Modèle unifié pour la conception conjointe logiciel-matériel

A Unified Model for Software-Hardware Co-design

C. Lavarenne, Y. Sorel

INRIA Rocquencourt, B.P.105 – F-78153 Le Chesnay Cedex
christophe.lavarenne@inria.fr, yves.sorel@inria.fr
http://www-rocq.inria.fr/syndex

Traitement du Signal 1997 - vol.14 no.6 pp 569–578,
numéro spécial "Adéquation Algorithme Architecture"

Résumé

On propose un modèle unifié de graphes factorisés, pour spécifier et optimiser des applications temps réel embarquées, basées sur des architectures composées de processeurs et/ou de circuits spécialisés. Tout d'abord on utilise un graphe de dépendances de données entre opérations pour spécifier l'ordre partiel des opérations de l'algorithme et donc son parallélisme potentiel, indépendamment des contraintes matérielles. On montre ensuite comment ce graphe peut être transformé par différentes formes de factorisation pour aboutir à une implantation sous forme de circuits spécialisés ou d'un exécutif spécialisé distribué sur des processeurs. Enfin on donne les principes de base de l'optimisation visant à minimiser les ressources matérielles tout en respectant les contraintes temps réel. On présente en perspective comment cette approche unifiée pourra conduire à l'optimisation de la conception conjointe logiciel-matériel.

Mots clés : applications temps réel embarquées, conception conjointe logiciel-matériel, graphes de dépendances factorisés, synthèse de circuits, génération d'exécutifs distribués, implantation optimisée.

Abstract

A unified model of factorized graphs is proposed for the specification and the optimization of real-time embedded applications based on architectures composed of processors and/or specific circuits. First, a graph of operations partially ordered by their data dependencies is used to specify the algorithm and hence its potential parallelism, independently of hardware constraints. Then, it is shown how this dependence graph may be transformed by different kinds of factorization to obtain an implementation, as specific circuits or as a specialized executive distributed on several processors. Finally, basic principles of optimization are given for minimizing hardware resources while satisfying real-time constraints. In prospect, this unified approach is expected to be used for optimized software-hardware co-design.

Keywords: embedded real-time applications, software-hardware co-design, factorized dependence graphs, circuit synthesis, distributed executives generation, optimized implementation.

1 Contexte

1.1 Systèmes réactifs temps réel embarqués

Dans les applications temps réel embarquées, de contrôle/commande et de traitement du signal et des images, on met en œuvre des systèmes informatiques qui interagissent avec leur environnement physique. Le système informatique doit être réactif [1], c'est-à-dire produire une commande en réaction à chaque variation d'état de l'environnement.

Par système informatique, on entend ici à la fois le logiciel (applicatif et système) et le matériel, comprenant des capteurs pour acquérir l'état de l'environnement, des actionneurs pour le commander, et un calculateur pour exécuter l'algorithme de contrôle. L'application correspond au couple système informatique et environnement, soumis à un ensemble de contraintes, comme des contraintes de performances temps réel (latence, cadence), des contraintes d'embarquabilité (volume, poids, consommation électrique etc ...) et des contraintes de coût d'études et de fabrication. Pour le concepteur de l'application, il s'agit de garantir que le comportement de l'application corresponde aux spécifications et que les contraintes soient respectées. Par exemple, les images acquises par un capteur CCD d'un système d'aide à la conduite dans les automobiles du futur, sont traitées par un calculateur embarqué, en vue de repérer les véhicules qui le précèdent pour avertir le conducteur, par une alarme sonore et visuelle, quand la situation relative des véhicules ne respecte plus une distance de sécurité. Le contrôle peut être plus complexe si un système de ralentissement automatique doit agir sur le moteur et sur les freins.

Le domaine de contrôlabilité de l'environnement impose une borne supérieure sur la durée du calcul (latence) entre une détection de variation d'état et la variation induite de la commande. Les systèmes informatiques étant numériques, les signaux d'entrée-sortie sont discrétisés (échantillonage, blocage), aussi bien dans l'espace des valeurs que dans le temps. Le domaine d'observabilité de l'environnement impose également une borne supérieure sur le délai qui s'écoule entre deux échantillons (cadence). En plus de ces contraintes temps réel, le système informatique est soumis à des contraintes technologiques d'embarquabilité et de coût incitant à minimiser les ressources matérielles nécessaires à sa réalisation.

C'est ce problème d'optimisation (minimisation de ressources matérielles) sous contraintes (temps réel et technologiques) qui fait l'objet de nos recherches. Nous avons développé une méthodologie appelée "Adéquation Algorithme Architecture" qui aborde ce problème de manière globale en unifiant les modèles d'algorithmes et d'architectures à base de processeurs [2]. Dans cet article, nous étendons cette unification aux circuits intégrés non programmables.

1.2 Architectures spécialisées

Bien que les processeurs d'usage général soient de plus en plus performants, grâce à l'augmentation très rapide de leur fréquence d'horloge et du nombre des transistors qui les composent, certaines applications de traitement du signal et des images nécessitent une puissance de calcul largement supérieure à celle actuellement disponible sur les processeurs les plus rapides utilisés au cœur des stations de travail. Pour satisfaire ce besoin très important de puissance de calcul, ou bien pour prendre en compte la délocalisation de certaines fonctionnalités lorsque l'on cherche à rapprocher les organes de calcul le plus près possible des capteurs et des actionneurs afin de limiter les problèmes liés au transport des signaux analogiques, des calculateurs à architecture parallèle, répartie ou distribuée sont nécessaires. Par exemple, dans le domaine du sonar ou du radar multi-voies il est parfois nécessaire d'utiliser une centaine de processeurs de traitement du signal, principalement pour résoudre les besoins de puissance de calcul. Dans le domaine de l'automobile on aura rapide-

ment à mettre en œuvre quelques dizaines de microcontrôleurs afin de rapprocher ces derniers des capteurs (température, pression, richesse etc ...) et des actionneurs (injection, suspension, freins etc ...) en vue de limiter le câblage, ou bien dans les transports aériens pour atténuer les effets des émissions électromagnétiques. Dans ces domaines, on ne peut pas utiliser des calculateurs généralistes conçus principalement pour le calcul scientifique. Il faut réaliser des calculateurs spécialisés construits à partir de processeurs s'interfaçant directement avec des capteurs et des actionneurs.

De plus, les contraintes temps réel et d'embarquabilité peuvent être tellement fortes que les processeurs disponibles sur le marché ne suffisent pas. Cela conduit à utiliser, en complément des processeurs, des circuits intégrés spécialisés (ASIC figés ou FPGA reconfigurables) qui réalisent généralement des fonctionnalités dites de bas niveau, intégrables dans des circuits parce qu'elles n'utilisent qu'un nombre très restreint de types différents d'opérations, utilisées de manière intensive et régulière. Les fonctionnalités qui utilisent, de manière très irrégulière, un nombre important de types d'opérations différentes et complexes, ne sont intégrables qu'en décomposant chaque type d'opération en une séquence d'opérations arithmétiques et logiques élémentaires communiquant par l'intermédiaire de mémoires temporaires (registres), c'est-à-dire en utilisant des jeux et des séquenceurs d'instructions de processeurs.

1.3 Conception conjointe logiciel-matériel

La conception conjointe logiciel-matériel ou "co-design" consiste à choisir pour chaque fonctionnalité soit une implantation sur circuits intégrés (qui évoluera peu), soit une implantation programmée sur des processeurs (qui évoluera beaucoup ou dont la complexité ne permet pas une implantation sur circuit). Ce compromis est difficile à réaliser, d'autant plus qu'il peut faire intervenir des considérations commerciales très fluctuantes. Ce dernier point est crucial dans le domaine des applications orientées vers le grand public, telles que les télécommunications mobiles, l'automobile etc... où les aspects prix de revient/satisfaction du client sont importants. Dans le domaine des applications militaires (radar, sonar, aviation, système d'armes ...), les aspects coûts sont moins importants, mais les fonctionnalités évoluent rapidement et le critère d'embarquabilité est prépondérant. Le problème de co-design apparaît aussi maintenant dans les domaines tels que l'aviation civile ou le transport ferroviaire. Dans les domaines militaire et civil, le compromis doit de plus prendre en compte des contraintes d'utilisation de composants logiciels et matériels standards (COTS).

1.4 Méthodologie "Adéquation Algorithme Architecture"

Il est tout d'abord nécessaire de préciser le sens que l'on donnera par la suite aux notions d'algorithme, d'architecture, d'implantation et d'adéquation.

Un algorithme, dans l'esprit de Turing [3], est une *séquence finie d'opérations* (réalisables en un temps fini et avec un support matériel fini). Ici, on étend la notion d'algorithme pour prendre en compte d'une part l'aspect *infiniment répétitif* des applications réactives et d'autre part l'aspect *parallèle* nécessaire à leur implantation distribuée. Le nombre d'interactions effectuées par un système réactif avec son environnement n'étant pas borné a priori, il peut être considéré infini. Cependant, à chaque interaction, le nombre d'opérations (nécessaires au calcul d'une commande en réaction à un changement d'état de l'environnement) doit rester borné, parce que les durées d'exécution sont bornées par les contraintes temps réel. Mais au lieu d'un ordre total (séquence) sur les opérations, on se contente d'un *ordre partiel*, établi par les dépendances de données entre opérations, décrivant un *parallélisme potentiel* inhérent à l'algorithme, indépendant du *parallélisme disponible* du calculateur. Le terme *opération* correspond ici à la notion de fonction dans la théorie

des ensembles, on le distingue volontairement de la notion d'*opérateur*, qui a ici un autre sens lié aux aspects implantation matérielle. L'algorithme est le résultat de l'approche formelle mathématique du problème consistant à décrire une solution que pourra donner un calculateur. Il sera codé, à différents niveaux, par des langages plus ou moins éloignés du matériel. Comme on peut obtenir, pour un algorithme donné, de nombreux codages différents selon le langage et l'architecture choisis, on préfère utiliser ce terme générique d'algorithme indépendant des langages et des calculateurs.

L'architecture correspond aux caractéristiques structurelles du calculateur, exhibant un *parallélisme disponible*, en général moindre que le *parallélisme potentiel* de l'algorithme. Par abus de langage le terme architecture sera ici souvent utilisé dans son sens générique pour signifier à la fois le calculateur lui-même ainsi que ses caractéristiques structurelles.

L'implantation consiste à mettre en œuvre l'algorithme sur l'architecture, c'est-à-dire à compiler, charger, puis lancer l'exécution sur le calculateur, avec dans le cas des processeurs le support d'un système d'exploitation ou d'un exécutif dont le surcoût est à ne pas négliger.

Enfin l'adéquation consiste à mettre en correspondance de manière efficace l'algorithme et l'architecture pour réaliser une implantation optimisée. On utilisera par abus de langage dans la suite, cette notion d'implantation optimisée bien qu'on ne puisse pas garantir l'obtention d'une solution optimale pour ce type de problème. On se contentera donc d'une solution approchée obtenue rapidement, plutôt que d'une solution exacte obtenue dans un temps rédhibitoire à l'échelle humaine à cause de la complexité combinatoire exponentielle de la recherche de la meilleure solution.

2 Modèle unifié

D'un point de vue purement spéculatif, si l'on supposait qu'on dispose d'un jeu infiniment riche d'opérations (au sens mathématique) couvrant tous les besoins imaginables, ainsi que d'un jeu correspondant d'opérateurs (au sens matériel) réalisant ce jeu d'opérations, tout algorithme pourrait être spécifié en une seule de ces opérations, et il suffirait d'utiliser l'opérateur correspondant pour réaliser la transformation d'un ensemble de données en l'ensemble des résultats correspondant. D'un point de vue un peu plus réaliste, l'ensemble des opérations pour lesquelles on dispose d'opérateurs étant limité, il est nécessaire de décomposer l'algorithme en termes de cet ensemble d'opérations.

2.1 Graphes de dépendances

Une opération, au sens mathématique pris dans un contexte numérique (par opposition à analogique), combine des données fournies en entrée pour fournir en sortie des résultats. Un opérateur, au sens matériel pris dans un contexte numérique, réalise, implante une opération.

Chaque opération se caractérise par la liste des types (codage numérique) de ses entrées et sorties, et par sa combinatoire qui calcule les valeurs des sorties à partir de celles des entrées : les valeurs des sorties *dépendent de* celles des entrées (dépendances "opératoires" intra-opération). La composition d'opérations consiste à fournir à une entrée d'une opération la valeur d'une (et une seule) sortie d'une autre opération : la valeur de l'entrée *dépend de* celle de la sortie (dépendance "de données" inter-opérations). La valeur d'une sortie peut par contre être donnée à plusieurs entrées (diffusion). La composition ne doit pas produire de cycle de dépendances (déterminisme).

En termes algébriques, chaque valeur correspond à une "variable" mathématique, donnée ou inconnue, chaque dépendance opératoire correspond à la définition d'une variable inconnue en termes d'autres variables, chaque dépendance de données correspond à l'utilisation d'une même variable dans plusieurs définitions, et le déterminisme correspond à l'existence d'un ordre entre

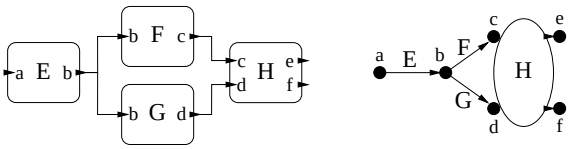

FIG. 1 – *Graphes duaux de dépendances*

définitions tel que chacune ne s'exprime qu'en termes de variables définies précédemment, c'est-à-dire consiste à exiger que les problèmes de points fixes aient été éliminés.

La décomposition d'un algorithme en opérations implantables (c'est-à-dire pour lesquelles on dispose d'opérateurs) consiste à construire, par composition d'opérations, un graphe de dépendances de données (hyperarcs) entre opérations (sommets), ou dualement un graphe de dépendances opératoires (hyperarcs) entre valeurs (sommets) [1]. La figure 1 montre les deux formes duales de décomposition d'un exemple d'algorithme en quatre opérations E, F, G, H (l'hyperarc H a pour origines $\{c, d\}$ et pour extrémités $\{e, f\}$) calculant les valeurs $\{e, f\}$ à partir de la valeur $\{a\}$ par l'intermédiaire des valeurs $\{b, c, d\}$. La première forme est préférée, car elle est plus lisible dans le cas de graphes plus complexes, et plus proche d'un circuit.

L'architecture réalisant l'algorithme peut être obtenue par compilation d'une "net-list" synthétisée par traduction directe du graphe de dépendances de données entre opérations (par exemple en VHDL [5] structurel), en remplaçant chaque opération par l'opérateur qui l'implante (instanciation d'un composant de bibliothèque VHDL), et chaque dépendance de données par un média de communication (signal VHDL) interconnectant les ports des opérateurs. Le parallélisme potentiel, lié à l'ordre partiel établi par les dépendances de données entre opérations, devient un parallélisme effectif entre opérateurs. Cependant, chaque opérateur n'étant utilisé qu'une seule fois, la traduction directe d'un simple graphe de dépendance en architecture est inefficace : on préfère généralement une formalisation de l'algorithme permettant une réduction de la taille de l'architecture en utilisant chaque opérateur plusieurs fois avec des données différentes.

2.2 Factorisations de graphes finis

Le processus de décomposition d'un algorithme en opérations implantables génère souvent des répétitions périodiques de motifs d'opérations (identiques mais opérant sur des données différentes), que l'esprit humain, se lassant vite des énumérations, préfère spécifier sous forme factorisée [6].

Par exemple, le produit $s = ce$ d'un vecteur $e \in \mathbb{R}^n$ (aussi noté $e = (e_1 \cdots e_n) = (e_j)_{1 \le j \le n}$) par une matrice $c \in \mathbb{R}^m \times \mathbb{R}^n$ (composée de m vecteurs : $c = (c_i)_{1 \le i \le m}$ avec $c_i = (c_{ij})_{1 \le j \le n}$), de résultat $s \in \mathbb{R}^m$, peut se décomposer en m produits scalaires $s = (c_i \cdot e)_{1 \le i \le m}$ qui peuvent se décomposer chacun en une somme de n produits $c_i \cdot e = c_{i1}e_1 + \cdots + c_{in}e_n = \sum_{1 \le j \le n}(c_{ij}e_j)$.

En notation algébrique, comme le montre cet exemple, l'énumération est soit intuitivement abrégée par des points de suspension (comme $e_1 \cdots e_n$), soit plus formellement factorisée par des foncteurs (opérations sur les opérations, comme $\sum_{1 \le j \le n}$ et $(\ )_{1 \le i \le m}$) spécifiant un nombre de répétitions et un indice utilisé pour indexer les symboles représentant les ensembles de valeurs regroupées par la factorisation.

La figure 2 présente le graphe de dépendances du produit matrice-vecteur pour $m = n = 3$

1. Dans les hypergraphes [4], une hyperarête (non orientée) peut relier plus de deux sommets, et un hyperarc (orienté) se définit par deux ensembles de sommets "origines" et "extrémités". L'hyperarc d'une valeur ne peut avoir qu'un seul sommet origine, ou dualement le sommet d'une valeur ne peut être l'extrêmité que d'un seul hyperarc.

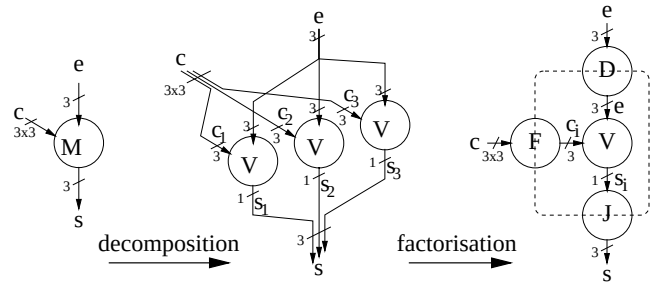

FIG. 2 – *Décomposition et factorisation d'un produit matrice-vecteur*

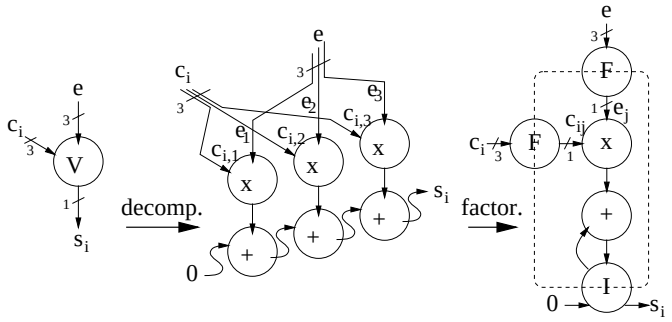

FIG. 3 – *Décomposition et factorisation d'un produit scalaire*

(à gauche), sa décomposition en produits scalaires (au centre), et sa factorisation (à droite) qui fait apparaître trois sommets spéciaux (D pour "Diffusion", F pour "Fork" et J pour "Join") qui délimitent la frontière, mise en évidence par des pointillés, entourant le motif de la factorisation pour le séparer du reste du graphe. La figure 3 présente le graphe de dépendances du produit scalaire, sa décomposition en somme de produits et sa factorisation qui fait apparaître le nouveau sommet frontière I (pour "Iterate"). Chaque sommet frontière spécifie l'une des différentes manières de factoriser les arcs en traversant la frontière de factorisation :

- le sommet D "entre" en diffusant le vecteur e à tous les produits scalaires
  (il correspond à la non-indexation par i de e dans $(c_i \cdot e)_{1 \leq i \leq m}$)
- le sommet F "entre" en factorisant le groupe d'arcs d'entrée $(c_i)_{1 \leq i \leq 3}$
  (il correspond à l'indexation par i de c dans $(c_i \cdot e)_{1 \leq i \leq m}$)
- le sommet J "sort" en factorisant le groupe d'arcs de sorties $(s_i)_{1 \leq i \leq 3}$
  (il correspond au foncteur de regroupement $(\ )_{1 \leq i \leq m}$)
- le sommet I "entre et sort" (par 0 et s_i) en factorisant le groupe d'arcs inter-motifs
  (il correspond à un foncteur de réduction, qui associé au $+$ correspond au foncteur $\sum_{1 \leq j \leq n}$, et qui associé au $\times$ correspondrait au foncteur $\prod_{1 \leq j \leq n}$, etc ...)

Il est important de noter que la factorisation d'un graphe de dépendances ne change *en rien* l'ordre partiel, et donc le parallélisme potentiel, entre les opérations du graphe. La factorisation d'un motif répétitif n'est *pas équivalent* à une boucle dans un langage impératif, qui impose un ordre total sur les itérations, même dans le cas d'un code à assignation unique. Elle n'est pas non

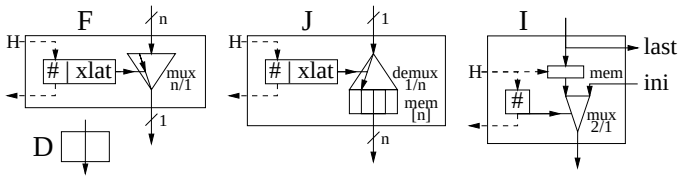

FIG. 4 – *Structure matérielle des opérateurs de multiplexage*

plus l'équivalent d'une projection, au sens de Kung [7], du graphe de dépendances sur un graphe flots de signaux, qui impose un ordonnancement des opérations suivant la direction de projection (notons aussi que Kung ne s'intéresse qu'aux graphes de dépendances entièrement réguliers, "shift-invariants", que l'on pourrait qualifier de "mono-cristallin", c'est-à-dire dont les sommets occupent de manière dense une grille homogène multi-dimensionnelle, alors que nous nous intéressons à des graphes de dépendances quelconques, pouvant contenir plusieurs parties régulières ainsi que des parties irrégulières, que l'on pourrait qualifier de "poly-cristallin"). La factorisation ne fait "que" réduire la taille de la spécification du graphe et mettre en évidence ses parties régulières (motifs périodiques). Les sommets frontières ne sont que les délimiteurs d'une "syntaxe graphique", analogues aux parenthèses qui délimitent la portée d'un foncteur dans la syntaxe algébrique. De la même manière que pour deux paires de parenthèses, deux frontières de factorisation ne peuvent être que soit l'une dans l'autre (en relation d'inclusion), soit l'une à côté de l'autre (en relation d'exclusion), elles ne peuvent être en relation d'intersection.

La factorisation n'a pas pour seul intérêt de réduire la taille des spécifications : elle permet aussi de réduire dans les mêmes proportions la taille des architectures obtenues par traduction directe du graphe de dépendances factorisé. Une opération à l'intérieur d'une frontière de factorisation, qui représente un groupe factorisé d'opérations identiques opérant sur un groupe factorisé de données différentes, se traduit directement par un seul opérateur utilisé itérativement, autant de fois qu'il y a d'opérations dans le groupe factorisé. Chaque groupe factorisé de données doit donc être multiplexé à la frontière : alors qu'ils ne jouent qu'un rôle "syntaxique" au niveau du graphe de dépendances, les opérateurs correspondant aux sommets frontières doivent réaliser le multiplexage (sauf l'opérateur D qui se contente de fournir la même valeur à chaque itération).

Le multiplexage nécessitant un choix d'ordonnancement (compatible avec l'ordre partiel), la traduction directe d'un graphe de dépendances factorisé en architecture (implantation) correspond à une réduction du parallélisme potentiel de l'algorithme. Comme on le verra au chapitre 3, si cette réduction ne permet pas de respecter les contraintes temps réel, on peut obtenir une architecture avec un parallélisme moins réduit en transformant la factorisation de l'algorithme avant implantation.

La figure 4 présente schématiquement la structure matérielle des opérateurs de multiplexage correspondant aux sommets frontières. Comme tout multiplexage implique un séquencement temporel, les opérateurs F, J et I comprennent un compteur périodique (schématisé par un "#") cadencé par l'horloge d'entrée H et produisant une horloge de sortie dans un rapport à l'horloge d'entrée égal à la période du compteur [2] (n sur la figure). Les relations d'horloges entre compteurs sont déduites des dépendances de données entre frontières. La synthèse de ces relations est décrite en détail au chapitre 2.4 pour un cas particulier de factorisation. Leur synthèse dans le cas général

2. Les compteurs des opérateurs frontières d'une même frontière étant identiques, il en suffit d'un seul ; un seul par composant dans le cas où le motif factorisé est distribué sur plusieurs composants, car la communication (entre composants) de l'horloge consommera moins de ressources d'interconnexion que celle de la valeur du compteur.

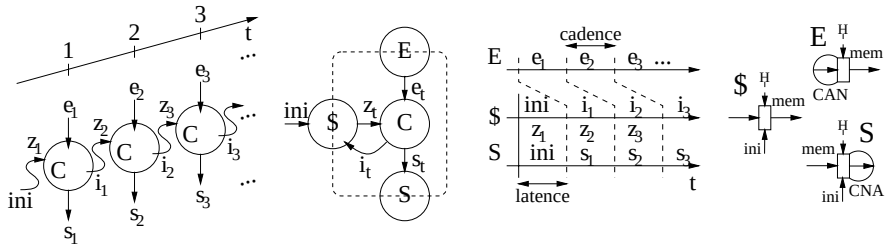

FIG. 5 – *Graphe flot de données et opérateurs d'entrée-sortie*

étant en cours d'étude, nous ne la décrirons pas ici.

Le compteur de l'opérateur F pilote un multiplexeur sélectionnant en sortie l'une des n entrées. Le compteur de l'opérateur J pilote un démultiplexeur sélectionnant l'une de ses n mémoires pour y stocker l'entrée, permettant d'obtenir en sortie simultanément les n valeurs stockées successivement. Les transcodeurs "xlat" accolés aux compteurs permettent, si nécessaire, de générer des séquences de sélection dans un ordre autre que linéaire. L'opérateur I sélectionne périodiquement son entrée *ini* lors de chaque première itération, puis la sortie de la mémoire *mem* lors des $n-1$ autres itérations ; le contenu de la mémoire est mis à jour à chaque itération avec la valeur de l'entrée (qui dépend de la valeur de la sortie puisque l'opérateur I est utilisé pour les dépendances inter-motifs).

2.3 Factorisation de graphes infinis

Les systèmes réactifs numériques interagissent avec leur environnement de manière discrète, donc répétitive (répétition de la séquence acquisition-calculs-commande). Cette répétition présente pour particularités d'une part d'être infinie, et d'autre part de contraindre les données d'entrée et les résultats de sortie à être multiplexés (le plus souvent périodiquement) à travers les capteurs et actionneurs d'interface avec l'environnement. Les sommets frontière F, J et I doivent donc avoir une autre implantation lorsqu'ils assurent l'interface avec l'environnement.

La figure 5 présente un graphe de dépendances répétitif de taille infinie (points de suspension) et sa factorisation. L'opération C représente l'algorithme de contrôle commande (non décomposé) qui à chaque réaction t acquiert une nouvelle entrée e_t par l'intermédiaire du capteur E, la combine avec l'entrée z_t dont la valeur est égale à la sortie i_{t-1} de la réaction précédente (égale à la valeur *ini* lors de la réaction initiale), pour fournir la sortie s_t à l'actionneur S et la sortie i_t à l'entrée z_{t+1} de la réaction suivante.

Un graphe de dépendances infini factorisé est un *graphe flot de données*, algébriquement équivalent à un système d'équations récurrentes (ici : $(i_t, s_t) = C(e_t, z_t) \mid z_t = i_{t-1} \mid i_0 = ini$) qui se code par exemple en langage SIGNAL [8] "`(| {i,s}:=C{e,z} | z:=i$1 init ini |)`".

Les sommets frontières E (de type "Fork"), S (de type "Join") et \$ (de type "Iterate") sont réalisés respectivement par les opérateurs de même nom, schématisés dans la partie droite de la figure 5 accompagnés de leur comportement temporel. L'opérateur E est un capteur échantillonneur-bloqueur cadencé par l'horloge H, constitué d'un convertisseur analogique-numérique (CAN[3]) dont le résultat est maintenu par une mémoire pendant toute la durée d'une réaction. L'opérateur S est un actionneur cadencé par l'horloge H, constitué d'un convertisseur numérique-analogique (CNA) dont l'entrée est maintenue, par une mémoire pendant toute la durée d'une réaction, égale à la valeur fournie en entrée de la mémoire à la fin de la réaction précédente, ou égale à la valeur *ini*

3. CAN et CNA sont pris ici au sens large comprenant les convertisseurs tout ou rien.

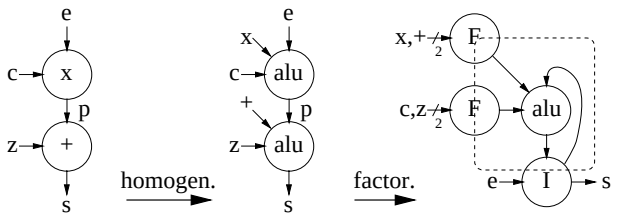

FIG. 6 – *Factorisation après homogénéisation*

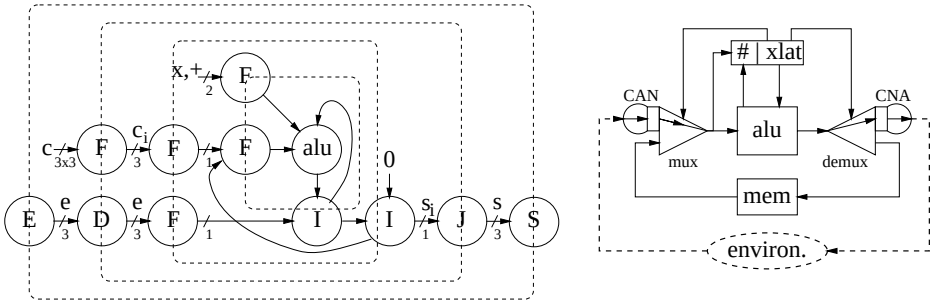

FIG. 7 – *Graphe complet du produit matrice-vecteur et processeur*

lors de la réaction initiale. L'opérateur \$ est une mémoire cadencée par l'horloge H, qui maintient en sortie, pendant toute la durée d'une réaction, la valeur qu'il avait en entrée à la fin de la réaction précédente, égale à la valeur *ini* lors de la réaction initiale.

2.4 Factorisation par homogénéisation

On peut pousser la factorisation à l'extrême en *homogénéisant* toutes les opérations par adjonction à chacune d'elles d'une entrée supplémentaire codant le type d'opération.

Par exemple, la figure 6 montre comment les deux opérations produit et somme du motif du produit scalaire sont remplacées par l'opération "alu" avec une entrée supplémentaire "code opératoire", prenant la valeur "$\times$" pour le produit et "$+$" pour la somme. La factorisation du graphe résultant nécessite un F pour factoriser le groupe (c, z), un I pour factoriser le groupe (e, p, s), et un autre F pour factoriser le groupe $(\times, +)$.

La partie gauche de la figure 7 présente le graphe complet (flot de données, factorisé) de l'exemple du produit matrice-vecteur. Du point de vue matériel, l'opérateur $F(\times, +)$ est un *séquenceur de codes opératoires* sélectionnant les opérations effectuées par l'opérateur "alu" pendant que les autres opérateurs E, F, I, J et S sélectionnent des mémoires. En considérant que les codes opératoires sont stockés dans des mémoires, comme les autres constantes (c et 0), tous les opérateurs E, F, I, J et S sont des *séquenceurs d'accès mémoire*. En regroupant séparément les mémoires, les multiplexeurs, les démultiplexeurs, les transcodeurs ("xlat") et les compteurs ("#"), on obtient l'architecture de la partie droite de la figure 7 qui correspond à une architecture générique de processeur, où le "xlat" correspond à un décodeur d'instructions contenant chacune un code contrôlant le multiplexeur (adresse source), un code opératoire, et un code contrôlant le démultiplexeur (adresse destination), et où le "#" correspond à un séquenceur d'instructions qui incrémente périodiquement un compteur générant les adresses successives des instructions à extraire de la mémoire.

L'ordre d'exécution des instructions est déterminé par trois sources :

- par les dépendances de données, qui requièrent qu'une opération soit exécutée après toutes ses anti-dépendantes, qui lui fournissent ses données d'entrée
- par les transcodeurs ("xlat") qui fixent l'ordre de multiplexage des données ; s'il y a un I sur la frontière du sommet du transcodeur, cet ordre est déterminé par les dépendances de données factorisées par le I, sinon cet ordre peut être choisi de manière arbitraire
- par l'ordre de regroupement des compteurs, qui dépend des relations entre les frontières des compteurs : si les frontières sont en relation d'inclusion, le compteur externe ne s'incrémente que lorsque tous les compteurs internes ont terminé leur période, sinon, s'il y a dépendance de données entre deux frontières, le compteur de la frontière anti-dépendante doit effectuer sa période avant celui de la frontière dépendante, sinon l'ordonnancement relatif des deux compteurs peut être fixé arbitrairement

Le but de la factorisation à l'extrême par homogénéisation n'est pas de synthétiser des architectures génériques de processeurs : les transformations à effectuer semblent trop complexes pour un résultat probablement peu performant comparé aux architectures de processeurs modernes disponibles commercialement. Le but est plutôt de montrer comment, avec le modèle de graphes factorisés utilisé pour synthétiser des net-lists, on peut aussi générer du code séquentiel pour des processeurs.

Génération de code séquentiel

Pour générer du code séquentiel, il n'est pas nécessaire d'effectuer explicitement la factorisation par homogénéisation du graphe, car celle-ci s'effectue plus simplement au niveau du code généré :

- les sommets frontières que générerait la factorisation par homogénéisation se traduisent par des "variables intermédiaires" procurant chacune un espace mémoire identifié par le nom de la variable (le nom correspond à une valeur de sélection d'un dé/multiplexeur)
- chaque opération se traduit par un appel de fonction prenant en argument la liste des noms des variables intermédiaires référençant les espaces mémoires utilisés par l'opération pour y lire ses données et y écrire ses résultats
- les autres sommets frontières de factorisation se traduisent par des délimiteurs de boucles, boucles dont le nombre d'itérations est égal au cardinal de chaque motif factorisé, et dont l'indice est utilisé pour indexer (dé/multiplexer) les variables tableaux qui représentent les groupes de données factorisées
- dans le cas des systèmes réactifs, la boucle la plus externe est infinie.

Par exemple, le peudo-code séquentiel correspondant à l'exemple produit matrice-vecteur est donné figure 8, où les opérations et les variables ont les mêmes noms que les sommets et les arcs des figures, et où les arguments des opérations sont préfixés d'un "?" pour les entrées et d'un "!" pour les sorties.

```
programme ProduitMatriceVecteur
: variable c[3][3]:={...}    \ initialisation matrice constante
: repeterToujours            \ boucle externe infinie
: : variable e[3], s[3]      \ declarations vecteurs
: : E(!e)                    \ acquisition capteur -> e
: : repeterPour i=1..3       \ boucle produits scalaires
: : : s[i]:=0                \ initialisation a chaque iteration
: : : repeterPour j=1..3     \ boucle somme de produits
: : : : variable t           \ variable intermediaire temporaire
: : : : x(?e[j] ?c[i][j] !t) \ produit, resultat dans t
: : : : +(?t ?s[i] !s[i])    \ accumulation de t dans s[i]
: : S(?s)                    \ rafraichissement actionneur <- s
```

FIG. 8 – *Pseudo-code séquentiel du produit matrice-vecteur*

3 Transformations de factorisations de graphes

L'implantation d'un algorithme, obtenue par la méthode présentée dans les paragraphes précédents, peut ne pas satisfaire les contraintes auxquelles sont soumis les systèmes réactifs embarqués. Les contraintes temps réel imposent une borne supérieure sur la latence et la cadence de l'exécution d'un algorithme par une architecture. Les contraintes technologiques imposent une borne supérieure sur la capacité mémoire et la vitesse des composants, circuits ou processeurs. L'ensemble de ces contraintes impliquent souvent la nécessité de distribuer l'algorithme sur plusieurs composants et en conséquence de gérer les communications inter-composants.

Enfin, la minimisation du coût de l'implantation requiert, pour les coûts récurrents de fabrication, une minimisation des ressources matérielles mises en œuvre (surface des circuits et/ou nombre de processeurs), et pour les coûts fixes d'étude, une méthode minimisant les durées de la spécification, de l'implantation et de sa mise au point. La méthode que nous proposons vise en premier lieu ce dernier objectif. Ce chapitre décrit le volet de la méthode visant la minimisation des ressources sous contraintes temps réel en prenant en compte les contraintes technologiques.

3.1 Distribution et ordonnancement

Si l'implantation d'un algorithme ne tient pas sur un seul composant, il faut distribuer l'algorithme sur une architecture multi-composants [2], qu'on peut décrire par un graphe dont les sommets représentent les composants, et dont les hyperarêtes représentent les média de communication bidirectionnelle interconnectant les composants. La distribution consiste à partitionner le graphe de l'algorithme en autant de parties "opérations" qu'il y a de composants, et à partitionner l'ensemble des dépendances de données inter-parties qui en résultent (entre opérations placées sur des composants différents), en autant de parties "dépendances" qu'il y a de médias de communication inter-composants.

Chaque partie "opérations" placée sur un composant séquentiel (processeur) doit de plus être ordonnée puisque les opérations doivent y être exécutées séquentiellement. Le plus souvent, plusieurs ordonnancements sont possibles, comme on l'a vu au chapitre précédent ; on choisira celui qui donne les meilleures performances temps réel.

De même, chaque partie "dépendances" placée sur un média de communication doit de plus être ordonnée, c'est-à-dire multiplexée sur le média. Ce multiplexage est obtenu au moyen d'opérateurs

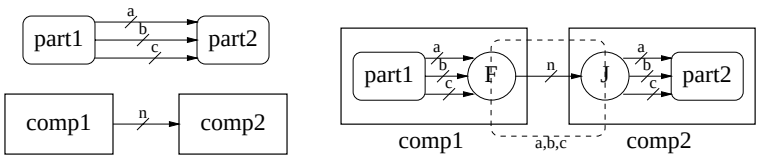

FIG. 9 – *Distribution des opérations et multiplexage des communications*

F et J par homogénéisation et factorisation des dépendances. Ces transformations du graphe de l'algorithme sont schématisées figure 9. Dans le domaine des processeurs, les unités de communication inter-processeur comme les liens de Transputer, de TMS320C40, d'ADSP21060 ou même les UART, réalisent à la fois un F et un J pour supporter des communications bidirectionnelles, soit simultanées ("full-duplex"), soit avec changement de direction programmable dynamiquement ("half-duplex") comme dans le cas de l'ADSP21060. Ces unités assurent de manière autonome le multiplexage de zones de mémoire contiguës (ou au mieux avec des adresses à intervalle constant ou brassées en "bit-reverse"), mais requièrent l'intervention du séquenceur d'instruction du processeur (interruption) pour synchroniser les communications avec les opérations de calcul et pour séquencer les étapes de communication entre les synchronisations.

3.2 Défactorisation et parallélisme

On a vu au chapitre 2.1 que la factorisation d'un graphe de dépendances ne change pas son parallélisme potentiel, donc il est de même pour la transformation inverse de "défactorisation". Si l'implantation directe d'un algorithme factorisé sur une architecture, en prenant en compte les contraintes technologiques, ne satisfait pas les contraintes temps réel de latence et/ou de cadence, il faut "défactoriser" le graphe de l'algorithme pour obtenir une implantation plus parallèle procurant de meilleures performances temps réel au prix de ressources supplémentaires (surface de circuit ou nombre de processeurs).

Un algorithme factorisé peut avoir de très nombreuses défactorisations partielles : pour n frontières de défactorisation, il existe 2^n variantes défactorisées de l'algorithme, et de plus chaque frontière peut n'être défactorisée que partiellement (par exemple, une factorisation de r répétitions d'un motif peut se décomposer en f factorisations de r/f répétitions du motif).

Certaines défactorisations ne procurent pas forcément plus de parallélisme. Par exemple, le graphe de la figure 5 ne comporte aucun parallélisme car ses dépendances établissent un ordre total sur toutes les opérations C. Dans ce cas, il faut décomposer l'opération C, comme on l'a fait pour l'exemple du produit matrice-vecteur, pour faire apparaître du parallélisme. Dans cet exemple, la première décomposition permet une exécution parallèle des produits scalaires, mais la seconde décomposition n'apporte de parallélisme supplémentaire qu'entre les produits, les sommes sont toutes interdépendantes. La présence d'un I dans une frontière de factorisation indique un parallélisme potentiel réduit par les dépendances inter-motif.

Un parallélisme supplémentaire "pipeline", améliorant la cadence, peut être obtenu par déplacement des frontières de factorisation. La figure 10 montre un exemple de ce type de transformation (connue sous le nom de "retiming"), qui consiste au niveau implantation à insérer des mémoires ($\$_2$ sur la figure) et/ou à en déplacer ($\$_1$ sur la figure). Si l'on n'insère pas des opérations supplémentaires (en pointillés dans la figure), la périodicité du motif répétitif est rompue au début de la répétition (et à sa fin pour une répétition finie, ce qui augmente d'autant le nombre de répétitions). Pour que cette insertion ne modifie pas la sémantique opératoire du graphe, il faut que ces insertions

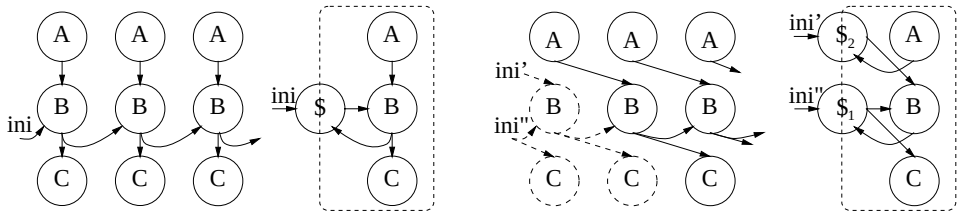

FIG. 10 – *Parallélisation "pipeline" par "retiming"*

aient un effet "neutre". Dans le cas d'un graphe flot de données, comme on ne se préoccupe pas de la fin de la répétition (qui est infinie), il suffit de fournir aux $ insérés ou déplacés des valeurs initiales telles que les valeurs produites par les opérations insérées soient les mêmes que les valeurs initiales fournies avant transformation aux $ et aux actionneurs. Dans le cas d'une répétition finie, il faut augmenter la période de répétition (sauf dans certains cas optimisables que nous n'avons pas la place de décrire ici), déphaser les compteurs périodiques des I déplacés et dévalider le multiplexeur d'initialisation des I insérés.

3.3 Caractérisation et optimisation

Parmi toutes les transformations possibles par défactorisation, on éliminera celles qui ne respectent pas les contraintes temps réel, et on choisira celle qui minimise les ressources nécessaires à l'implantation. Pour ne pas être fastidieux, ce choix doit être le plus automatisé possible, et surtout la comparaison des caractéristiques (latence, cadence, quantité de ressources) des différentes implantations doit se faire sans que l'utilisateur n'ait à les réaliser. Pour cela, il faut des heuristiques d'optimisation basées sur des modèles de caractérisation de performances.

Les problèmes d'allocation de ressources sont connus pour être NP-complets, c'est-à-dire qu'on ne connait pas d'autre moyen, pour trouver une solution optimale, que l'exploration globale de l'espace des solutions, ce qui reste inaccessible à l'échelle humaine pour des cas de figure réalistes et justifie l'utilisation d'heuristiques qui n'explorent qu'une petite partie de l'espace des solutions, et choisissent leur chemin exploratoire en optimisant une fonction de coût locale. Nous avons décrit dans [9] une heuristique d'optimisation dans le cas d'architectures composées de processeurs.

Comme les modèles d'algorithme et d'architecture sont compositionnels (composition d'opérations par un graphe de dépendances, composition d'opérateurs par un graphe d'interconnexions), les modèles de caractérisation de performances doivent permettre de déduire les caractéristiques de l'implantation par composition des caractéristiques de ses composantes, obtenues par mesure lorsqu'elles ne peuvent être obtenues par calcul. Nous avons décrit dans [10][11] un modèle de caractérisation des performances temps réel dans le cas d'architectures composées de processeurs, basé sur des calculs de longueur de chemin dans les graphes étiquetés par les caractéristiques temps réel des opérations de calcul et de communication.

4 Conclusions et perspectives

Nous avons montré comment modéliser algorithmes et architectures, de circuits aussi bien que de processeurs, dans un cadre unifié de graphes factorisés de dépendances de données entre opérations, côté algorithmes, et d'interconnexions entre opérateurs, côté architectures. Grâce à cette unification, le passage d'un algorithme à l'architecture optimisée qui le réalise, peut se faire de ma-

nière systématique donc automatisable. Nous avons réalisé le logiciel SynDEx pour supporter cette automatisation, qui s'est limitée jusqu'à présent aux cas d'architectures composées uniquement de processeurs, et aux optimisations par distribution et ordonnancement.

Pour étendre les optimisations aux cas d'implantations sur circuits, nous avons donné ici les principes de base d'autres formes d'optimisations par "transformations de factorisations", qui sont également utilisables dans le cas des processeurs. Nous envisageons de développer de nouvelles heuristiques pour automatiser ces autres formes d'optimisation, dans un premier temps séparément pour des architectures composées uniquement soit de circuits, soit de processeurs. À plus long terme, on peut espérer que cette approche permettra d'optimiser la conception conjointe logiciel-matériel ("co-design").

Références

[1] A. Benveniste, G. Berry. *The synchronous approach to reactive and real-time systems.* Proc. IEEE, 79(9):1270–1282, September 1991.

[2] Yves Sorel. *Real-time embedded image processing applications using the A^3 methodology.* Proc. IEEE International Conference on Image Processing, Lausanne, September 1996.

[3] A.M. Turing. *On computable numbers, with an application to the* Entscheindungs *problem.* Proc. London Math. Soc., 1936.

[4] M. Gondran, M. Minoux. *Graphes et algorithmes.* ed. Eyrolles 1979.

[5] R. Airiau, J.M. Berge, V. Olive, J. Rouillard. *VHDL, du langage à la modélisation.* Presses Polytechnique Romande, diff. Lavoisier, 1990.

[6] C. Lavarenne, C. Milan, M. Paindavoine, G. Richard, Y. Sorel. *Implantation d'algorithmes de traitement d'images sur une architecture multi-DSP avec l'environnement d'aide à l'implantation SynDEx.* Actes du Quatorzième Colloque GRETSI, Juan-Les-Pins, Septembre 1993.

[7] S.Y. Kung. *VLSI Array Processors.* Prentice Hall Information and System Sciences Series, Englewood Cliffs NJ, 1988. ISBN 0-13-942749-X.

[8] P. Leguernic, T. Gautier, M. Leborgne, C. Lemaire. *Programming real-time applications with SIGNAL.* Proc. IEEE, 79(9):1321–1336, September 1991.

[9] Y. Sorel. *Massively Parallel Systems with Real Time Constraints. The "Algorithm Architecture Adequation" Methodology.* Proc. Massively Parallel Computing Systems, the Challenges of General-Purpose and Special-Purpose Computing Conference, Ischia Italy, May 1994.

[10] C. Lavarenne, Y. Sorel. *Specification, Performance Optimization and Executive Generation for Real-Time Embedded Multiprocessor Applications with SynDEx.* Proc. Real-Time Embedded Processing for Space Applications, CNES International Symposium, 1992.

[11] F. Ennesser, C. Lavarenne, Y. Sorel. *Méthode chronométrique pour l'optimisation du temps de réponse des exécutifs SynDEx.* Rapports de Recherche INRIA no1769, 1992.

Présentation des auteurs

Yves Sorel : Directeur de recherche à l'INRIA (Institut National de Recherche en Informatique et Automatique). Ses recherches concernent la formalisation et l'optimisation d'implantations d'algorithmes de commande, de traitement du signal et d'images, s'exécutant en temps réel sur des architectures multi-composants. Il est responsable du groupe de travail "Adéquation Algorithme Architecture" du GDR/PRC ISIS.

Christophe Lavarenne : Ingénieur Arts et Métiers en 1979, ingénieur conseil dans la conception de systèmes temps réel embarqués et d'outils interactifs de développement croisé. Il collabore, en tant qu'ingénieur expert à l'INRIA, aux recherches sur la méthodologie Adéquation Algorithme Architecture et au développement du logiciel SynDEx qui la supporte.