Un nouveau modèle générique d'architecture hétérogène pour la méthodologie AAA

Thierry Grandpierre[1], Yves Sorel[2]

[1]Laboratoire A2SI
ESIEE Paris
93162 Noisy le Grand Cedex
t.grandpierre@esiee.fr

[2] Action OSTRE
INRIA Rocquencourt
78153 Le Chesnay Cedex
yves.sorel@inria.fr

Résumé – Dans le cadre de la méthodologie AAA[1], nous présentons un nouveau modèle d'architectures matérielles distribuées et hétérogènes, ainsi qu'un nouveau modèle d'implantation des algorithmes de TSI sur notre modèle d'architecture. Ces modèles, basés sur les graphes, permettent de faire de la prédiction de performances temporelles et spatiales de l'implantation d'un algorithme sur une architecture, d'effectuer des optimisations de façon à respecter des contraintes temps réels et/ou de taille d'architecture, et enfin ils permettent de générer automatiquement l'exécutif distribué nécessaire à l'exécution d'un algorithme sur une architecture. Pour cela ces modèles permettent d'utiliser efficacement les différents moyens de communications (passage de messages, mémoires partagées) des architectures distribuées, tout en offrant la possibilité de prendre en compte l'occupation et la bande passante des mémoires dans le processus d'optimisation de l'implantation d'un algorithme sur une architecture.

Abstract – This paper presents a new model of distributed heterogeneous hardware architecture and a new model of implementation of an algorithm onto an architecture for the AAA methodology[1]. These models are graph based and allow to make spatial and temporal performances prediction of the implementation of an algorithm onto an architecture, but also to make spatial and/or temporal optimisation and automatic dedicated executive generation. For these goals, these models allow to take into account different types of communication media (message passing, shared memory) and memories characteristics (size, bandwith) during the implementation optimisation.

1. Introduction

Afin d'aider les concepteurs à développer rapidement des applications temps réel distribuées et optimisées (i.e. qui respectent les contraintes temps réel et minimisent la taille des architectures) nous développons la méthodologie AAA (Adéquation Algorithme Architecture) [1] qui prend en compte toutes les étapes du développement d'une application, de sa spécification haut niveau jusqu'à l'exécution du code dans les composants. Cette méthodologie est basée sur un formalisme d'hypergraphes pour modéliser l'algorithme, l'architecture et l'implantation de l'algorithme sur l'architecture. Une implantation optimisée est obtenue par transformation de graphes, elle correspond à une distribution et un ordonnancement hors-lignes des éléments de l'algorithme sur l'architecture. Ceci permet, après une caractérisation préalable, de prédire le comportement de l'application et d'en construire son exécutif distribué et optimisé. Nous présentons ici un nouveau modèle d'architecture. Plus fin et plus générique, il prend en compte les spécificités des architectures hétérogènes distribuées, par exemple les hiérarchies de mémoires de caractéristiques différentes, les communications à la fois par mémoires partagées et par passage de messages (bus point à point, multipoints avec ou sans broadcast). Après un bref rappel de notre modèle d'algorithme, nous présentons un nouveau modèle d'implantation qui prend en compte les nouveautés de notre modèle d'architecture et qui permet de prédire plus finement les performances (temporelles, mais aussi spatiales : occupation mémoire) de l'implantation.

2. Modèle d'architecture

Les modèles formels d'architecture existant peuvent être classés en deux grandes parties : les modèles de haut niveau (PRAM, DRAM, BSP, LogP, CGM, etc) et les modèles de bas niveau (basés sur les Hardware Description Langages). Après les avoir soigneusement étudiés [2], nous avons constaté leurs principales limites dans le cadre AAA : soit ils sont trop abstraits dans le sens ou des hypothèses trop simplificatrices sont effectuées sur l'architecture sous-jacente (durée de communication constante...), soit leur granularité est trop fine comparée à celle de la spécification algorithmique souhaitée dans AAA. Ceci a aussi pour conséquence d'augmenter à la fois la complexité de spécification d'une architecture et la complexité du problème d'optimisation de l'implantation. Les outils logiciels tels que Ptolemy, Casch, Trapper semblent reposer sur des modèles d'architecture plus adaptés, mais ils sont implicites et ne se fondent donc pas sur un formalisme réutilisable. De plus, les caractéristiques des architectures de ces modèles sont relativement figées (par exemple réseau de communications de type complètement connecté, communication en temps constant, etc.). Nous présentons donc un modèle d'architecture [2] basé sur les graphes orientés, d'un niveau de granularité intermédiaire à ceux présentés, permettant de spécifier des architectures communiquant par différents moyens de communication (passage de message sur bus point à point ou multipoints, communication par mémoire partagée mono-port, multiports etc.). Ce modèle doit permettre de mettre en évidence le parallélisme qu'offre une architecture ainsi que toutes les ressources fondamentales qu'il est nécessaire d'allouer pour exécuter un algorithme (processeurs, mémoires, canaux DMA). Ce modèle doit à la fois

permettre la prédiction de performance d'une application, l'optimisation de son implantation, et finalement permettre de générer automatiquement son exécutif distribué. L'architecture distribuée hétérogène est modélisée par un graphe orienté G_h=(V,E) où V est l'ensemble des sommets de G_h et E l'ensemble de ses arcs. L'ensemble des V se décompose en quatre sous-ensembles correspondant à quatre types de machines à état fini (FSM) appelés respectivement *opérateur* (V_O) , *communicateur* (V_C), mémoire (V_M) et *Bus/Mux/Demux* avec ou sans arbitre (V_B): $V = V_O \cup V_C \cup V_M \cup V_B$, et $V_O \cap V_C \cap V_M \cap V_B = \varnothing$. Chaque arc $s \in E$ représente une connexion entre les entrées et les sorties des FSMs, le graphe d'architecture forme ainsi un réseau d'automates. Un sommet mémoire est soit de type RAM (Random Access Memory, $V_{RAM} \in V_M$) soit de type SAM (Sequential Access Memory, $V_{SAM} \in V_m$). Les RAM sont utilisables pour stocker le code des opérations du graphe d'algorithme (Cf. section 3), ils sont alors notés RAM_P (mémoires programme). Quand ils sont utilisés pour stocker des données, on les note RAM_D (mémoires de données) et quand ils stockent programmes et données on les note RAM_{DP}. Quand ils sont partagés (connectés à plusieurs opérateurs et/ou communicateurs), les sommets mémoires servent à communiquer des données. Les mémoires de type SAM sont uniquement utilisées pour communiquer des données selon le paradigme du passage de messages. Une SAM impose par définition que les données soit temporellement lues dans le même ordre qu'elles y ont été écrites (ordre FIFO). Les accès y sont qualifiés de synchronisés alors que dans les RAM il est possible de lire indépendamment de l'ordre d'écriture, les accès sont qualifiés de non synchronisés. Une SAM peut être de type point-à-point, multipoints et supporter ou non le broadcast matériellement. Chaque sommet mémoire est caractérisé par sa taille et sa bande passante. Les sommets opérateurs et communicateurs reposent sur un automate séquentiel : chaque opérateur exécute séquentiellement un sous-ensemble fini des opérations du graphe d'algorithme stockées dans une mémoire RAM_P (ou RAM_{DP}) qui doit être connectée à l'opérateur. La durée d'exécution d'une opération o sur un opérateur s_o dépend des caractéristiques de l'opérateur, elle est notée $\Delta(o,s_o)$. Une opération exécutée par un opérateur lit ses données d'entrée stockées dans une RAM_D (ou RAM_{DP}) connectée à cet opérateur et produit des données de sortie pour les stocker dans une des RAM_P (ou RAM_{DP}) connectée à ce même opérateur (l'opérateur peut aussi lire et écrire dans la même RAM). Chaque communicateur exécute séquentiellement des *opérations de communication* stockées dans une mémoire RAM_P (ou RAM_{DP}) connectée au communicateur. Ces opérations transfèrent les données entre une mémoire (SAM, RAM_D, RAM_{DP}) connectée au communicateur et une autre mémoire elle aussi connectée au sommet communicateur (chaque communicateur est connecté au minimum à deux sommets V_M). La durée d'exécution d'une opération de communication se calcule à partir du type et de la quantité des données à transférer, ainsi que des caractéristiques des communicateurs et mémoires utilisés. Quand un sommet séquenceur (opérateur ou communicateur) doit accéder à plus d'une mémoire, il faut ajouter un multiplexeur démultiplexeur aux bus d'adresses et de données. Nous le modélisons à l'aide d'un sommet *bus/multiplexeur/démultiplexeur* (BMD) inséré entre le sommet séquenceur et les sommets mémoires. Quand une mémoire est partagée par plusieurs sommets séquenceurs, des conflits d'accès peuvent se produire. Il faut donc arbitrer l'accès aux mémoires partagées à l'aide d'un arbitre que nous modélisons par un BMD incluant un arbitre. Ce BMD doit être inséré entre tous les séquenceurs et la mémoire partagée. L'arbitrage et ses caractéristiques (lois d'arbitrage etc.) sont modélisés par une fonction et stockées dans des tables afin de calculer la bande passante disponible à chaque instant afin d'en déduire les durées de communications. Les protocoles d'arbitrage actuellement supportés sont les arbitrages à priorités fixes et tournantes qui régissent la plupart des DMA utilisés (TMS320C40, ADSP21060, etc.).

La figure 1-a représente le graphe d'une architecture composée d'un unique processeur capable d'accéder à deux bancs mémoires de caractéristiques différentes. La figure 1-b

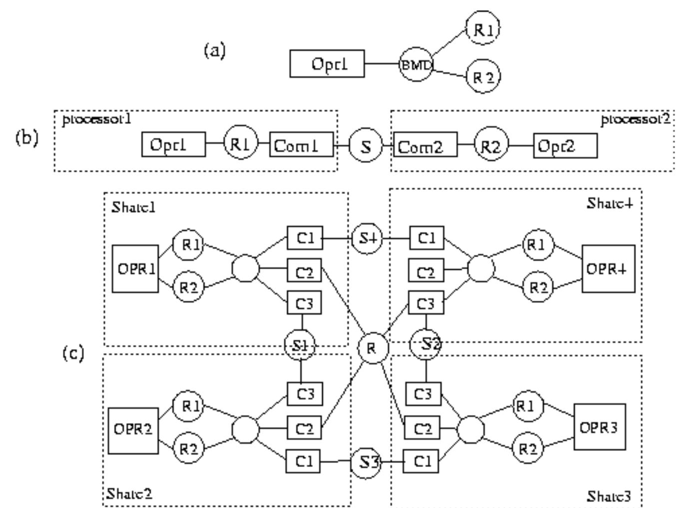

Figure 1 : trois exemples de graphes d'architectures

représente le graphe d'une architecture composée de deux processeurs accédant chacun à une mémoire locale, et communicant par un bus série. La figure 1-c représente le graphe d'une architecture faite de quatre DSP Analog Device (Sharc ADSP21060) capables de communiquer à la fois par quatre liens série (S1 to S4) et une mémoire partagée (R au centre du graphe).

3. Modèle d'algorithme

Pour modéliser l'algorithme à implanter, et pour mettre en évidence son parallélisme potentiel, AAA est basée sur le modèle classique de graphe flot de données. Afin de pouvoir spécifier et prendre en compte les spécificités d'éventuelles parties répétitives (boucles) de ces graphes, ainsi que le cas de calculs exécutés conditionnellement (if..then..else), nous avons enrichi ce modèle que nous appelons graphe flot de données factorisé et conditionné [3], [4]. Les sommets de notre graphe d'algorithme sont des opérations d'entrée-sortie, de calcul ou de répétition. Les sommets de calcul transforment les données transmises au moyen des arcs qui spécifient

donc aussi une relation de précédence d'exécution entre les opérations productrices et consommatrices.

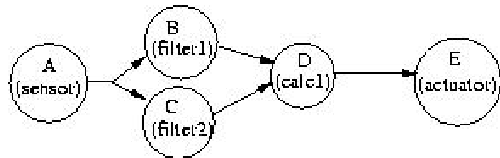

Figure 2 : exemple simplifié de graphe d'algorithme

La figure 2 présente un graphe simple d'algorithme sans conditionnement ni répétition composé de deux opérations d'entrée sortie (A et E) et trois opérations de calcul (B,C et D).

4. Modèle d'implantation

A partir d'un graphe d'algorithme et d'un graphe d'architecture, il est possible de construire un grand nombre de graphes d'implantation Nous appelons distribution la première étape de l'implantation. Elle consiste à faire une partition des éléments du graphe de l'algorithme en fonction du graphe de l'architecture. Il faut en effet définir pour chaque opération (resp. dépendance de données), l'opérateur (resp. le chemin dans le graphe d'architecture) qui va l'exécuter. Pour exécuter une opération de calcul o (resp. de communication), un opérateur (resp. un communicateur) doit lire ses instructions dans une mémoire programme RAM_P qui lui est connectée : nous modélisons l'allocation de cette mémoire programme en commençant par ajouter un sommet *allocation* ($alloc_{Po}$) dans le graphe d'algorithme puis en l'associant à la mémoire allouée RAM_P. L'association entre l'opération o et ce sommet allocation est modélisé par un arc non-orienté. Si o requiert des variables locales pour son exécution, nous ajoutons un sommet allocation noté $alloc_{lo}$ que nous associons cette fois à une mémoire dédiée au stockage des données : RAM_D. L'exécution d'une opération par un opérateur consiste en la lecture de ses données d'entrée dans une des mémoires données (RAM_D, RAM_{DP}) connectées à l'opérateur, puis à combiner ces données pour calculer des données de sortie qui sont finalement écrites à leur tour dans l'une des mémoires connectées à cet opérateur. Ainsi, quand deux opérations exécutées par le même opérateur sont en dépendance de données (i.e. connectées par un arc), l'opération productrice doit être exécutée en séquence avant l'opération consommatrice des données transmises. Nous modélisons l'allocation des données transmises entre producteurs et consommateurs en ajoutant un sommet allocation $alloc_D$ associé à la mémoire allouée. Ce sommet $alloc_D$ est associé à l'ensemble de ses sommets producteur et consommateur(s) par un hyperarc. Quand deux opérations en dépendances de données sont exécutées par des opérateurs différents, la dépendance est dite *inter-opérateur*. Pour implanter une telle dépendance, il faut choisir une route (chemin dans le graphe d'architecture) qui connecte une mémoire de l'opérateur producteur à une mémoire de l'opérateur consommateur. Ensuite, les données doivent être transférées de la mémoire RAM de l'opérateur producteur vers la mémoire RAM de l'opérateur consommateur. Pour cela il faut, pour chaque communicateur (resp. BMD) de la route, créer et insérer une opération de communication[1] (resp. opération identité) dans le graphe d'algorithme. Il faut aussi ajouter et associer des sommets allocation à chacun des sommets mémoire de la route. La deuxième étape de l'implantation, l'ordonnancement, consiste à construire un ordre total dans chaque partition associée à chaque opérateur et communicateur puisqu'ils reposent sur des séquenceurs. Pour cela on ajoute, si nécessaire, des précédences d'exécution entre les opérations d'une même partition en vérifiant que l'ordre total ainsi créé inclut l'ordre partiel du graphe de l'algorithme initial.

La figure 3 donne un exemple de graphe d'implantation de l'algorithme de la figure 2 sur l'architecture de la figure 1-b.

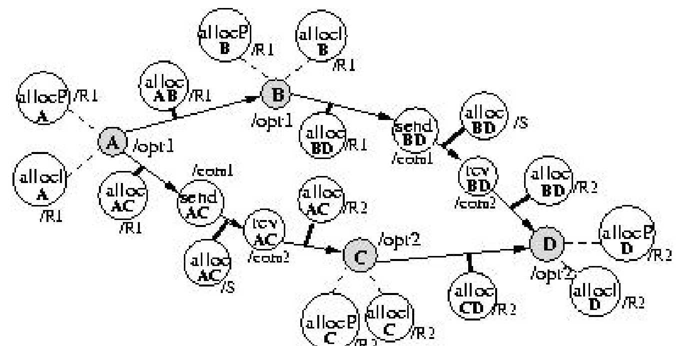

Figure 3 : exemple de graphe d'implantation

5. Optimisation et exécutif

Pour un couple donné de graphes d'algorithme et d'architecture, il existe un ensemble très vaste, mais fini, de graphes d'implantation possibles. Parmi ceux là, nous recherchons celui qui minimise les ressources utilisées et qui satisfait nos contraintes temps réel. C'est un problème d'optimisation qui, comme la plupart des problèmes d'allocation de ressources est connu pour être NP-difficile. Le nombre de cas à traiter étant très élevé pour réaliser des applications réelles, nous recherchons des solutions approchées à l'aide d'heuristiques. Celle que nous utilisons [1] est basée sur un algorithme de liste glouton, associé à une fonction de coût qui prend en compte la durée d'exécution des opérations et des communications pour indiquer le degré d'urgence qu'il y a à placer une opération. La description détaillée de cet algorithme n'est pas l'objet de cet article. Indiquons simplement qu'il requiert une phase préalable de caractérisation dans laquelle on associe une durée d'exécution à chaque opération de calcul (cette durée est fonction de l'opérateur qui est capable de l'exécuter). Ainsi, après exécution de notre heuristique, nous obtenons un graphe d'implantation complet étiqueté par des durées. A partir de ce graphe étiqueté, il est possible de cons-

[1] send et receive pour une paire de communicateurs connectés à une SAM, read et write pour une paire de communicateurs connectés à une RAM

truire un diagramme temporel d'exécution de l'algorithme dans lequel la taille verticale des boites correspond à la durée d'exécution des sommets (Cf. figure 1-a), ainsi qu'un diagramme temporel d'utilisation de chaque mémoire (Cf. figure 1-b pour le sommet mémoire R1). En explorant séquentiellement les sommets associés à chaque ressource (opérateur, communicateurs, mémoires) nous pouvons construire automatiquement l'exécutif distribué de l'application [2], [5].

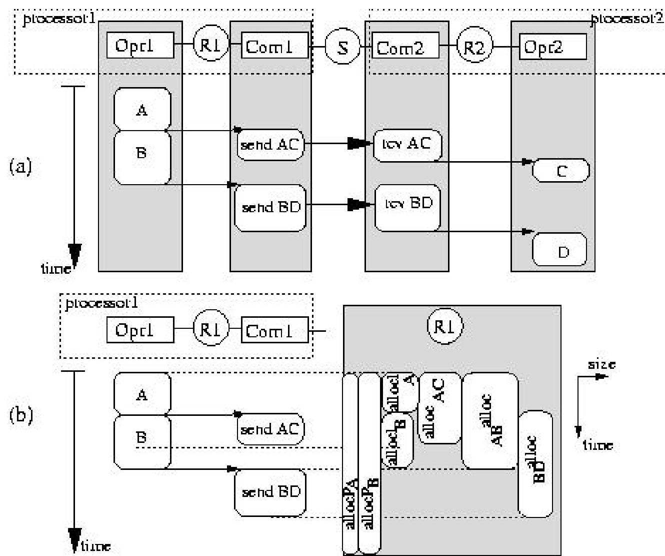

Figure 4 : diagrammes temporels d'allocation

6. Conclusion

Dans cet article nous avons proposé un modèle générique d'architecture distribuée permettant de spécifier aussi bien des communications par passage de messages (cas des SAM) que par mémoires partagées (cas des RAM partagées). Nous avons montré que ce modèle permettait de formaliser l'implantation d'un algorithme sur une architecture en terme de transformation de graphes en faisant apparaître à la fois l'allocation de chaque mémoire et de chaque ressource séquentielle. Enfin, avec le même formalisme, il est possible de prédire et optimiser cette implantation, et d'en générer un exécutif distribué. Ce travail a été validé par son implantation dans le logiciel de CAO niveau système ``SynDEx'' [1] utilisé, entre autre, pour développer des applications sur architectures à base de Texas TMS320C40[6], de TMS320C6x, de PowerPC Motorala MPC555[7], ainsi que pour modéliser un SoC (System on a Chip) [8]. Nous travaillons actuellement sur une extension de ce modèle d'architecture pour prendre en compte les circuits reconfigurables .

7. Bibliographie

[1] T. Grandpierre, C. Lavarenne, Y. Sorel - *Optimized Rapid Prototyping For Real Time Embedded Heterogeneous Multiprocessors* - CODES'99, IEEE/ACM 7th Int. Workshop on Hardware/Software Co-Design, Rome Italie, mai 1999.

[2] T. Grandpierre - *Modèle d'architecture parallèle hétérogène pour la génération automatique d'exécutif temps réel optimisé* - Univ. Paris XI - Orsay, novembre 2000.

[3] C. Lavarenne, Y. Sorel - *Modèle unifié pour la conception conjointe logiciel/matériel* - Revue Traitement du Signal, vol. 14/6, 1997.

[4] A. Vicard - *Formalisation et optimisation des systèmes informatiques distribués temps réel embarqués* - Université Paris XIII, Juillet 1999.

[5] T. Grandpierre, C. Lavarenne, Y. Sorel – *Modèle d'exécutif distribué temps réel pour SynDEx* - Rapport de Recherche INRIA n°3476, Août 1998.

[6] V. Fresse, M. Assouil, O. Desforges - *Rapid prototyping for mixed architectures* - proc. of IEEE Int. Conf. on Acoustics, Speech, and Signal Processing, Istanbul, Turquie, Juin 2000.

[7] W. Mooncheol, T.Grandpierre, G. Fleutot, Michel Parent - *A Joystick Driving Algorithm with a Collision Stop Feature on an Electric Vehicle (Cycab)* - IEEE Intelligent Vehicle Symposium, Juin, 2002.

[8] M. Barreteau, P. Bonnot, T. Grandpierre, P. Kajfasz, C. Lavarenne, J. Mattioli, Y. Sorel - *PROMPT : A Mapping Environment for Telecom Applications on ``System on a Chip''* - CASE2000, Int. Conf. on Compilers, Architecture and Synthesis for Embedded Systems, San Jose, Cal. US, novembre 2000.