.
La répartition du code consiste à distribuer le code intermédiaire issu de la phase d'analyse entre n processus communiquants, chaque processus gérant les attributs et appliquant les méthodes des objets d'une seule classe du programme source. Le code des processus communiquants est représenté à l'aide du même code intermédiaire augmenté des instructions d'envoi, de diffusion et de réception de valeurs.
La répartition du code se décompose en deux parties : le calcul de l'ensemble des *PO* impliqués par chaque instruction, suivi de la génération du code pour chaque *PO*.

4.1.1. Calcul des ensembles de *PO*

L'algorithme de calcul de l'ensemble des *PO* concernés par chaque instruction du code intermédiaire est réalisé sur le graphe de flot de contrôle (*CFG*) de chaque méthode du code intermédiaire séquentiel. Le *CFG* est une structure de données couramment utilisée dans les optimiseurs de code [1] : c'est un graphe dont les sommets contiennent les blocs de base du programme et les arcs représentent les branchements (conditionnels et inconditionnels) possibles entre bloc ; un bloc de base est une suite d'instructions sans branchement, sauf éventuellement la dernière. Globalement le calcul des ensembles de *PO* se décompose en deux étapes :

1. On effectue une traversée en profondeur du *CFG* et pour chaque instruction rencontrée, autre qu'un branchement ou un appel de méthode, on calcule l'ensemble des *PO* concernés par l'instruction ; il s'agit de l'union des *PO* de chaque opérande. Le *PO* associé à une opérande dépend de la nature de l'opérande, par exemple :
   - le *PO* d'une variable référençant un objet est le *PO* associé à la classe (au type) de l'objet,
   - le *PO* d'une variable ou d'une constante de type primitif est le *PO* associé à la classe où est déclarée la variable ou la constante...
2. À chaque instruction de branchement et d'appel de méthode on associe l'union des ensembles de *PO* des instructions présentes dans le bloc de base (cas d'un branchement) ou dans la méthode (cas d'un appel) ciblée par le branchement.

Il se peut qu'au cours de la seconde étape de nouveaux *PO* soient associés au bloc courant. Comme le bloc courant peut lui-même être la cible d'un branchement ou d'un appel de méthode, il faut réitérer la seconde étape jusqu'à obtention d'un point fixe (aucun *PO* n'est plus rajouté dans aucun bloc).

4.1.2. Génération du code des *PO*

À l'issue du calcul on dispose, pour chaque instruction, de l'ensemble des *PO* concernés, c-à-d. ceux qui auront à effectuer l'instruction ou à y contribuer (par l'envoi ou la réception d'une valeur). La phase de génération du code intermédiaire de chaque *PO* consiste ensuite à parcourir les instructions du code intermédiaire séquentiel pour répartir les instructions dans chaque *PO* concerné, en ajoutant quand cela est nécessaire (cas d'instruction ayant des opérandes appartenant à des *PO* différents) des instructions d'envoi et de réception de données.

Plus précisément on donne à la suite le détail de l'algorithme de génération pour trois instructions représentatives du code intermédiaire.

- Exemple d'une instruction du type <var>=new_object(<class>) :
  C'est l'instruction d'instanciation d'un nouvel objet dans une classe. La variable <var> contient l'adresse de début de la zone mémoire allouée. Par construction du code, cette instruction ne concerne qu'un seul *PO* celui associé à la classe <class> ; elle est donc ajoutée uniquement dans le code du *PO* concerné.
- Exemple d'une instruction du type <var>=get_field(<object>,<field>) :
  C'est l'instruction de lecture d'un attribut d'un objet. Deux cas sont à considérer selon que le *PO* associé à la variable et à l'objet sont identiques ou non. S'ils sont identiques, cette instruction est générée à l'identique dans le *PO* concerné. Si le *PO* associé à la variable (noté PO_d) est distinct de celui associé à l'objet (noté PO_s) alors une instruction <tmp>=get_field(<object>,<field>) est générée dans le code du PO_s suivie d'une instruction snd_data(PO_d,<tmp>) pour envoyer le résultat au PO_d ; une instruction <var>=rcv_data(PO_s) pour la réception du résultat en provenance du PO_s est générée dans le code du PO_d.
- Exemple d'une instruction du type if_goto(<var>,<label_then>,<label_else>) :
  C'est une instruction de branchement conditionnel. Elle concerne le *PO* de la classe (noté PO_s) où est déclarée la variable booléenne <var> et les n *PO* associés aux blocs de base (notés PO_x) étiquetés par <label_then> et <label_else>. Le compilateur génère n instructions snd_flag(PO_x) dans le code du PO_s pour chaque envoi de la valeur de <var> à tous les PO_x. Une instruction de réception correspondante <tmp>=rcv_flag(PO_s) est générée dans le code des PO_x. Enfin une instruction de branchement conditionnel est générée dans le code du PO_d et de chaque PO_x.

La conformité de la répartition obtenue repose sur le fait que l'ordre des opérations appliquées à chaque classe d'objet ainsi que le flot de contrôle initial sont conservés.

4.2. Phase d'optimisation

À l'issue de la phase de répartition on dispose du code intermédiaire de chaque *PO*. On peut appliquer ensuite à chacun d'entre eux les techniques d'optimisation les plus pointues [10]. Mais auparavant il est important de réduire au maximum les synchronisations entre *PO* de manière à accroître l'efficacité globale de la ROOM, c-à-d. le nombre de *PO* actifs simultanément. La réduction des synchronisations passe par la réduction du nombre de communications, principalement dans les itérations.

Nous avons adapté les techniques de déplacement du code invariant dans les itérations au problème spécifique des instructions de communications. Une communication est considérée comme redondante et candidate à la suppression lorsque la valeur échangée est invariante dans l'itération. Un cas typique est celui de l'adresse d'un objet sur lequel une opération (accès à un attribut, méthode appliquée,...) est effectuée à chaque itération : le *PO* qui gère la classe de l'objet référencé n'a pas besoin de recevoir son adresse à chaque itération. Dans ce cas l'adresse de l'objet n'est communiquée qu'une seule fois, avant l'itération. Bien sur, la communication doit être déplacée à la fois dans le code de l'émetteur et dans celui du récepteur, ce qui implique de conserver un lien entre les instructions d'émission et de réception des deux *PO*.

En dehors des itérations il existe de nombreuses communications redondantes. Pour les détecter un historique des variables échangées est associé à chaque couple de *PO*. À la rencontre d'une instruction d'envoi de donnée, l'optimiseur examine l'historique des échanges entre les deux *PO* concernés et supprime la communication (dans les deux *PO*) si la variable qui possède la valeur à envoyer apparait dans l'historique et si elle n'a pas été modifiée entre temps[5].

[5] Déterminer si le contenu d'une variable a été modifiée est un problème classique en optimisation qui nécessite de recourir à des techniques lourdes – mais parfaitement connues – d'analyse du flot de données.

5. Conclusion

Nous avons montré dans cet article comment il était possible d'exploiter une autre forme de concurrence dans un langage à objets – le parallélisme de classe – pour résoudre le problème de parallélisation d'une application dans un composant *FPGA*. Nous avons développés les techniques de compilation et d'optimisation qui rende implicite l'expression de la distribution et totalement transparent le contrôle de la concurrence. Un compilateur a été réalisé et valide l'ensemble de ces techniques ; il génère du code pour des processeurs RISC disponibles sous forme d'IP [3]. Un composant de la série Virtex II peut contenir une quarantaine de processeurs de ce type.
La synthèse architecturale des différents éléments d'une ROOM n'est pas totalement finalisée et ne nous a pas permis de réaliser de réels tests de performance ; néanmoins, nos évaluations nous laissent envisager un facteur d'accélération d'au moins un ordre de grandeur par rapport à un microprocesseur généraliste dans le cas de l'application de comparaison de séquences présentée dans l'article.
Par rapport aux objectifs cités en début d'article, nous avons apporté une solution à l'exploitation du parallélisme présent sous deux formes dans une application : le recouvrement de tâches par le parallélisme inter-classe et la spécialisation des opérateurs par l'utilisation de classes prédéfinies. Nous projetons d'étendre notre modèle de programmation avec l'expression du parallélisme de données. Reprenant l'exemple de la comparaison de séquences, il s'agit de pouvoir exprimer l'exécution simultanée de plusieurs instances des classes Sequence et HWsequence. Beaucoup d'extensions des langages à objets pour gérer ainsi le parallélisme et la distribution des données ont été proposées. Leur intégration dans le contexte du parallélisme de classe et leur support dans le modèle d'exécution vont faire l'objet d'une prochaine évaluation dans le contexte des grappes de *PC*.
Notons enfin que le parallélisme de classe et les mécanismes de compilation associés sont applicables à d'autres cibles que les *FPGA*, comme les circuits multiprocesseurs, dont l'émergence des systèmes enfouis (SoC) laisse présager un développement rapide.

Bibliographie

1. A. Aho, R. Sethi, et J. Ullman. Compilateurs : principes, techniques et outils. InterEditions, 1991.
2. J-P. Briot et R. Guerraoui. Objets pour la programmation parallèle et répartie : intérêts, évolutions et tendances. TSI, 15 :765–800, 1996.
3. J. Gray. Designing a Simple FPGA-Optimized RISC CPU and System-on-a-Chip. DesignCon 2001, 2001. http ://www.fpgacpu.org/index.html.
4. P. Guerdoux-Jamet et D. Lavenier. Systolic filter for fast dna similarity search. ASAP'95, International Conference on Application SpecificArray Processors, Strasbourg, France, 1995.
5. R. Guerraoui. Les langages concurrents à objets. TSI, 8 :945–972, 1995.
6. D.T. Hoang. Searchnig genetic databases on splash2. IEEE Workshop on FPGAs for custom computing machines, Napa, California, 1993.
7. Dominique Lavenier. Samba : Systolic accelerator for molecular biological applications. Rapport Technique 2845, INRIA, 1996.
8. E. Lemoine, J. Quinqueton, et J. Sallantin. High speed pattern matchnig in genetic data base with reconfigurable hardware. Second International Conference on Intelligent System for Molecular Biology, 1994.
9. E. Mosanya. A Reconfigurable Processor For Biomolecular Sequence Processing. PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 1999.
10. S. Muchnick. Advanced Compiler Design Implementation. Morgan Kaufmann, 1997.
11. F. Raimbault et D. Lavenier. ROOM : des machines reconfigurables orientées objet. 8ième Symposium en Architectures Nouvelles de Machines (Sympa), 2002.
12. F. Raimbault et D. Lavenier. ROOM : des machines reconfigurables orientées objet pour les applications spécifiques. TSI, 2003. À paraître (Rapport de Recherche INRIA 4588).
13. S. Rubini et D. Lavenier. Les architectures reconfigurables. Calculateurs Parallèles, 9(1), 1997.
14. T.F. Smith et M.S. Waterman. Identification of common molecular subsequences. Journal of Molecular Biology, 147 :195–197, 1981.
15. G. Snider, B. Shackleford, et R. Carter. Attacking the semantic gap between application programming languages and configurable hardware. FPGA 2001, pages 115–124, 2001.