RENPAR'14 / ASF / SYMPA
Hamamet, Tunisie, 10 – 13 avril 2002

ROOM : des machines reconfigurables orientées objet

Frédéric Raimbault[1] – Dominique Lavenier[2]
[1] *VALORIA*, Université de Bretagne Sud, Vannes
[2] *IRISA/CNRS*, Rennes

Résumé
Les travaux présentés dans cet article ont pour objectif de *programmer* une application sur un composant FPGA à partir d'un langage orienté objet. L'architecture dérivée de la spécification est une machine parallèle, baptisée ROOM : *Reconfigurable Object Oriented Machine*. Elle est composée d'autant de *processeurs d'objets* qu'il y a de classes présentes dans l'application. L'article montre l'intérêt de cette architecture et présente les premières expérimentations.

Mots-clés : architecture, reconfigurable, objet, FPGA, parallélisme.

1. Introduction

Depuis quelques années, la technologie reconfigurable prétend combler le fossé entre les microprocesseurs et les ASICs (*Application Specific Integrated Circuit*). On met à la fois en avant sa flexibilité de programmation et une puissance de calcul potentielle très importante, notamment pour toute une gamme d'applications spécifiques pour lesquelles les microprocesseurs sont trop lents et les ASICs trop rigides. La téléphonie embarquée, avec les mobiles de troisième génération, par exemple, est une bonne illustration : le téléphone doit s'adapter instantanément aux différents standards, supporter un éventail de services, dont certains sont encore à imaginer, et développer des capacités de calcul bien supérieures à celles des microprocesseurs ou DSP usuels. Une alternative possible est l'intégration d'un système complet (ou SoC pour *System on Chip*) sur une structure reconfigurable, par un exemple sur un composant FPGA (*Field Programmable Gate Array*). Ce système s'adapte à l'application car sa structure matérielle se *moule* aux exigences des calculs. Cette solution est très sérieusement envisagée du fait de l'évolution des technologies CMOS sub-microniques qui conduisent à des structures reconfigurables extrêmement denses, capables de supporter des systèmes numériques très complexes.
Cependant, si l'accroissement des ressources reconfigurables satisfait de mieux en mieux les attentes des concepteurs, il n'en est pas de même de la convivialité des environnements logiciels. Les outils de programmation sont intégrés dans des plateformes traditionnelles de CAO de circuits intégrés : la mise en oeuvre d'une application ne peut se concevoir sans une réflexion préalable du support d'exécution. En d'autres termes, il faut d'abord concevoir l'architecture, puis porter l'application sur cette architecture. Ce n'est donc pas le schéma de compilation traditionnel auquel est confronté un programmeur habitué aux environnements de compilation et de génie logiciel. Du coup, cette technologie, séduisante par ses capacités, est très largement sous-exploitée.
Idéalement, pour programmer une application sur un composant FPGA, on aimerait procéder comme on le fait pour un microprocesseur : spécifier son application dans un langage de haut niveau, inclure éventuellement des bibliothèques de composants optimisés, compiler le tout, configurer le système et lancer l'exécution. L'ambition du projet ROOM (*Reconfigurable Object Oriented Machine*) est de tendre vers ce scénario.
La difficulté est de trouver un bon compromis entre la généralité du support d'exécution et sa capacité à se spécialiser. En effet, le support doit être suffisamment général pour programmer différents algorithmes dans un langage de haut niveau ; mais il est également indispensable de le spécialiser pour accélérer les parties de l'application coûteuses en calcul. Enfin, il convient de s'assurer qu'un compilateur est à même d'exploiter les parties matérielles spécifiques.

Nous proposons de concilier cette double exigence, programmabilité et spécificité, au sein d'un cadre unique à l'aide d'une méthodologie de développement orientée objet. L'idée développée dans cet article est que l'adoption d'une approche hiérarchique, de l'application jusqu'au matériel, est non seulement profitable au développeur de logiciel, mais aussi réaliste et efficace avec les technologies reconfigurables actuelles et à venir.

Le reste de cet article est organisé de la manière suivante. Dans le paragraphe 2 nous présentons d'abord le support d'exécution de la ROOM. Puis, dans le paragraphe 3, nous décrivons plus précisément son architecture. Le paragraphe 4 est consacré à une première mise en oeuvre pour valider le concept. Le paragraphe 5 situe nos travaux, tandis que le paragraphe 6 conclut et propose quelques perspectives.

2. Support d'exécution

Notre support d'exécution se présente sous la forme d'une machine distribuée, dont les composants principaux sont appelés *processeurs d'objets*. Dans cette partie, nous exposons les principes de fonctionnement de cette architecture, ainsi que son intérêt. Nous terminons par 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 distribué 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. Les processeurs d'objets n'exécutent donc pas la totalité du programme, mais uniquement les parties de l'application qui les concernent.

Le second principe qui a guidé la définition du support d'exécution est que le code de chaque processeur d'objets ainsi que leur synchronisation doivent pouvoir être générés automatiquement à partir d'un programme source séquentiel. Or, en vertu du principe précédent, la distribution du code ne repose que sur l'identification de la classe de chaque objet impliqué dans une instruction. 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 ;
- 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 ;
- 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.

Les deux derniers cas supposent l'existence d'un mécanisme de communication et de synchronisation global. Le choix du protocole de communication/synchronisation a été fait en privilégiant la simplicité et l'extensibilité de l'architecture. Le routage des données et des conditions s'effectue dans une unité spécialisée : le *routeur*. Les processeurs d'objets se synchronisent sur des files d'attentes internes (une file en entrée et une en sortie pour les communications et deux autres pour les synchronisations). Le

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

code du routeur est généré automatiquement à partir du programme source comme pour les autres processeurs d'objets.
Le compilateur effectue la distribution des instructions dans les processeurs d'objets en synthétisant un attribut de classe dans l'arbre syntaxique. Cette technique a été appliquée avec succès dans un autre contexte [14]. 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 flots est effectuée au moment des évaluations de conditions dans les structures de contrôle du programme original.

2.2. Intérêt 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 interprocesseurs mais n'exclut pas l'exploitation d'un parallélisme de données, par exemple sous la forme d'un opérateur 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 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 où 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 [11, 12, 14], et pour une mise en oeuvre dans un composant spécialisé, à [7, 10].
Développée dans un langage à objet, les classes d'objets nécessaires à notre application se limitent à cinq : une classe `Sequence` pour toutes les opérations relatives aux séquences, une classe `Main` qui

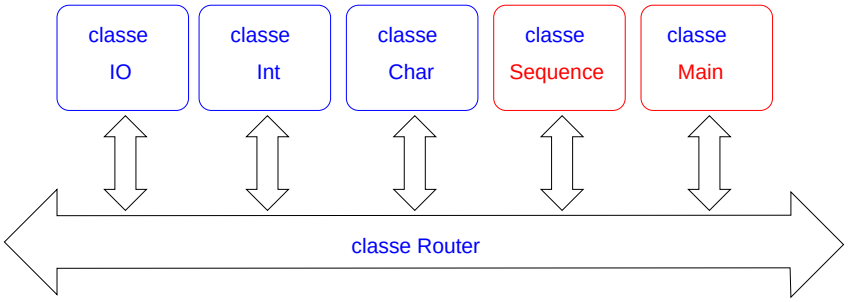

FIG. 1 – Architecture d'une ROOM pour la comparaison de séquences d'ADN

contient la partie initialisation 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. La figure 1 reprend ces éléments sous forme de processeurs d'objets et contient une représentation schématique de ROOM. Le seul ajout (systématique) est le composant (classe `Router`) qui gère le routage 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. Architecture d'une ROOM

La mise en oeuvre de machines selon le modèle présenté dans le paragraphe précédent est contraint par la technologie des composants reconfigurables. Nous présentons à la suite l'architecture globale d'une ROOM, puis la micro-architecture d'un processeur d'objets.

3.1. Architecture générale

La figure 2 contient les différents composants de l'architecture d'une ROOM. Seuls deux processeurs d'objets y sont représentés :

1. un processeur d'objets primitifs : ce type de processeur d'objets contient toutes les unités nécessaires à la mémorisation et au calcul de valeurs élémentaires. On trouve autant de processeurs d'objets primitifs dans la machine que l'application nécessite de classes d'objets prédéfinies.
2. un processeur d'objets non primitifs : par rapport aux processeurs d'objets primitifs, ce second type de processeur d'objets ne contient pas d'unités de calcul. Les objets manipulés sont composés de références (d'adresses) vers d'autres objets non primitifs ou vers des objets primitifs. On trouve autant de processeurs d'objets dans la machine que l'application comprend de classes d'objets définies par l'utilisateur.

Le troisième élément de l'architecture est le routeur : c'est l'unité de communication et de synchronisation. Il n'existe qu'une seule unité de ce type pour gérer toutes les communications internes à la machine (les communications avec l'extérieur de la machine sont déléguées à un ou plusieurs autres processeurs d'objets spécialisés dans cette tâche). Synchronisation et communication utilisent deux bus distincts contrôlés par le routeur.

3.2. Architecture d'un processeur d'objets

L'architecture interne d'un processeur d'objets est assez semblable à celle d'un microprocesseur RISC. Elle possède une mémoire de code, un séquenceur, des registres, une mémoire de données et des files d'entrées-sorties. Les processeurs d'objets primitifs possèdent en plus une unité de calcul (une UAL dans le cas du processeur d'objets des entiers) adaptée aux types des données de la classe correspondante.

Par ailleurs, la gestion dynamique de la mémoire, caractéristique des langages à objets, implique un surcoût important à l'exécution quand elle est réalisée de manière logicielle. C'est pourquoi, un processeur d'objets intègre une unité particulière : l'unité de gestion des objets (UGO) qui possède les fonctionnalités suivantes :

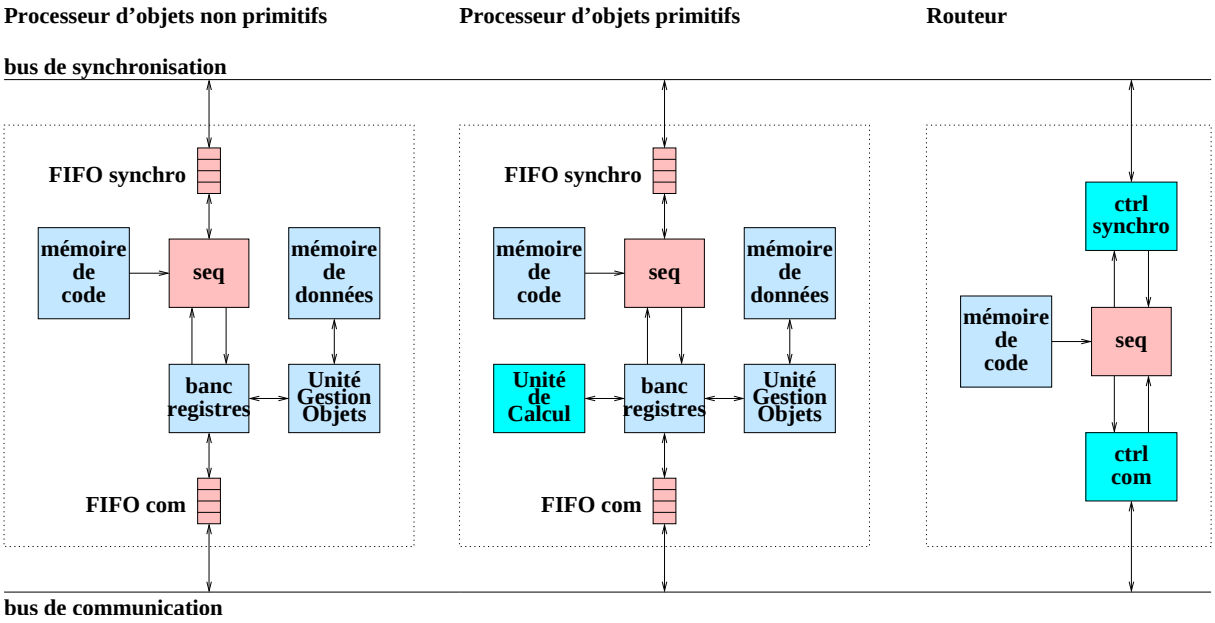

FIG. 2 – Architecture d'une ROOM

– l'allocation mémoire, à la demande, d'un objet ; l'UGO rend l'adresse de base de l'objet alloué. Un point intéressant est que la taille des objets gérés par chaque processeur d'objets est fixe, ce qui simplifie considérablement la gestion mémoire ;
– le calcul de l'adresse d'un attribut d'un objet à partir de l'adresse d'un objet et du numéro de l'attribut ;
– l'accès à un attribut d'un objet ;
– la libération mémoire d'un objet.

Telle qu'elle est définie, cette unité trouve aussi bien sa place dans un processeur d'objets primitifs comme dans un processeur d'objets non primitifs. Seule change la taille des objets considérés : atomique ou à plusieurs champs.

En fonction de l'application, les processeurs d'objets sont spécialisés par rapport aux objets qu'ils manipulent. En dehors du séquenceur et des FIFOs, toutes les unités peuvent être optimisées (adaptées à l'application), que se soit sur la taille des chemins de données, la taille des mémoires, la gestion des objets ou la fonctionnalité de l'unité de calcul.

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 primitifs standards ;
2. une version optimisée avec un processeur d'objets spécialisé qui contient un opérateur optimisé de comparaison de séquences (un mini réseau systolique).

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 basique 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 probant 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, primitifs et non primitifs, ont été spécifiés en VHDL, validés par simulation puis synthétisés sur un composant FPGA de la famille Virtex de Xilinx.
Un processeur d'objets non primitifs (sur 32 bits) a été spécifié avec les caractéristiques suivantes : une pile de 32 registres, une mémoire d'objets de 64 mots, une mémoire de code de 256 mots, et 2 FIFOs de communication de 16 mots. 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 primitifs, leur taille est plus importante car elles intègrent, en plus, des opérateurs matériels qui implémentent 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 Virtex 2 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.
D'un point de vue performance, la version optimisée sur l'exemple des séquences d'ADN reste extrêmement compétitive par rapport à une exécution sur un microprocesseur : le gain varie bien sûr suivant la complexité de l'opérateur de comparaison de séquences, mais il se situe dans une fourchette de 10 à 20 (comparaison par rapport aux micro-processeurs de dernière génération).
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 primitifs, 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. Situation des travaux

L'approche objet a déjà été étudiée pour implanter des applications spécifiques sur FPGA. Mais le plus souvent – par exemple dans [5]– 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 seule seule autre tentative de réalisation d'une machine parallèle

programmable dans un seul composant FPGA. Il s'agit de SoCrates [4], 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 [17]. 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 [2]. 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 [1] – soit l'association avec un microprocesseur hôte comme accélérateur de calcul – par exemple [8] – soit l'interconnexion de plusieurs FPGA pour former une machine spécialisée [16].
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 Smalltalk [15]. 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 [9] 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 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 [6]. 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 [13]. 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.

6. Conclusion

Dans cet article, nous avons montré comment la ROOM, un support d'exécution pour programmer un composant FPGA à partir d'un langage à objet, a été conçue en appliquant une décomposition hiérarchique de l'application jusqu'au niveau du matériel.
Si une validation préliminaire a été faite, il reste encore plusieurs aspects non résolus. Le premier concerne la gestion dynamique de la mémoire d'objets, et plus particulièrement la spécification d'un système matériel distribué de *ramasse-miettes* entre les différentes mémoires des processeurs d'objets.
En second lieu, il convient de mettre en place un environnement de programmation objet pour la ROOM. C'est un travail en cours qui s'appuie à la fois sur la définition d'un langage objet simple et sur la mise en oeuvre de son compilateur. Connecté au simulateur OAS déjà réalisé, il devrait fournir un outil puissant pour évaluer rapidement les performances d'une application que l'on souhaite synthétiser sur une ROOM. Ce compilateur doit également donner une description de l'architecture de la ROOM.
Enfin, cet environnement doit être complété par un dispositif permettant de spécifier matériellement les unités de calcul des processeurs d'objets primitifs. 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[2], un ensemble d'outils CAO – dans un environnement Java – spécialement conçu pour la simulation et la synthèse d'architecture de composants FPGA [3].

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

Remerciements
Nous remercions Pierre Le Mignant, étudiant en stage de Maîtrise, pour l'étude VHDL d'un prototype d'architecture de ROOM et sa synthèse sur Xilinx (Virtex).

Bibliographie

1. Agency European Space, Leon-1 vhdl model, http ://www.estec.esa.nl/wsmwww/leon/, 2001.
2. R. Barua, W. Lee, S. Amarasinghe, A. Agarwal, Maps : A compiler-managed memory system for raw machines, *Proceedings of the Twenty-Sixth International Symposium on Computer Architecture (ISCA-26)*. Atlanta, GA, 1999.
3. P. Bellows, B. Hutchings, Jhdl-an hdl for reconfigurable systems, *FCCM 98*, Napa, CA, USA, 1998.
4. M. Collin, R. Haukilahti, M. Nikitovic, J. Adomat, Socrates - a multiprocessor soc in 40 days, *Conference on Design, Automation and Test in Europe 2001*, Designer's Forum, Munich, Germany, 2001.
5. G. Fabregat, G. Leòn, B. Pottier, P. Le Berre, L. Lagadec, Embedded system modeling and synthesis in oo environments. A smart-sensor case study, *Cases'99*, 1999.
6. J. Frigo, M. Gokhale, D. Lavenier, Evaluation of the streams-c c-to-fpga compiler : an applications perspective, *International Symposium on Field Programmable Gate Arrays*, Monterey, CA, USA, 2001.
7. P. Guerdoux-Jamet, D. Lavenier, C. Wagner, P. Quinton, Design and implementation of a parallel architecture for biological sequence comparison, *EURO-PAR'96 : European Conference on Parallel Processing*, Lyon, France, 1996.
8. J.R. Hauser, J. Wawrzynek, GARP : A MIPS processor with a reconfigurable coprocessor, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, USA, 1997.
9. A. Kim, J.M. Chang, Designing a java microprocessor core using fpga technology, *Proceedings of 1998 IEEE International ASIC Conference*. Rochester, NY, USA, 1998.
10. D. Lavenier, SAMBA : Systolic Accelerator for Molecular Biological Applications, Rapport technique INRIA No. 2845, 1996.
11. D. Lavenier, P. Quinton, F. Raimbault, Architectures systoliques et parallélisme de données. *Technique et Science Informatiques*, vol. 12, No. 5, pp. 597–620, 1993.
12. D. Lavenier, F. Raimbault, Relacs for systolic programming, *Applications-Specific Array Processors (ASAP)*, 1993.
13. A. Mozipo, D. Massicote, P. Quinton, T. Risset, Automatic synthesis of a parallel architecture for kalman filtering using mmalpha, *IEEE Canadian Conference on Electrical and Computer Engineering*, Edmonton, Canada, 1999.
14. F. Raimbault, *Etude et réalisation d'un environnement de simulation parallèle pour les algorithmes systoliques*, Thèse de PhD, Université de Rennes I, 1994.
15. D. Ungar, R. Blau, P. Foley, D. Samples, D. Patterson, Architecture of soar : Smalltalk on a risc, *11th Annual Symposium on Computer Architecture*, 1984.
16. J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, P. Boucard, Programmable active memories : the coming of age, *IEEE Trans. on VLSI*, vol. 4, No. 1, pp. 56–69, 1996.
17. E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, A. Agarwal, Baring it all to software : Raw machines. *IEEE Computer*, pp. 86–93, September 1997.