HAL
open science

ROOM : reconfigurable object-oriented machines for specific applications

Frédéric Raimbault, Dominique Lavenier

To cite this version:

Frédéric Raimbault, Dominique Lavenier. ROOM : reconfigurable object-oriented machines for specific applications. [Research Report] RR-4588, INRIA. 2002. inria-00071997

HAL Id: inria-00071997

https://inria.hal.science/inria-00071997

Submitted on 23 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

ROOM: reconfigurable object-oriented machines for specific applications

Frédéric Raimbault, Dominique Lavenier,

N°4588

Octobre 2002

THÈME 3

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--4588--FR+ENG

ROOM: reconfigurable object-oriented machines for specific applications

Frédéric Raimbault, * Dominique Lavenier, †

Thème 3 — Interaction homme-machine,
images, données, connaissances
Projet Symbiose

Rapport de recherche n°4588 — Octobre 2002 — 19 pages

Abstract: In this article, an object-oriented approach to *program* applications onto reconfigurable hardware is presented. The underlying runtime support is a multiprocessor structure, called a ROOM (Reconfigurable Object-Oriented Machine). It is composed of a number of *object processors* equal to the number of classes required by the application. Some of these object processors can be tailored to the application domain to achieve high performance. We show how this approach facilitates the mapping of applications onto FPGA components. We also give preliminary results based on a pattern-matching algorithm example.

Key-words: architecture, reconfigurable, object, FPGA, parallelism, genomic

(Résumé : tsvp)

* Laboratoire *VALORIA*, (en délégation CNRS à l'IRISA), Université de Bretagne Sud, Campus de Tohannic, 56000 Vannes, frederic.raimbault@univ-ubs.fr

† Laboratoire *IRISA*, Campus de Beaulieu, 35042 Rennes Cedex, dominique.lavenier@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

ROOM :
des machines reconfigurables orientées objet pour les applications spécifiques

Résumé : Les travaux présentés dans cet article ont pour objectif de faciliter l'implantation d'accélérateurs de calcul dans de la logique reconfigurable. Nous proposons une nouvelle approche basée sur le concept de ROOM (machine reconfigurable orientée objet). L'idée fondatrice consiste à décrire l'application dans un langage orienté objet et à implanter chaque classe de manière indépendante. L'architecture dérivée est une machine parallèle qui est composée d'autant de *processeurs d'objets* qu'il y a de classes présentes dans l'application. L'article expose les motivations de cette approche qui réconcilie synthèse d'opérateurs dédiés et microprogammation du contrôle. Une application en génomique est présentée en détail. Les aspects liés à la programmation et à la compilation de ce modèle de machine sont examinés et les résultats de nos premières expérimentations sont analysés.

Mots-clé : architecture, reconfigurable, objet, FPGA, parallélisme, génomique.

1 Introduction

Malgré la progression constante des performances des microprocesseurs, les accélérateurs matériels restent d'actualité dans un grand nombre de domaines. Dans cet article, nous nous intéressons aux accélérateurs traitant un flux continu de données, et plus spécifiquement à leur conception.

Le domaine d'applications visé est la recherche d'information par le contenu dans de grandes bases de données. Un exemple d'actualité est la fouille des données génomiques où l'on cherche à extraire des séquences d'ADN ou des séquences protéiques en fonction de leur similarité avec une séquence de référence. Comme il n'y a – à l'heure actuelle – aucune possibilité d'organiser ces données d'après leur contenu, la seule solution est de parcourir systématiquement la totalité des informations. Un autre exemple est celui de la recherche d'images sur la base de similarités avec une image requête.

L'usage d'un microprocesseur standard pour ce type de traitement est certes possible, mais non optimal. Par exemple, le mécanisme de mémoire cache est inutile pour gérer un flux de données ; les opérateurs flottants sont également peu exploités lorsqu'il s'agit de traiter ce type d'information ; les calculs sont en général relativement simples, sur une dynamique réduite (très inférieure à 64 bits), mais très coûteux en terme de complexité. La ressource silicium est donc mal exploitée. On peut alors basculer sur des choix de type DSP (*Digital Signal Processor*), ASIP (*Application Specifique Instruction Set Processor*), FPGA (circuit reconfigurable) ou VLSI (circuit à façon).

Par rapport à un circuit dédié (VLSI) qui constitue incontestablement la solution la plus optimale (en terme de vitesse, de ressource, de consommation, *etc.*), le circuit reconfigurable [18] est une alternative intéressante car il ne fige pas définitivement sur silicium une solution. De plus, les progrès technologiques offrent maintenant, pour ces composants, des capacités d'intégration équivalentes à quelques millions de portes, ce qui permet l'implantation d'accélérateur complet (et complexe!) dans un seul composant.

Mais aujourd'hui, porter une application sur une structure reconfigurable – en l'occurence, un composant FPGA – passe essentiellement par la définition d'une architecture à l'aide d'un langage de description matériel (le langage VHDL, par exemple). L'architecture doit être spécifiée dans ses moindres détails et de manière extrêmement précise. Cette approche oblige une transcription de l'application en terme d'opérateurs, de modules ou de fonctions matérielles qu'il faut ensuite interconnecter, controler, synchroniser, *etc.*

En pratique, la conception des opérateurs dédiés est relativement aisée. A partir d'une description de haut niveau, de nombreux outils synthétisent efficacement des opérations complexes. Les bibliothèques d'IP (Intellectual Properties) apportent également une aide précieuse dans le cycle de développement : ce sont généralement des modules complexes (interface PCI, coeur de microprocesseur, etc.), mis au point par des spécialistes, et que l'on peut réutiliser dans différents contextes.

Par contre, les difficultés surviennent lorsqu'il faut assembler cet ensemble de briques de base et s'assurer que les fonctionalités répondent au cahier des charges. Le contrôle d'un tel système est assurément une tâche difficile car l'ensemble évolue de manière concurrente, et doit généralement répondre à des stimulis externes non forcément synchrones avec le cadencement interne. L'expérience montre que la partie contrôle et synchronisation d'un système numérique représente une part très importante dans la durée du cycle de conception. Elle s'accompagne de simulations intensives pour vérifier le plus exhaustivement possible son comportement. Il s'avère en général plus aisé de réaliser ce contrôle par microcode, sans perte notable de performance.

Le concept de ROOM (*Reconfigurable Object Oriented Machine*) s'appuie sur cette analyse pour proposer un environnement matériel et logiciel d'aide à la conception d'accélérateurs matériels. L'objectif est de s'affranchir au maximum des problèmes de conception rencontrés lors de la mise au

point, en particulier lorsqu'il s'agit d'interfacer le coeur de l'accélérateur – la partie matérielle originale – avec le reste du système (entrées/sorties, contrôle, initialisation, *etc.*). Notre approche repose essentiellement sur les choix suivants :

Une description en terme d'objets de l'application. Il ne s'agit pas d'utiliser un langage objet comme support de description matérielle (ex. JHDL), mais bel et bien de décrire l'algorithme réalisé par l'accélérateur sur la base d'une spécification objet. A ce niveau, aucune connaissance en architecture de machine n'est requise. La ou les parties originales du traitement sont modélisées par des classes spécifiques dont les méthodes sont appliquées de manière classique à des objets définis par l'utilisateur.

Une description architecturale minimale. Les performances d'un accélérateur proviennent souvent d'une structure originale qui ne peut être implémentée que sur la base d'une spécification matérielle précise. Cette partie relève des compétences d'un architecte. Il a toute liberté pour spécifier son architecture, mais doit néanmoins respecter quelques contraintes d'interface. C'est ici que s'effectue le lien entre classes spécifiques (manipulées dans la description de l'application) et opérateurs spécifiques (implantés dans le matériel).

Un modèle d'exécution unique pour la gamme d'applications visée. Il s'agit d'une structure parallèle où chaque classe est implantée sur un processeur distinct. Il y a donc autant de processeurs que de classes spécifiées dans l'algorithme. Suivant l'application, leur nombre varie mais l'organisation (le modèle d'exécution) reste identique.

Dans ce schéma, la conception d'un accélérateur consiste d'abord à décrire matériellement (i.e. à l'aide d'un langage de type VHDL) des opérateurs non conventionnels justifiant la conception d'une machine spécialisée ; puis, à décrire leur usage – via leur classe associée - au sein de l'application. L'intérêt de cette approche est qu'elle se focalise sur les parties intéressantes et *productives* de l'accélérateur, à savoir le design d'opérateurs ou d'unités de traitement relatifs à un calcul intensif. Les aspects assemblage et synchronisation sont cachés et automatiquement traités pendant le processus de synthèse/compilation.

L'approche objet a déjà été étudiée pour implanter des applications spécifiques sur FPGA. Mais le plus souvent – par exemple dans [6]– il s'agit de s'appuyer sur la méthodologie de conception objet pour répartir (en partie automatiquement) la mise en œuvre entre plusieurs composants matériels et logiciels. Nous n'avons identifié qu'une autre tentative de réalisation d'une machine parallèle programmable dans un seul composant FPGA. Il s'agit de SoCrates [5], un multi-processeur à mémoire distribuée partagée basée sur un bus partagé sur lequel sont connectés deux processeurs (clones ARM) et un contrôleur global temps réel. A la différence d'une ROOM le contrôle de l'exécution est centralisé et l'ensemble n'est ni extensible ni paramétrable.

Une architecture plus proche de la ROOM est celle de la machine RAW [22]. C'est une machine distribuée programmable intégrée sur un seul chip. Les noeuds de la machine contiennent un processeur élémentaire de type RISC, de la logique reconfigurable, un peu de mémoire, et une unité d'interconnexion. L'originalité de cette architecture réside dans la configuration des chemins de données par le compilateur [3]. Il serait envisageable de mettre en oeuvre notre support d'exécution sur une telle architecture ; le facteur limitant semble être le peu de ressources reconfigurables dans chaque noeud. Les autres projets relevés en architecture de machine autour des FPGAs visent soit la mise en oeuvre d'un processeur unique à des fins de simulation – par exemple [2] – soit l'association avec un microprocesseur hôte comme accélérateur de calcul – par exemple [10] – soit l'interconnexion de plusieurs FPGA pour former une machine spécialisée [21].

La construction d'une machine dédiée à l'exécution d'un langage à objets a été envisagée dès le début des années 1980. Il s'agissait d'accélérer l'exécution de la machine virtuelle du langage Small-

talk [20]. Plus récemment plusieurs entreprises proposent des microprocesseurs dédiés à l'exécution du bytecode Java. Toutefois les accélérations obtenues permettent d'atteindre au mieux les performances d'un code natif dont les sources sont écrits en C. De plus, il s'agit de systèmes fermés, sans adaptation possible aux applications. Il a été aussi démontré dans [11] qu'il est possible d'implanter une JVM (Java Virtual Machine) dans un FPGA. Mais la mise en oeuvre choisie – une implantation directe de l'interface abstraite d'une JVM – ne donne pas de bonnes performances et ne permet pas, par ailleurs, de spécialisation.

Dans le domaine de la compilation pour les architectures reconfigurables, les principaux travaux de recherche visent à produire automatiquement le fichier de configuration du FPGA à partir d'un programme impératif, par exemple un *C* parallèle [7]. Les inconvénients de ce type d'approche sont d'une part le temps très important de compilation et d'autre part la difficulté d'une programmation efficace car la parallélisation est entièrement à la charge du programmeur. D'autres recherches visent la synthèse automatique de descriptions de très haut niveau, par exemple des équations récurrentes [16]. Les problèmes que l'on sait traiter par ces approches sont limités à des traitements réguliers et font abstraction, le plus souvent, des aspects extérieurs aux calculs, comme l'alimentation en données. Cependant ce genre d'outil pourrait être utilisé comme complément dans la programmation d'une ROOM, pour spécifier et générer les opérateurs des processeurs d'objets spécialisés. Nous ne traitons pas de ce problème dans cet article. Nous nous focalisons sur l'assemblage de composants élémentaire et leur exploitation dans un environnement logiciel.

Le reste de cet article est organisé de la manière suivante. Dans le paragraphe 2 nous présentons d'abord le concept de ROOM. Puis, dans le paragraphe 3, nous nous étudions les aspects liés à la programmation et à la compilation. Le paragraphe 4 contient une description des premières expérimentations et leur résultats. Le paragraphe 5 conclue cet article et propose plusieurs voies d'amélioration et de recherche.

2 Le concept de ROOM

La méthode que nous proposons pour mettre en œuvre une application dans de la logique reconfigurable se décompose en trois phases. Premièrement, l'application est décrite dans un langage de programmation objet. Deuxièmement, un support d'exécution programmable, spécifique au domaine d'application, est défini et implanté dans le composant reconfigurable. Troisièmement, du microcode pour le support d'exécution est généré par compilation du programme de l'application.

Par rapport au processus de développement standard d'un programme, la phase de définition du support d'exécution est celle qui exploite la flexibilité offerte par la technologie reconfigurable : le support d'exécution est taillé sur mesure pour accueillir le code généré pour l'application et accélérer les parties de calcul intensif. Ce support d'exécution respecte un modèle général que nous nommons une ROOM. Cette première partie de l'article contient une présentation générale de la ROOM, son principe de fonctionnement, ses propriétés et un exemple significatif que nous reprendrons tout au long de l'article pour expliciter nos propos.

2.1 Principe de la ROOM

L'idée, à l'origine de la ROOM, est d'appliquer la méthodologie de la programmation objet à l'architecture de la machine elle-même. Le support d'exécution obtenu repose sur le parallèle suivant :

- la programmation orientée objet consiste à découper une application en classes d'objets, prédéfinies ou construites par l'utilisateur : pour chaque application une ROOM est constituée d'autant de *processeurs d'objets* que l'application contient de classes ;
- chaque classe contient les fonctions (ou méthodes) et les variables (ou attributs) des objets qu'elle instancie : chaque *processeur d'objets* mémorise l'état des objets qu'il a créé et n'effectue que les opérations applicables sur ces objets ;
- idéalement, les données d'un objet ne sont accessibles et modifiées que par les méthodes (publiques) de sa classe : les opérateurs de calculs contenus dans un *processeur d'objets* n'affectent que la valeur des objets créés dans ce processeur.

Nous appliquons ce principe d'abstraction de manière très stricte à tous les objets, même les plus élémentaires, comme les entiers ou les caractères[1]. Toute opération sur des entiers, par exemple, ne peut avoir lieu que dans le processeur d'objets des entiers[2]. Les processeurs d'objets n'exécutent donc pas la totalité du programme, mais uniquement les parties de l'application qui les concernent.

De manière générale, un programme complet est composé de classes définies par l'utilisateur qui font appel à des objets de classes prédéfinies. De manière similaire, une ROOM est composée de processeurs d'objets prédéfinis et de processeurs d'objets définis par l'utilisateur. Tous ces processeurs sont interconnectés et l'ensemble constitue une machine dont le schéma général ressemble à celui d'une machine parallèle de type MIMD à mémoire distribuée.

Toutefois, les machines parallèles sont connues pour être difficiles à programmer. En particulier, la gestion explicite par le programmeur des communications et des synchronisations inter-processeurs est fastidieuse et source d'erreur. C'est pourquoi le modèle de programmation d'une ROOM est séquentiel (langage à objets séquentiel) et c'est un compilateur qui prend en charge la distribution du code pour chaque processeur d'objets ainsi que leur synchronisation (voir paragraphe 3.4).

2.2 Propriétés de la ROOM

L'intérêt bien connu de l'abstraction par les données est d'utiliser des objets sans connaître leur implémentation et de remplacer la mise en oeuvre d'une classe par une autre sans remettre en cause le reste du programme. L'application de cette propriété au support d'exécution a comme première conséquence l'adaptation de la machine à l'application : ne sont présentes à l'exécution que les processeurs d'objets des classes définies dans le programme source.

Une deuxième conséquence est la spécialisation possible et indépendante de chaque classe. Cette souplesse est particulièrement intéressante pour le prototypage ou la mise au point progressive d'un accélérateur matériel dédié à une application. On peut se contenter de spécialiser la machine pour les parties critiques en terme de performances, le reste de l'application s'accommodant d'une mise en oeuvre standard par programmation.

Une autre conséquence de la répartition spatiale des opérations suivant la classe de leur opérande est le parallélisme potentiel à l'exécution. Ce parallélisme est implicitement limité aux activités inter-processeurs mais n'exclut pas l'exploitation d'un parallélisme de données, par exemple sous la forme d'un opérateur systolique ou vectoriel, à l'intérieur d'un processeur d'objets spécialisé. Le parallélisme implicite lié aux classes – ou parallélisme au niveau classe, par analogie à parallélisme au niveau instruction – ne demande aucun effort particulier au programmeur et ne soulève pas de problème nouveau en compilation. Le programmeur découpe de manière naturelle son application

[1] On ne considère pas de types primitifs particuliers au niveau de l'implémentation comme en Java ou en SmallTalk.

[2] Il est possible, pour des raisons de performance, d'utiliser la notion de classe interne pour limiter les goulots d'étranglement que provoquerait, par exemple, une classe unique pour gérer tous les entiers du programme.

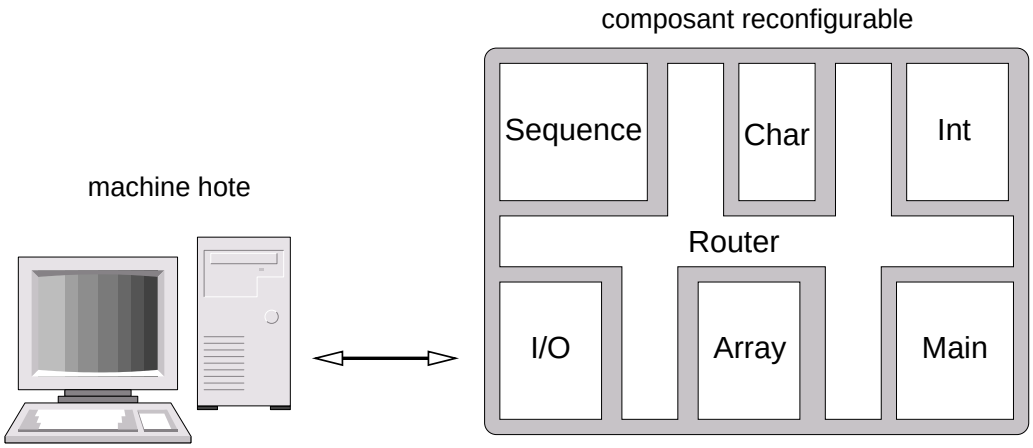

FIG. 1 – Architecture d'une ROOM pour la comparaison de séquences d'ADN

en classes d'objets ; ce découpage peut éventuellement être guidé par la spécialisation future de la machine pour certaines parties de l'application.

2.3 Un exemple d'application

L'exemple développé dans ce paragraphe a pour but d'illustrer nos propos. Il a été choisi pour mettre en exergue la spécialisation d'une application coûteuse en temps de calcul : la comparaison de chaînes de caractères. Les domaines d'applications potentiels vont de la fouille des données textuelles à la comparaison de documents entiers en passant par l'exploration des banques génomiques. Dans tous les cas, l'objectif est identique : mettre en évidence des zones d'informations qui se ressemblent. D'un point de vue informatique, qu'il s'agisse de manipuler des mots, des paragraphes ou des gènes, les opérations de base sont les mêmes. Avec la ROOM ces traitements sont pris en charge par une classe spécialisée. Seuls, la taille des éléments, les fonctions d'évaluation ou les critères de sélection changent d'un domaine d'application à l'autre.

Les besoins en puissance de calcul caractérisent ces applications car il faut souvent explorer un volume de données conséquent sur lequel on applique des traitements coûteux. Les techniques d'indexation, qui structurent les bases de données et accélèrent fortement les recherches, ne peuvent être mises en oeuvre car elles reposent sur des ressemblances exactes. Dès lors qu'une certaine approximation est tolérée, l'espace de recherche devient gigantesque et requiert une puissance de calcul énorme. Celle-ci peut être apportée par des opérateurs matériels spécialisés qui prennent en compte la structure des données (des tableaux de caractères) et le parallélisme potentiel des traitements.

Le but n'est pas ici de décrire en détail ces familles d'applications. Elles ont été intensivement étudiées et décrites dans le cadre d'autres travaux de recherche par les auteurs ; pour une mise en oeuvre sur machine parallèle le lecteur peut se référer à [14, 15, 17], et pour une mise en oeuvre dans un composant spécialisé, à [9, 13].

Nous nous focalisons dans cet article sur la conception d'un accélérateur pour la recherche d'une séquence dans une base de données génomique. Dans ce domaine l'algorithme de référence est celui de Smith et Waterman [19]. Il consiste à calculer l'indice de similarité entre la séquence de test et toutes les séquences de la base de données pour retenir la séquence de référence la plus proche. L'algorithme compare deux séquences R et T en réalisant la comparaison deux à deux de chaque élément r_i et t_j des séquences. Plus précisément, le calcul de similarité est un algorithme de programmation dynamique qui repose sur les équations récurrentes suivantes :

$$
\begin{aligned}
& d_{0,j} = 0 \\
& d_{i,0} = 0 \\
& (\forall i)(i > 0), (\forall j)(j > 0)\ d_{i,j} = max \begin{cases} 0 \\ d_{i,j-1} - 1 \\ d_{i-1,j} - 1 \\ d_{i-1,j-1} + \begin{cases} 1 \text{ si } r_i = t_j \\ -1 \text{ si } r_i \neq t_j \end{cases} \end{cases}
\end{aligned} \tag{1}
$$

Dans un langage à objet, cinq classes d'objets sont nécessaires à notre application : une classe `Sequence` pour toutes les opérations relatives aux séquences, une classe `Main` qui contient la partie initialisation, séquencement des calculs et exploitation des résultats, les classes de base `Int` et `Char` qui effectuent les opérations sur les objets élémentaires et une classe `IO` pour les entrées/sorties avec la base de donnée. La figure 1 reprend ces éléments sous forme de processeurs d'objets et contient une représentation schématique de la ROOM. Le seul ajout (systématique) est le composant (classe `Router`) qui gère les communications internes et la synchronisation globale. On peut remarquer que toute application réalisant des calculs sur des séquences utilise la même structure de ROOM : si on souhaite accélérer les traitements, en y intégrant des opérateurs spécifiques, seul le processeur d'objets `Sequence` est à modifier.

3 Programmation et compilation

Nous décrivons dans cette partie les aspects relatifs à la compilation pour une ROOM. Nous commençons par une description de l'ensemble du processus de compilation, suivi d'une présentation succinte du langage de programmation. Les particularités de l'architecture cible sont ensuite résumées et nous expliquons la distribution et la synchronisation des instructions dans les processeurs. Enfin, nous terminons par une discussion sur les moyens à la disposition du programmeur pour exprimer et contrôler le parallélisme de son application.

3.1 Chaîne de compilation pour une ROOM

Compiler un programme pour une ROOM c'est analyser le source écrit dans son langage de programmation (nommé SOL– Simple Object Language) et produire du code pour les différents processeurs d'objets et de routage de la machine cible. Ce processus de compilation, depuis le programme source jusqu'à la ROOM ciblée, est schématisé dans la figure 2.

On y observe que le compilateur prend en entrée un fichier contenant le code source (exprimé dans le langage SOL) mais aussi un fichier contenant l'interface des classes prédéfinies, c-à-d. principalement la signature (nom et types des paramètres) de leur méthodes. Ces deux fichiers permettent au compilateur de procéder à la vérification sémantique complète du programme source (définition des classes, appels de méthode, résolution de noms,...). Si la syntaxe et la sémantique du langage à objets SOL sont respectées, un code intermédiaire séquentiel équivalent est produit. Le niveau d'expression de ce code intermédiaire est comparable à celui du code de la machine virtuelle Java (bytecode JVM).

La seconde phase consiste à répartir les instructions du code intermédiaire dans les programmes qui seront chargés dans les différents processeurs de la machine ROOM et d'ajouter des instructions de communication. Les processeurs se répartissent en trois catégories : les processeurs des classes utilisateurs, les processeurs des classes prédéfinies et le processeur de routage. Ils se distinguent par un jeu d'instructions légèrement différent (voir paragraphe 3.3).

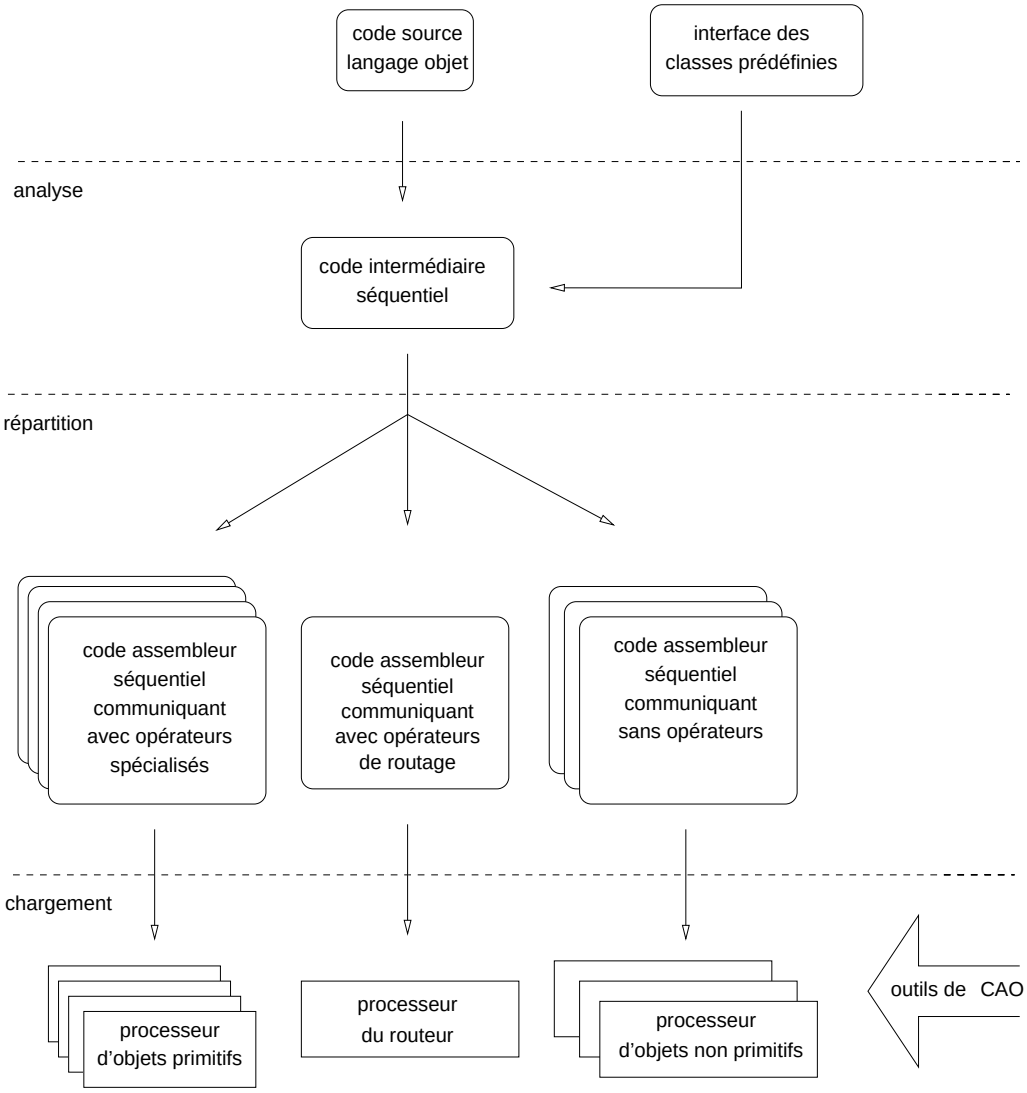

FIG. 2 – La chaîne de compilation pour une ROOM

Le code de chaque processeur produit à l'issue de cette phase pourra alors être chargé dans le processeur correspondant au début de l'exécution. Au préalable, le circuit reconfigurable aura été configuré pour accueillir les processeurs et leur réseau de communication.

La synthèse des processeurs et de leurs opérateurs n'est pas abordée dans cet article. Elle est actuellement effectuée à l'aide des outils de CAO standard pour la plateforme 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 çi peuvent être utilisés pour tout une gamme d'algorithmes, par programmation comme avec un microprocesseur standard.

3.2 Langage de programmation

La programmation d'une application pour une ROOM s'effectue dans un langage de programmation orienté objet, dénommé SOL. Nous énumérons ci dessous les principales différences avec le langage Java dont la syntaxe a été reprise en partie.

- La dichotomie entre valeurs et objets, présente dans Java pour des raisons de performance de la machine virtuelle, a été supprimée. Il n'existe donc pas en SOL de type primitifs[3] (**int**, **char**...) et toutes les valeurs sont contenues dans des objets de classe prédéfinies (Int, Char...).
- Les tableaux à une dimension sont explicitement définis comme des classes[4] génériques qui prennent en paramètre le type des éléments. De fait, la généricité (à la C++) est introduite.
- Les notions de *package*, de *classes abstraites* et d'*interface* de Java n'apportant rien de plus à la structuration matérielle d'une ROOM, ne sont pas implantés dans le langage SOL.
- Les structures de contrôle pour la gestion des *exceptions* ne sont pas supportées pour l'instant pour des raisons de complexité de mise en oeuvre.
- La liaison dynamique des méthodes n'est pas supportée, en attendant de trouver une mise en oeuvre satisfaisante dans la ROOM. Tout appel de méthode est résolu statiquement et traduit comme une simple fonction.

En guise d'illustration, nous donnons dans les figures 3,4 et 5 le programme SOL pour la programmation de l'accélérateur de la comparaison de séquences.

La figure 3 contient le programme SOL de la classe Main qui effectue la comparaison d'une séquence de test avec un flôt de séquences de référence provenant d'une base de données génomique. Les séquences sont des instances de la classe utilisateur Sequence décrite ci-après. Le flôt de données est exprimé à l'aide de la classe prédéfinie IO.

La figure 4 contient la classe Sequence. La classe Sequence est définie comme un sous-type de la classe générique prédéfinie Array – une séquence est un tableau de caractères – auquel on adjoint une méthode qui calcule la similarité entre deux séquences selon l'algorithme de Smith et Waterman décrit au paragraphe 2.3. Dans cette première version, l'algorithme de comparaison est exprimé à l'aide des opérateurs élémentaires sur les entiers, sur les tableaux et sur les caractères.

La figure 5 contient une seconde version de la méthode de comparaison dans laquelle un opérateur systolique est utilisé[5]. Dans ce cas les séquences sont définies comme des instances de la classe prédéfinie HWsequence et la méthode smithWaterman2 sert à adapter l'utilisation de l'opérateur à la taille des séquences. Cet opérateur reprend celui développé dans [8]. C'est un réseau systolique composé de SIZEOP cellules que l'on initialise avec la séquence requête (les deux premières itérations du pro-

[3]Dans Java un type primitif possède une valeur et non une référence vers un objet.

[4]Dans Java une classe est implicitement contruite par le compilateur à la rencontre d'une déclaration de tableau.

[5]Le programme donné est une version simplifi ée et n'est pas optimal ; il est par exemple inutile de charger la séquence de test pour chaque comparaison avec une séquence de référence.

```
class Main
{
    // comparaison d'une sequence test avec toutes les séquences d'une BD
    // mémorisation et affichage de la plus proche trouvée

    public static Void main (){
        IO db= new IO("database.txt",' r' ); // ouverture de la BD en lecture
        Int best_score= 0; // meilleur score actuel
        Sequence best_seq= null; // meilleure séquence trouvée
        Sequence test_seq= new Sequence(IO.stdin);// lecture de la sequence test
        while (! db.eof()){ // parcours de la BD
            Sequence ref_seq = new Sequence(db);// lecture d'une référence
            Int score= test_seq.smithWaterman(ref_seq);// résultat de la comparaison
            if ( score > best_score){ // mémorisation du meilleur score
                best_score= score;
                best_seq= ref_seq;
            }
        }
        // affichage à l ' écran de la séquence la plus proche trouvée
        IO.stdout. write ( best_seq.toString());
    }
}
```

FIG. 3 – Classe Main pour la comparaison de séquences

gramme de la figure 5), puis dans lequel on injecte, caractère par caractère, une des séquences de référence de la base de données (seconde boucle **while** du programme de la figure 5).

Par rapport à la version précédente, les performances sont apportées d'une part par la spécialisation du calcul – une cellule systolique réalise en un cycle d'horloge l'équivalent de l'équation 1 – et, d'autre part, par le parallélisme du réseau systolique – SIZEOP cellules fonctionnent en même temps –.

3.3 Machine cible

La machine cible est perçue par le compilateur du langage SOL comme un multiprocesseur à mémoire distribuée : chaque processeur possède une mémoire locale et échange des données avec les autres via un réseau de communication. A titre d'exemple, la figure 6 contient la schématisation de la machine cible pour la comparaison de séquences.

Le nombre de processeurs est égal au nombre de classes définies par l'utilisateur dans son programme, auquels sont ajoutés les processeurs des classes prédéfinies et le routeur. Les processeurs des classes utilisateurs et ceux des classes prédéfinies ont un jeu d'instruction (de type RISC) commun à une exception près. Les objets gérés par les processeurs des classes utilisateurs contiennent des références vers d'autres objets de classes utilisateurs ou vers des objets de classes prédéfinies. Toute opération de calcul sur une instance (objet) d'une classe utilisateur s'effectue dans les processeurs qui gèrent les classes des objets préfinis qui la compose. De fait le jeu d'instruction des processeurs des classes utilisateurs ne contient pas d'instruction de calcul.

Le processeur du routage a un jeu d'instruction plus spécifique puisqu'il intègre les instructions de communication et de synchronisation entre les processeurs. Ces instructions permettent de réaliser les échanges de données entre les processeurs.

Le choix du protocole de communication et de synchronisation a été fait en privilégiant la simplicité et l'extensibilité de l'architecture. Dans le cas de l'envoi d'une donnée d'un processeur à un autre, l'émetteur place la donnée dans sa file de données en sortie, le routeur déplace cette donnée dans la file de données en entrée du récepteur qui peut ensuite y accéder. Dans le cas de la diffusion d'une condition d'un processeur aux autres processeurs concernés, l'émetteur place la condition dans

```
class Sequence extends Array<Char>
{
    // version sans opérateur spécialisé

    Sequence (IO f){
    // construit une séquence en lisant un fichier

        super(f);
    }

    Int smithWaterman (Sequence S){

    // compare la séquence de test courante avec la séquence de référence S
    // selon l'algorithme de Smith et Waterman

        Int res = 0;                          // résultat
        Int size = this.getLength()+1;   // taille d'une ligne
        Array<Int> lVect = new Array<Int>(size); // ligne précédente
        Array<Int> cVect = new Array<Int>(size); // ligne courante
        Array<Int> tVect = new Array<Int>(size); // ligne temporaire
        // initialisation de la premiere ligne
        lVect.fill (0);
        // calcul de la matrice, ligne par ligne
        S.setIterator (0); // initialise l'itérateur sur S
        while(S.hasNext()){ // parcours de la séquence S
          Char c1= S.getNext();
          cVect.putAt(0,0); // conditions initiales
          Int j = 0; // indice de la colonne précédente
          this.setIterator (0); // initialise l'itérateur sur this
          while(this.hasNext()){ // parcours de la séquence this
              Char c2= this.getNext();
              Int d;
              if ( c1==c2){
                  d= lVect.getAt(j)+1;
              }
              else {
                d = lVect.getAt(j)-1;
              }
              Int h = cVect.getAt(j)-1;
              j ++; // j devient la colonne courante
              Int v = lVect.getAt(j)-1;
              d= h.max(v.max(d.max(0))); // calcul du max
              cVect.putAt(j, d);
              // on retient le max dans res
              res= res.max(d);
          }
          // la ligne courante devient précédente
          lVect = cVect;
          cVect= tVect;
          tVect= lVect;
        }
        return res;
    }
}
```

FIG. 4 – Classe Sequence sans opérateur spécialisé

```
class Sequence extends HWsequence
{
    // version avec un opérateur systolique spécialisé de taille SIZEOP
    // ( suppose que la taille de la séquence de test < taille de l' opérateur)

    Int smithWaterman2 (HWsequence S){
        Int size= this . getLength();
        this . setIterator (0);
        while(this . hasNext()){ // chargement de la séquence de test
            this . set(this . getNext());
        }
        for ( Int i =size; i <HWsequence.SIZEOP; i++){ // complément fictif
            this . unset(HWsequence.NULL_CHAR);
        }
        this . compute(HWsequence.START_CHAR); // initialise le début du calcul
        S.setIterator (0); // initialise l' itérateur sur S
        while ( S.hasNext()){ // parcours de la séquence S
            this . compute(S.getNext()); // cycle de calcul systolique
        }
        for ( Int i = 0; i <HWsequence.SIZEOP−1; i++){ // vidage du réseau
            this . compute(HWsequence.STOP_CHAR);
        }
        return this . compute(HWsequence.STOP_CHAR); // récupération du résultat
    }
```

FIG. 5 – Méthode SmithWatermann avec opérateur spécialisé

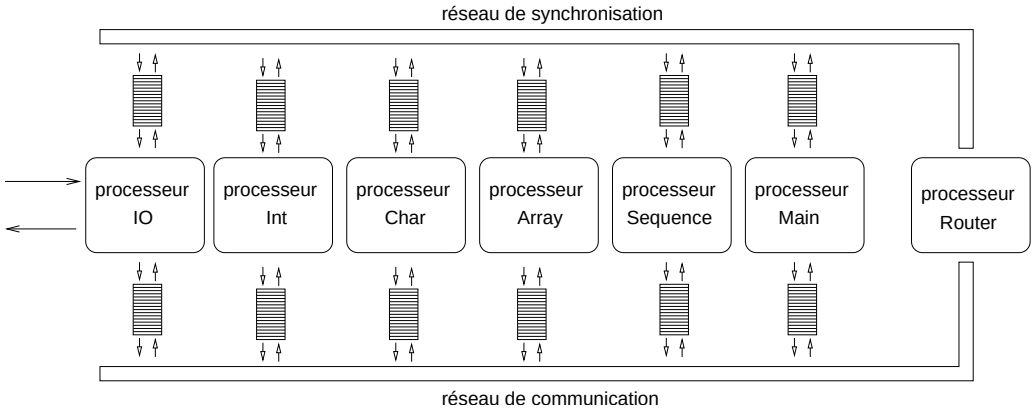

FIG. 6 – Architecture cible pour la comparaison de séquences

sa file de condition en sortie, le routeur duplique cette condition dans la file de données en entrée des récepteurs qui effectueront tous le même saut conditionnel. Ce mécanisme est totalement asynchrone et permet à chaque processeur de progresser de manière concurrente dans son exécution.

3.4 Distribution des instructions

La distribution automatique du code repose sur l'identification de la classe de chaque objet impliqué dans une instruction. La figure 7 illustre les différentes possibilités : dans la partie gauche le code intermédiaire séquentiel est schématisé sous la forme d'une séquence d'instructions et d'un saut conditionnel; chaque instruction est numérotée et marquée par la classe d'objet qu'elle concerne (3 classes nommées A, B et C sont présentes). Dans la partie droite figure le code de chaque processeur d'objets (nommés PO_A, PO_B et PO_C) après la distribution. Les cas suivants peuvent se présenter selon la classe des opérandes impliquées dans une instruction du programme source.

- Une seule classe d'objets est concernée. Il s'agit par exemple d'une instruction d'instanciation (`new`) ou d'un calcul élémentaire. Dans ce cas, le code correspondant est généré uniquement pour le processeur d'objets correspondant. Sur la figure 7 c'est le cas pour les instructions 1 et 8 (classe A), 4 (classe B) et 2,3,5 (classe C).
- Plusieurs classes d'objets sont concernées. Il s'agit par exemple d'une instruction qui nécessite un trans-typage (cast) ou d'un appel de méthode avec passage de paramètres. Dans ce cas, le processeur d'objets sur-typé ou en paramètre envoie sa valeur au processeur d'objets qui l'utilise pour effectuer l'instruction. Sur la figure 7 c'est le cas pour les instructions 7 et 9. Une instruction d'émission, resp. de réception, est ajoutée au code de PO_B, resp. de PO_C, dans le code distribué.
- L'instruction est un test (`if` ou `while`) qui contrôle le flot d'exécution. Dans ce cas, le processeur d'objets qui possède l'opérateur adéquat évalue la condition et la communique à ceux qui sont concernés par le corps de la structure de contrôle. Sur la figure 7 c'est ce qui se produit pour le saut conditionnel de l'instruction 6. La condition est évaluée par le processeur d'objet de la classe A qui transmet le résultat (un booléen) aux processeurs d'objets des classes B et C. Ceux ci contiennent une instruction de réception du booléen, suivi d'une instruction de saut conditionnel (représentées par une seule instruction sur le schéma).

A noter que le code du routeur est généré automatiquement selon le même principe. Il contient une instruction de gestion du réseau de communication à chaque communication inter-processeur et une instruction de gestion du réseau de synchronisation à chaque instruction conditionnelle. Il suit le même flôt de contrôle.

D'un point de vue technique de compilation, la distribution des instructions dans les processeurs d'objets est réalisée en synthétisant un attribut de classe dans l'arbre syntaxique. Cette technique a été appliquée avec succès par l'un des auteurs dans un autre contexte [17]. Elle repose sur le fait que pour préserver la sémantique du programme séquentiel initial il suffit de respecter l'ordre des opérations appliquées à chaque classe d'objet. Le code est donc distribué en autant de flots d'exécution que le programme source contient de classes. La synchronisation entre ces flôts est effectuée au moment des évaluations de conditions dans les structures de contrôle du programme original tel que celà a été décrit dans la figure 7.

3.5 Gestion du parallélisme

A l'intérieur d'une ROOM il existe plusieurs formes de parallélisme exploitables, implicitement ou explicitement, par le programmeur.

La première source de parallélisme provient de l'exécution concurrente des processeurs d'objets. Ce parallélisme est implicite – non géré par le programmeur – mais repose sur la structuration en classes décidée à la programmation. Le programmeur peut donc influencer cette forme de parallélisme de contrôle (ou de tâche) de manière à provoquer, par exemple, un recouvrement entre le processus d'alimentation en données et le processus de calcul ; ou, autre exemple, pour pipeliner deux tâches selon un fonctionnement lecteur-rédacteur.

La seconde source de parallélisme est normalement présente au niveau des opérateurs spécialisés. En effet, leur conception doit exploiter un parallélisme massif dans l'application pour justifier l'usage d'un accélérateur. Mais ce parallélisme, mis en oeuvre par le concepteur de l'opérateur, est totalement caché au programmeur d'application.

Il existe une autre source de parallélisme que nous envisageons d'exploiter dans une ROOM. Les applications que nous visons exhibent un fort potentiel de parallélisme par les données ; par exemple, il est possible de comparer de manière simultanée et indépendante une séquence test avec

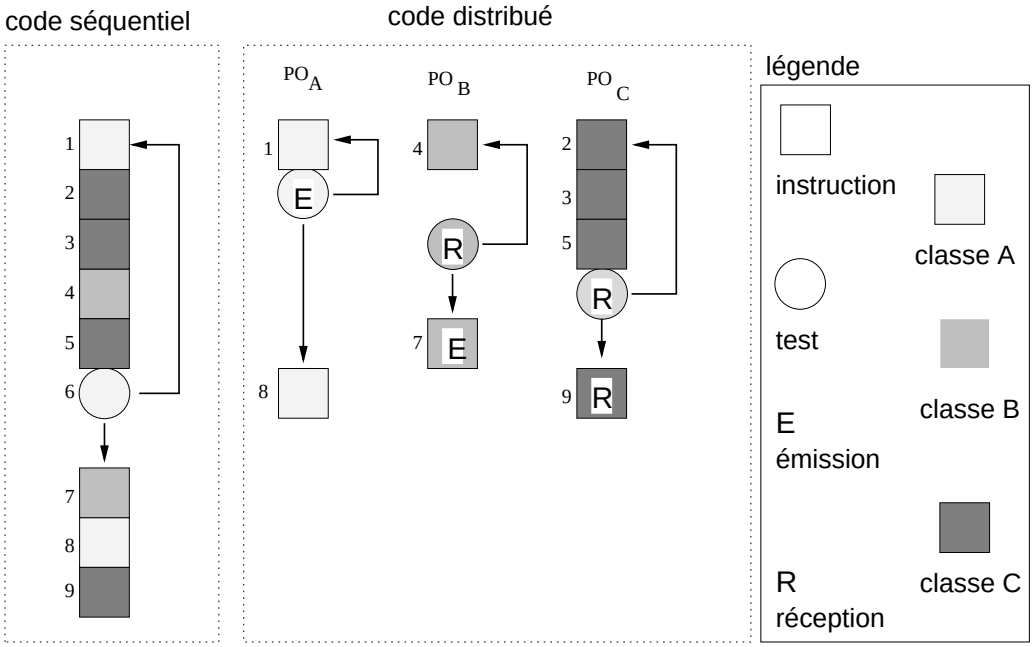

FIG. 7 – Distribution et synchronisation des instructions

plusieurs séquences de référence, puis en fin de traitement, de retenir le meilleur résultat entre les différentes comparaisons. Concrètement celà consiste à implanter dans une ROOM plusieurs processeurs d'objets identiques contenant chacun un opérateur dédié à la comparaison de séquences. Au vue des capacités des circuits reconfigurables actuels, et compte tenu de nos expérimentations déjà réalisés dans ce domaine (voir paragraphe 4), il est envisageable d'implanter dans un seul composant FPGA une dizaine de processeurs spécialisés pour la comparaison de séquence. L'accélération attendue étant proportionnelle au nombre de processeurs, un facteur d'accélération de 10 par rapport à la solution actuelle peut être atteint sans apporter de modification au modèle d'exécution de la ROOM. Par contre, du coté du modèle de programmation il est souhaitable de fournir au programmeur un moyen d'exprimer la duplication de classe. Une solution est de créer une classe générique, par exemple Parallel <Sequence,10>, qui prend en paramètre la classe et le nombre d'occurences à dupliquer. Ce type d'extension des langages à objets a déjà été proposé pour exprimer la distribution de données, par exemple dans [12].

4 Expérimentations

Afin de valider le support d'exécution deux actions ont été menées de front : la validation fonctionnelle, puis la synthèse des processeurs d'objets à partir d'une spécification VHDL.

4.1 Validation fonctionnelle

Un assembleur de code objet parallèle, baptisé OAS a d'abord été conçu. Il prend en entrée un code assembleur tel qu'un compilateur pourrait le produire à partir d'un langage objet de haut niveau. C'est une suite séquentielle de micro-instructions qui ont la particularité d'être préfixées par la classe où elles doivent s'exécuter. Le but d'OAS est de générer un code assembleur pour chaque processeur d'objets en fonction du marquage des micro-instructions. Le code initial est alors réparti entre les

divers processeurs, tout en incluant les mécanismes de synchronisation entre les classes. Le code assembleur pour le routeur – qui n'apparaît pas explicitement dans le code source – est ainsi produit automatiquement. OAS produit également une version binaire du code assembleur requis pour la simulation VHDL des processeurs d'objets.

Un exemple relatif à la comparaison de séquences d'ADN a été écrit en assembleur et simulé. Deux versions ont été considérées :

1. une version basique où toutes les opérations sont programmées à l'aide d'objets prédéfinis standards (cf programme figure 4) ;
2. une version optimisée avec un processeur d'objets spécialisé qui contient un opérateur optimisé de comparaison de séquences (cf programme figure 5).

La simulation fonctionnelle du code produit par OAS permet de mesurer l'apport de la spécialisation en donnant dans chaque cas le nombre de cycles relatif à l'exécution du programme. Par exemple, pour comparer deux séquences de tailles 300, le nombre de cycles est de 5.6×10^6 pour la version séquentielle contre 22000 cycles pour la version avec un opérateur systolique capable de manipuler des séquences de cette taille, ce qui conduit à un gain de l'ordre de 250.

Un autre résultat intéressant est le degré de parallélisme entre les processeurs d'objets. Il est obtenu en faisant le rapport entre le nombre total d'instructions exécutées et le nombre de cycles. Toujours avec le même exemple, le rapport obtenu vaut 1.8 pour la version basique et 2.2 pour la version optimisée. Il signifie donc, qu'en moyenne, plus de 2 processeurs d'objets sur les 5 sont actifs en même temps.

4.2 Synthèse architecturale

Pour valider la faisabilité de l'implantation d'une ROOM sur une structure reconfigurable, et en connaître les performances, les deux types de processeurs d'objets, prédéfinis et utilisateurs, ont été spécifiés en VHDL, validés par simulation puis synthétisés sur un composant FPGA de la famille Virtex de Xilinx.

Un processeur d'objets 32 bits utilisateur a été spécifié. Après synthèse, cet entité représente l'équivalent d'environ $100\,000$ portes logiques (*system gates*). En d'autres termes, un composant FPGA de type Virtex 1000 peut en contenir dix. La fréquence estimée par les outils de CAO est autour de 40 MHz. Ces chiffres montrent d'une part la faisabilité d'une ROOM sur un composant FPGA et, d'autre part, donnent un ordre de grandeur sur la complexité (en termes de nombre de portes) des processeurs d'objets.

Si on considère maintenant les processeurs d'objets prédéfinis, leur taille est plus importante car ils intègrent, en plus, des opérateurs matériels qui réalisent les fonctions de base des classes primitives. Par exemple, le processeur d'objets associé à la classe primitive des entiers doit au moins inclure une unité arithmétique et logique (UAL) et un multiplieur. En fait, le surcoût est faible, surtout dans la perspective des circuits FPGA, comme le VirtexII par exemple, qui intègrent directement de tels opérateurs. Par contre, pour des processeurs d'objets plus spécialisés, comme celui associé à la classe `Sequence` par exemple, l'opérateur de comparaison qui porte sur une chaîne de caractères complète est nettement plus conséquent, et peut consommer à lui seul une bonne partie des ressources reconfigurables.

Ce qu'il faut retenir de ces premières évaluations c'est que la complexité d'un processeur d'objets est finalement faible au regard de l'évolution annoncée des densités des futurs composants FPGA. Si un processeur d'objets représente aujourd'hui $10\,\%$ des ressources d'un composant standard d'un million de portes (Virtex-1000), ce pourcentage tombe à 1 ou $2\,\%$ avec les nouvelles familles de

composants qui annoncent d'ores et déjà des complexités de 10 millions de portes. La majorité des ressources pourra alors être consacrée à des opérateurs complexes encapsulés dans des processeurs d'objets prédéfinis, et c'est l'usage de ces opérateurs qui apportera la puissance de calcul. L'encapsulation dans un environnement de programmation objet apporte, quant à lui, toute l'infrastructure de synchronisation, la facilité et la rapidité de mise en oeuvre.

5 Conclusion

Dans cet article, nous avons présenté la ROOM, un support d'exécution pour programmer un composant FPGA à partir d'un langage à objet. La ROOM a été conçue en appliquant une décomposition hiérarchique de l'application jusqu'au niveau du matériel. Le modèle d'exécution est une machine parallèle dans laquelle deux types de processeurs cohabitent : des processeurs d'objets prédéfinis et des processeurs d'objets utilisateurs.

Rappellons que notre but est de faciliter la conception d'une gamme d'accélérateurs matériel pour extraire de l'information sur la base d'un flux de données. Dans ce type de système, la difficulté n'est pas de concevoir l'architecture originale qui génère la puissance de calcul, mais son intégration dans un environnement informatique. Aussi, notre optique est de considérer l'intégration d'une unité de calcul – aussi complexe soit-elle – dans un processeur d'objet prédéfini et de la manipuler à travers un jeu de méthodes adéquates. Nous avons illustré cette démarche par un exemple classique de recherche de similarité entre séquences d'ADN. Deux implémentations ont été proposées : la première fait appel à des opérateurs arithmétiques standards pour calculer une ressemblance, alors que la seconde met en œuvre une unité de calcul spécialisée (un réseau systolique) pour réaliser le même calcul. A l'aide d'un simulateur (OAS) nous avons estimé un gain de l'odre de 250 (pour des séquences de taille 300) entre la première et la deuxième solution.

Ces performances doivent cependant être nuancées par la *perte* relative dû au contrôle microprogrammé. En effet, une unité spécialisée supporte aisément le traitement d'une nouvelle opérande à chaque cycle d'horloge alors que l'environnement proposé n'est pas calé sur ce rythme optimal : dans l'exemple, plusieurs cycles machines sont nécessaires pour l'alimenter (acquisition des données, gestion du contrôle, etc.).

La conception d'une unité de calcul spécialisée, en vue de son intégration dans une ROOM, doit donc tenir compte de ce contexte, voire même en tirer avantage. Par exemple, une architecture à base d'opérateurs multiplexés dans le temps serait probablement mieux adaptée par rapport à une architecture spatialement optimisée. Elle offrirait, en outre, l'avantage de réduire le volume de ressources matérielles nécessaires (le nombre de portes logiques programmables).

Nous avons également *quantifié* le coût matériel d'un processeur d'objet en terme de ressources reconfigurables. Un tel élément spécialisé pour manipuler des objets représente aujourd'hui 10% des ressources d'un circuit FPGA de milieu de gamme (Virtex-1000 de Xilinx), soit la possibilité d'en implanter un nombre suffisant pour la gamme d'applications visée. Mais en dehors de cet aspect de faisabilité purement technique, la disponibilité de cœurs de processeur standard (IP) et de leurs outils de compilation associés doit également être pris en compte. Si ces derniers sont moins optimals en terme de gestion des objets, ils présentent cependant l'avantage d'être plus compactes, plus rapides et directement exploitables. On peut également viser des composants FPGA de dernière génération (par exemple le Virtex-II pro de Xilinx[1]) sur lesquels plusieurs microprocesseurs sont intégrés. Les processeurs d'objet ont alors un support matériel direct.

Enfin, l'environnement de programmation de la ROOM doit maintenant être complété par un dispositif permettant de spécifier matériellement les unités de calcul des processeurs d'objets prédéfinis.

Cette spécification doit servir à la fois pour la simulation fonctionnelle et pour la synthèse. Puisque les outils en cours de développement (compilateur) ou déjà développés (simulateur) utilisent Java comme langage pivot, nous étudions la possibilité d'établir une connexion avec JHDL[6], un ensemble d'outils CAO – dans un environnement Java – spécialement conçu pour la simulation et la synthèse d'architecture de composants FPGA [4].

Références

[1] *Virtex-II Pro (TM) Platform FPGA Data Sheet (DS083) (09/27/02)*.

[2] European Space Agency. Leon-1 vhdl model. http://www.estec.esa.nl/wsmwww/leon/.

[3] Rajeev Barua, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Maps: A compiler-managed memory system for raw machines. In *Proceedings of the Twenty-Sixth International Symposium on Computer Architecture (ISCA-26)*. Atlanta, GA, 1999.

[4] P. Bellows and B. Hutchings. JHDL-an HDL for reconfigurable systems. In *FCCM 98*. Napa (CA, USA), 1998.

[5] Mikael Collin, Raimo Haukilahti, Mladen Nikitovic, and Joakim Adomat. Socrates - a multi-processor soc in 40 days. In *In Conference on Design, Automation and Test in Europe 2001*. Designer's Forum Munich, Germany, 2001.

[6] G. Fabregat, G. Leòn, B. Pottier, P. Le Berre, and L. Lagadec. Embedded system modeling and synthesis in oo environments. a smart-sensor case study. In *Cases'99*, 1999.

[7] Jan Frigo, Maya Gokhale, and Dominique Lavenier. Evaluation of the streams-c c-to-fpga compiler: an applications perspective. In *International Symposium on Field Programmable Gate Arrays*, pages 134–140, 2001.

[8] P. Guerdoux-Jamet and D. Lavenier. Systolic filter for fast dna similarity search. In *ASAP'95, International Conference on Application SpecificArray Processors*, Strasbourg, France, 1995.

[9] P. Guerdoux-Jamet, D. Lavenier, C. Wagner, and P.Quinton. Design and implementation of a parallel architecture for biological sequence comparison. In *EURO-PAR'96: Parallel Processing*, 1996.

[10] John R. Hauser and John Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 12–21, Napa, CA, 1997.

[11] A. Kim and J.M. Chang. Designing a java microprocessor core using fpga technology. In *Proceedings of 1998 IEEE International ASIC Conference*, pages 13–16. Rochester, NY, 1998.

[12] Pacale Launay. *Génération de programmes parallèles distribués dans un environnement à objet*. PhD thesis, Université de Rennes I, 1999.

[13] Dominique Lavenier. Samba: Systolic accelerator for molecular biological applications. Technical Report 2845, INRIA, 1996.

[14] Dominique Lavenier, Patrice Quinton, and Frédéric Raimbault. Architectures systoliques et parallélisme de données. *Techniques et Sciences Informatique*, 12(5):597–620, 1993.

[15] Dominique Lavenier and Frédéric Raimbault. Relacs for systolic programming. In *Applications-Specific Array Processors (ASAP)*, pages 132–135, 1993.

[6] JHDL : http://www.jhdl.org

[16] A. Mozipo, D. Massicote, P. Quinton, and T. Risset. Automatic synthesis of a parallel architecture for kalman filtering using mmalpha. In *IEEE Canadian Conference on Electrical and Computer Engineering*, Edmonton, Canada, 1999.

[17] Frédéric Raimbault. *Etude et réalisation d'un environnement de simulation parallèle pour les algorithmes systoliques*. PhD thesis, Université de Rennes I, 1994.

[18] Stéphane Rubini and Dominique Lavenier. Les architectures reconfigurables. *Calculateurs Parallèles*, 9(1), 1997.

[19] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[20] D. Ungar, R. Blau, P. Foley, D. Samples, and D. Patterson. Architecture of soar: Smalltalk on a risc. In *11th Annual Symposium on Computer Architecture*, 1984.

[21] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable active memories: the coming of age. *IEEE Trans. on VLSI*, 4(1):56–69, 1996.

[22] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, pages 86–93, 1997.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399