Génération de micro-code parallèle pour la carte coprocesseur Rapid-2

Laurent Winckel

Laboratoire MASI / Université Pierre et Marie Curie,
4 Place Jussieu, 75252 Paris cedex 05, France

Résumé. La carte coprocesseur Rapid-2 est une carte accélératrice massivement parallèle SIMD pour PC. Un microprogramme, résident dans une RAM de la carte, contrôle son comportement. Elle permet de réduire le temps d'exécution de certaines applications très coûteuses en temps de calcul. Le langage L1, qui est une extension du langage C ANSI permettant d'exploiter le parallélisme de données, permet d'écrire des applications utilisant la carte. Le compilateur RAC réalise la compilation des programmes en L1, il exploite le parallélisme de traitement des différentes unités fonctionnelles de la carte pour fournir un micro-code efficace.

1 Introduction

Dans le cadre du projet Rapid-2, nous avons développé une architecture SIMD microprogrammable complète [1][2] ainsi que les différents outils logiciels permettant de l'utiliser. Nous avons défini un langage de microprogrammation et le micro-assembleur MicA. Ce langage permet de programmer la carte en spécifiant à chaque cycle les opérations à réaliser par les différentes unités de traitement de la carte. Pour chaque application, il faut également écrire un programme de contrôle de l'ordinateur hôte.

Cet environnement de bas niveau ne permet pas de programmer facilement et rapidement une application. Parmi les différents langages de programmation pour machine parallèle, tel que le langage C* [3] de la Connection Machine ou le langage Adapt [4] pour le traitement d'image, aucun ne permettait de tirer pleinement profit de notre architecture. En effet, C* est un langage permettant de manipuler des types de données non appropriées à la carte Rapid-2, tel que les nombres flottants. D'autre part, des mécanismes originaux de la carte (éléments de mémoire étiquetés, opérations 32 bits ou quatre fois 8 bits, arithmétique multi-précision, etc...) n'auraient pas été exploités par ce langage. Les langages spécialisés comme Adapt ne convenaient pas non plus car nous souhaitions disposer d'un langage plus général.

Nous avons donc spécifié le langage de programmation L1 adapté à l'architecture Rapid-2 [5]. Il permet de programmer une application à l'aide d'un seul programme source. Le compilateur RAC convertit le fichier source du programme L1 en deux fichiers résultats, un premier en langage C pour l'ordinateur hôte et un second en micro-assembleur MicA pour la carte coprocesseur Rapid-2.

Les performances de l'application compilée dépendent fortement de la qualité du micro-code généré. L'architecture de la carte Rapid-2 permet peu de contrôle

dans le séquencement des microprogrammes, l'ordinateur hôte effectue la majorité des opérations de test ainsi que le contrôle des boucles. De plus, les applications envisagées [6] pour la carte contiennent de longue séquence de calculs séquentiels. De ce fait, l'optimisation du micro-code porte essentiellement sur de longs blocs de micro-code dépourvus d'instruction de branchement. Les techniques d'optimisation de programmes parallèles tel que trace scheduling [7], percolation Scheduling [8], etc... [9][10], cherchant à exploiter le parallélisme au-delà des blocs, n'ont que peu d'intérêt dans ce contexte. RAC utilise une méthode d'optimisation simplifiée des blocs de micro-code.

2 Le compilateur RAC

Le compilateur est un outil essentiel dans l'environnement de développement du projet Rapid-2. Il permet la programmation rapide et efficace d'applications utilisant la carte Rapid-2. Son rôle est de convertir le fichier source en L1 de l'application en un programme pour l'ordinateur hôte et un microprogramme pour la carte. Le processus de conversion comporte trois phases :

Le processus de conversion commence par une analyse lexicale et syntaxique classique. Elle est réalisée à l'aide des outils " yacc " et " lex ". Le résultat de cette première phase est une structure en mémoire représentant l'arbre syntaxique du fichier analysé.

La deuxième opération consiste à séparer les parties du programme source destinées à l'ordinateur hôte des parties destinées à la carte Rapid-2. L'appartenance des parties du programme est déduite du type des variables utilisées dans chaque partie.

Les parties du programme concernant la carte Rapid-2 sont alors converti en micro-code, les autres partie du programme permettent de générer un fichier source en langage C pour l'ordinateur hôte.

2.1 Génération du micro-code parallèle

La parallélisation du micro-code est réalisée en deux phases. La première phase est statique. Pour chaque opération élémentaire d'une expression de L1, une courte séquence de micro-code générique est décrite dans le compilateur. L'enchaînement des micro-instructions dans ces séquences reflète les dépendances de données et de ressources. Ces séquences ne peuvent pas être restructurées, elles forment les briques de base manipulées lors de la génération du microprogramme. La taille de ces séquences varie d'une à une dizaine de micro-instructions. Toutefois, la taille moyenne est de quelques micro-instructions.

La deuxième phase de la génération concerne l'assemblage des séquences de base d'écrites dans le paragraphe précédant. Si on se contente de placer bout à bout les séquences de base, le résultat obtenu offre un faible taux d'exploitation du parallélisme. En effet, seul le parallélisme contenu dans les séquences de base intervient. Pour augmenter celui-ci l'assemblage sera réalisé en exploitant le pipeline logiciel lorsque les contraintes de données et de ressources le permettent.

Pour pouvoir exploiter cette technique de manière automatique il est nécessaire de contrôler si les séquences de micro-code, générées successivement par le compilateur, contiennent des dépendances de ressources ou de données. Cela est réalisé dans RAC grâce à des masques associés à chaque micro-instruction. La structure d'une micro-instruction manipulée par le compilateur contient un tableau de pointeurs vers les mnémoniques, un masque de ressources utilisées, un masque de données consommées et un masque de données produites.

En 1966, Berstein a énoncé les conditions suffisantes d'indépendance de deux opérations. Si on note respectivement $M(\sigma)$ et $L(\sigma)$ les ensembles d'éléments mémorisants modifiés et lus par une opération σ, les conditions d'indépendance de deux opérations s et t prennent la forme suivante : $M(s) \cap L(t) = \emptyset$, $L(s) \cap M(t) = \emptyset$ et $M(s) \cap M(t) = \emptyset$. Une contrainte supplémentaire concerne les ressources utilisées. Si on note $U(\sigma)$ les ressources utilisées par une micro-instruction σ, les micro-instructions s et t sont fusionnables si la condition suivante est respectée : $U(s) \cap U(t) = \emptyset$.

En utilisant ces contraintes, on peut déterminer la profondeur d'imbrication possible d'une séquence de micro-instruction s2 dans la séquence précédente s1, comme le montre l'algorithme 1. Dans cette algorithme la fonction longueur(σ) renvoie le nombre de micro-instructions contenues dans la séquence de micro-instruction σ. Les fonctions $U(\sigma)$, $L(\sigma)$ et $M(\sigma)$ renvoient respectivement les masques des ressources utilisées, des données consommées et des données produites par la séquence σ. Les fonctions premieres(σ, δ) et dernieres(σ, δ) renvoient respectivement les δ premières ou dernières micro-instructions de la séquence σ.

```
profondeur=0; i=0;
tant que i ≤ longeur(s1) et i ≤ longeur(s2)
   si U(dernieres(s1, i))∩U(premieres(s2, i))=∅
      alors si M(dernieres(s1, i))∩L(premieres(s2, i))=∅
         alors profondeur=i;
         sinon sortie;
      fin si
   fin si
   si L(dernieres(s1, i))∩M(premieres(s2, i))≠ ∅ ou
     M(dernieres(s1, i))∩M(premieres(s2, i))≠ ∅
      alors sortie;
   fin si
   i=i+1;
fin tant que
sortie
```

Algorithme 1. Calcul de la profondeur d'imbrication possible de séquences.

Une fois connue la profondeur π d'imbrication possible des séquences, il est possible de fusionner deux à deux les π dernières micro-instructions de la première séquence et les π premières micro-instructions de la seconde séquence.

2.2 Résultats

Une version systolique de l'algorithme de recherche d'alignement entre des séquences génétique décrit par Smith et Waterman [6] a servi de base pour l'analyse des performances du code généré par RAC. Nous avons réalisé une version microprogrammée manuellement (MicA) de cet algorithme. Cette version a été écrite avant l'apparition du langage L1 et des outils permettants de l'exploiter. Environ six mois ont été nécessaire pour cette première version de l'application. La programmation de la version L1 a pris seulement deux semaines. Le tableau 1 présente la taille des micro-codes, pour la boucle principale et l'application entière, de la version MicA et des versions compilées normalement (RAC) et avec la compression de micro-code (RAC -O). On peut constater une importante réduction dans la taille des micro-codes entre la version normale et la version optimisée. D'autre part, le rapport entre la taille de la version optimisée et celle de la version MicA est faible (124 pour la boucle principale et 199 pour l'application entière). La compaction est très bonne dans la boucle principale qui est constituée d'un long bloc de calculs séquentiels, par contre sur le reste du programme elle est un peu moins bonne car les blocs sont de taille moindre.

	MicA	RAC	RAC -O	RAC/MicA	RAC -O/MicA
Boucle principale	171	371	212	217 %	124 %
Application entière	298	899	593	302 %	199 %

Tableau 1. Taille des microprogramme et rapport entre les tailles.

Le tableau 2 présente les temps d'exécution des différents versions de l'application. Ces temps ont été obtenus par simulation de l'exécution des applications à l'aide de ROLLS, la carte Rapid-2 n'étant pas encore prête. Les rapports entre les temps d'exécution des versions RAC par rapport à la version MicA sont faibles, surtout avec la version optimisée 141 Ils confirment les bonnes performances fournis par la technique de compaction de micro-code.

Taille des séquences	MicA	RAC	RAC -O	RAC/MicA	RAC -O/MicA
400	211756	435520	297999	206 %	140 %
800	365549	785568	516947	215 %	141 %

Tableau 2. Temps d'exécution des applications et rapport entre les temps.

3 Conclusion

Cet article présente le langage de programmation L1 et son compilateur RAC pour la carte Rapid-2. Le langage L1 permet une programmation efficace et simple de la carte. La programmation en L1 d'une application est beaucoup plus

rapide que la micro programmation manuelle en MicA. De plus les performances obtenues par la version compilée de l'application sont très peu dégradées. Les bonnes performances obtenues sur l'application d'alignements montre que le langage est bien adapté à l'architecture de la carte. Le compilateur RAC utilise une technique d'optimisation de micro-code simple mais efficace dans le contexte de la carte Rapid-2. En effet, la dissymétrie des unités opératives parallèle VLIW de la carte offre peu de liberté sur le réordonnancement des opérations. De plus la quasi-absence d'instruction de branchement dans les blocs de micro-code rend peu intéressant l'emploi d'algorithmes de réordonnancement plus complexes.

Remerciements

Je tiens à remercier toutes les personnes impliquées dans le projet Rapid-2 et plus particulièrement A. Greiner, P. Faudemay, J. Penné, D. Archambaud et I. S. Silva. Le projet est partiellement financé par le GdR "Architecture Nouvelle de Machine ", le GdR " Informatique et Génome " et le GIP GREG "Groupement de Recherches et d'Etudes sur les Génomes".

Références

1. Archambaud, D., Faudemay, P.: Rapid-2, An Object-Oriented Associative Memory Applicable to Genome Processing. Proc. of 27th Annual Hawaii Inter. Conf. on System Sciences Maui Hawaii (January 1994)
2. Faudemay, P.: A Paginated Set-Associative Architecture for Databases. In: Emerging Trends in Database and Knowledge-Base Machines : the Application of Parallel Architectures to Smart Information Systems. Edited by M. Abdelguerfi et S. Lavington, IEEE (1995)
3. Franckel, J.: C* language reference manual. Thinking Machines Corporation (1991)
4. Webb, J. A.: Steps Toward Architecture-Independent Image Processing. Computer 25 2 (February 1991)
5. Winckel, L., Faudemay, P.: A high level language for the Rapid-2 massively parallel accelerator board. IBP research report MASI (1995)
6. Archambaud, D. et al.: Systolic Implementation of Smith and Waterman Algorithm on a SIMD Coprocessor. In: Algorihms and parallel VLSI Architectures III. Edited by M. Moonen and F. Catthoor (1995)
7. Fisher, J. A.: Trace Scheduling : A technique for global microcode compaction. IEEE Transactions on Computers 7 (1981)
8. Bcioglu, K. E., Nicolau, A.: A global resource-constrained parallelization technique. Proc. of 3rd inter. Conf. on SuperComputing Crete Greece (June 1989)
9. Rau, B. R.: Data Flow and Dependence Analysis for Instruction Level Parallelism. Proc. of 4th inter. Workshop on Language and Compilers for Parallel Computing Santa Clara California USA (August 1991)
10. Fautrier, P.: Fine-grain Scheduling under Ressource contraints. Proc. of 7th inter. Workshop on Language and Compilers for Parallel Computing Ithaca NY USA (August 1994)