Modèle d'exécutif distribué temps réel pour SynDEx

Thierry Grandpierre, Christophe Lavarenne, Yves Sorel

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Modèle d'exécutif distribué temps réel pour SynDEx

Thierry Grandpierre, Christophe Lavarenne, Yves Sorel

N° 3476

Août 1998

THÈME 4

ISSN 0249-6399

INRIA
ROCQUENCOURT

Modèle d'exécutif distribué temps réel pour SynDEx

Thierry Grandpierre, Christophe Lavarenne, Yves Sorel

Thème 4 — Simulation et optimisation
de systèmes complexes
Projet sosso

Rapport de recherche n° 3476 — Août 1998 — 71 pages

Résumé :

Ce document s'adresse aux concepteurs d'applications distribuées temps réel embarquées, qui désirent optimiser l'implantation de leurs algorithmes de commande et de traitement du signal et des images sur des architectures multiprocesseurs. Il s'adresse ensuite plus particulièrement aux utilisateurs du logiciel SynDEx v4 de CAO niveau système, qui supporte la méthodologie "Adéquation Algorithme Architecture", développée pour améliorer la productivité de ces concepteurs.

Le but de ce document est d'abord de permettre au public visé de comprendre les tenants et aboutissants de la méthodologie et de ses modèles, et plus particulièrement les exécutifs distribués générés par le logiciel SynDEx, optimisés pour le temps réel et pour les architectures embarquées multiprocesseur. Son but est ensuite de permettre à un public averti de porter le jeu de macros qui constituent le "noyau générique d'exécutif SynDEx v4", pour obtenir un générateur d'exécutif spécifique à un nouveau type de processeur.

Mots-clé : exécutif, distribué, multiprocesseur, temps réel, embarqué, SynDEx, méthodologie Adéquation Algorithme Architecture

(Abstract: pto)

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : 01 39 63 55 11 - International : +33 1 39 63 55 11
Télécopie : (33) 01 39 63 53 30 - International : +33 1 39 63 53 30

Distributed Real Time Executive Model for SynDEx

Abstract:

This document is addressed to designers of distributed real time embedded applications in the domain of signal and image processing, and control, who wish to optimize the implementation of their algorithms on multiprocessor architectures. It is addressed more particularly to the users of the system-level CAD software SynDEx v4, which supports the "Algorithm Architecture Adequation" methodology developed to improve the designer's productivity.

The purpose of this document is first to allow the targeted public to understand the issues and models of the methodology, and more particularly the distributed executives generated by the SynDEx software, which are optimized for real time and for embedded multiprocessor architectures. Its purpose is moreover to enable experienced users to port the set of macros which constitute the "SynDEx v4 generic executive kernel", in order to obtain an executive generator specific to a new processor type.

Key-words: executive, distributed, multiprocessor, real time, embedded, SynDEx, Algorithm Architecture Adequation methodology

Table des matières

Chapitre 1

Introduction

Ce premier chapitre d'introduction présente le contexte réactif, temps réel, embarqué, des applications et précise les domaines applicatifs qui nécessitent l'utilisation d'architectures distribuées, puis présente une démarche systématique pour résoudre les problèmes que pose l'implantation distribuée d'un algorithme applicatif sur de telles architectures, surtout lorsque l'on cherche à obtenir une implantation optimisée. À la fin de ce chapitre d'introduction, l'accent est mis sur l'importance des synchronisations dans les architectures distribuées pour assurer la correction de l'ordonnancement des opérations.

Le chapitre 2 présente la méthodologie "Adéquation Algorithme Architecture" en reprenant les points abordés pendant l'introduction, formalisés en termes de modèles d'algorithme, d'architecture et d'implantation, sur lesquels sont basés des heuristiques d'adéquation permettant d'obtenir des implantations optimisées pour lesquelles on peut générer automatiquement des exécutifs distribués temps réel. Ce chapitre méthodologique se termine sur une brève présentation du logiciel SynDEx.

Le chapitre 3 est consacré au modèle d'exécutif correspondant au modèle d'implantation dans le cas d'architectures multiprocesseurs. Ce chapitre décrit les principes de fonctionnement et la structure du macrocode généré pour charger les mémoires programme des processeurs, pour allouer statiquement la mémoire de données, pour séquencer et synchroniser sur chaque processeur d'une part les macro-opérations de calcul, et d'autre part les macro-opérations de transfert pour chaque média de communication interprocesseur. Toute la fin de ce chapitre est consacrée à l'implantation des dépendances de données, surtout dans le cas interprocesseur où elles donnent lieu à des communications et synchronisations par l'intermédiaire de différents types de médias.

Le chapitre 4 est une spécification de la syntaxe et de la sémantique des macros du jeu de macros constituant le "noyau générique" des exécutifs générés par le logiciel SynDEx dans sa version 4. Des exemples génériques de codage et d'utilisation des macros, dans un pseudo-assembleur très fortement inspiré du langage C, illustrent les spécifications pour faciliter la compréhension des noyaux d'exécutifs déjà développés et le travail du programmeur d'un nouveau noyau générique spécifique à un nouveau type de processeur.

Enfin, le chapitre 5 décrit la démarche habituellement suivie pour tester et mettre au point progressivement les macros d'un nouveau noyau générique d'exécutif. Cette démarche se décompose en une série de tests permettant chacun de valider quelques macros supplémentaires, ou des contextes supplémentaires d'utilisation de ces macros. Chaque test comprend une brève introduction, le source du test et souvent le résultat attendu.

1.1 Systèmes réactifs temps réel embarqués

On s'intéresse dans ce document à la programmation de systèmes informatiques pour des applications de commande et de traitement du signal et des images, soumises à des contraintes temps réel et d'embarquabilité. Dans ces applications, le système programmé commande son environnement en produisant, par l'intermédiaire d'actionneurs, une commande qu'il calcule à partir de son état interne et de l'état de l'environnement, acquis par l'intermédiaire de capteurs.

Les systèmes informatiques étant numériques, les signaux d'entrée acquis par les capteurs, ainsi que ceux de sortie produits par les actionneurs, sont discrétisés (échantillonnage-blocage-quantification), aussi bien dans l'espace des valeurs que dans le temps. La précision de la commande dépend de la résolution de cette discrétisation. Une analyse mathématique utilisant la théorie de la commande permet de déterminer d'une part une borne supérieure sur le délai qui s'écoule entre deux échantillons (cadence), et d'autre part une borne supérieure sur la durée du calcul (latence) entre une détection de variation d'état de l'environnement (stimulus) et la variation

induite de la commande (réaction). C'est en ce sens qu'on parle de *systèmes réactifs* [1] : la commande est calculée en réaction à chaque stimulus[1]. En plus de ces contraintes temps réel, l'application est soumise à des contraintes technologiques d'embarquabilité et de coût, qui incitent à minimiser les ressources matérielles (architecture) nécessaires à sa réalisation (l'architecture peut être composée de plusieurs processeurs, et de circuits spécialisés, pour satisfaire les contraintes temps réel).

C'est ce problème d'optimisation (minimisation de ressources matérielles) sous contraintes (temps réel et technologiques) qui fait l'objet de nos recherches.

L'algorithme de commande lui-même, la nature et la résolution des capteurs et des actionneurs, les contraintes temps réel, la nature des composants programmables (processeurs) et les contraintes d'embarquabilité et de coût, sont supposés ici avoir été déterminés au préalable. On ne s'intéresse ici qu'à l'implantation, à optimiser sous contraintes temps réel, de l'algorithme de commande sur l'architecture, en particulier lorsque celle-ci est composée de plusieurs processeurs.

1.2 Architectures distribuées embarquées

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 et/ou d'entrée-sortie 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 *multicomposants*, 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 satisfaire les besoins en puissance de calcul. Dans le domaine de l'automobile on aura rapidement 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 la quantité de câblage, c'est aussi le cas 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, et avec des médias de communication interprocesseur dont les débits doivent être en rapport avec les contraintes temps réel et avec le volume des informations à communiquer entre processeurs.

De plus, les contraintes temps réel et d'embarquabilité peuvent être tellement fortes que les processeurs disponibles sur le marché ne suffisent pas toujours. Cela conduit à utiliser, en complément des processeurs, des circuits intégrés spécialisés (ASIC[2] figés ou FPGA[3] configurables) qui réalisent généralement des fonctionnalités dites "de bas niveau", intégrables dans des circuits parce qu'elles ne nécessitent qu'un nombre très restreint de types différents d'opérations, utilisées de manière intensive et régulière. Les fonctionnalités qui nécessitent, 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.

Dans ce cadre, la conception d'une application consiste à choisir la composition de l'architecture puis à y implanter l'algorithme de commande, en cherchant à minimiser les ressources de l'architecture tout en respectant les contraintes temps réel.

1. Si la loi de commande doit varier non seulement en fonction des entrées mais aussi en fonction du temps, comme par exemple pour un filtre ou une transformée temps-fréquence, les stimuli doivent être ceux d'une horloge périodique conditionnant l'échantillonnage des capteurs et le rafraîchissement des actionneurs ; sinon les réactions du système peuvent être apériodiques, comme par exemple pour un système "auto-cadencé", où la durée d'une réaction ne dépend que des durées d'exécution des calculs de la commande, qui peuvent varier d'une réaction à l'autre en fonction des données.

2. ASIC : Application Specific Integrated Circuit

3. FPGA : Field Programmable Gate Array

1.3 Implantation distribuée optimisée

Dans ce document, comme nous nous intéressons uniquement aux processeurs, plutôt qu'aux circuits intégrés spécialisés, l'implantation de l'algorithme sur l'architecture consiste donc à traduire (coder) l'algorithme de commande en programmes à charger dans les mémoires des processeurs pour que ceux-ci les exécutent.

1.3.1 Parallélisation

Pour une architecture monoprocesseur, l'algorithme serait traduit en un seul programme, c'est-à-dire codé en un ensemble d'instructions exécutées séquentiellement par le séquenceur d'instructions du processeur. Pour une architecture multiprocesseur, composée de plusieurs séquenceurs d'instructions opérant en parallèle, ainsi que de médias de communication leur permettant d'échanger des données, il faut partitionner l'ensemble des instructions en fonction du nombre de séquenceurs d'instructions, allouer un séquenceur d'instructions à chacun des sous-ensembles d'instructions et enfin ajouter des instructions de synchronisation et de transfert de données, et leur allouer des médias de communication, pour supporter les dépendances de données entre les instructions de l'algorithme qui sont exécutées par des séquenceurs d'instructions différents.

Un partitionnement simple, par découpage linéaire du programme séquentiel en étapes successives exécutées chacune par un séquenceur d'instructions différent, permet rarement une exploitation efficace du parallélisme disponible de l'architecture, car les dépendances de données entre étapes sont alors souvent telles que les séquenceurs d'instructions passent une partie importante de leur temps à exécuter les instructions de synchronisation ajoutées pour supporter les dépendances de données entre étapes. Pour permettre une exploitation plus efficace du *parallélisme disponible* de l'architecture, il faut étendre l'*ordre total*, d'exécution des instructions du programme séquentiel monoprocesseur, à un *ordre partiel*, extrait par une analyse de dépendances de données entre instructions [2], exhibant le *parallélisme potentiel* de l'algorithme, et permettant une distribution (partitionnement ou "allocation spatiale") et un réordonnancement ("allocation temporelle", limitée au respect de l'ordre partiel) individuel des instructions.

La parallélisation est donc un problème d'allocation de ressources, que l'on désire réaliser de manière *efficace*, où les ressources sont les séquenceurs d'instructions et les médias de communication inter-séquenceurs, et où les tâches à allouer à ces ressources sont les instructions de l'algorithme ainsi que celles de synchronisation et de communication.

1.3.2 Optimisation

Pour un codage monoprocesseur d'un algorithme donné, on peut choisir n'importe quelle architecture cible multiprocesseur, et pour chaque architecture choisie il existe un grand nombre d'implantations possibles (c'est-à-dire de distributions et d'ordonnancements des instructions, qui respectent l'ordre partiel). Parmi toutes ces implantations possibles, seules sont elligibles celles dont les performances temps réel (calculées à partir des durées connues d'exécution des instructions et des transferts de données interprocesseurs) respectent les contraintes temps réel. Parmi les implantations elligibles, pour satisfaire les contraintes technologiques d'embarquabilité et de coût, il faut choisir celle qui minimise les ressources matérielles (nombre de processeurs, de médias de communication interprocesseur, et de cellules mémoire). Dans ce chapitre d'introduction, on ne cherchera pas à formaliser plus en détails ce problème d'optimisation (minimisation de ressources sous contraintes temps-réel), mais plutôt à en explorer les tenants et les aboutissants.

Le plus difficile n'est pas de comparer les coûts de deux architectures (il suffit d'en sommer les coûts des composants), ni même de vérifier si une implantation respecte les contraintes temps-réel (ce qui nécessite un modèle prédictif des performances temps-réel de n'importe quelle implantation possible), c'est d'éliminer rapidement les solutions inadéquates, afin d'aboutir dans un temps raisonnable au choix d'une bonne implantation. Or ce problème s'apparente aux problèmes d'allocation de ressources, reconnus NP-complets, et est en général de taille gigantesque (variant exponentiellement avec le nombre de processeurs et d'instructions de l'algorithme), ce qui justifie l'utilisation d'heuristiques pour le résoudre.

1.3.3 Minimisation de l'exécutif

Tout d'abord, pour minimiser les ressources, le bon sens (même s'il n'est pas commun) voudrait qu'on commence par minimiser leur gaspillage [4]. Pour cela, il faut :

- d'une part équilibrer la charge des ressources (mémoires, séquenceurs d'instructions et médias de communication), c'est-à-dire paralléliser au maximum calculs et communications afin de minimiser les durées d'inactivité (attentes) nécessaires aux synchronisations entre calculs et communications, autrement dit optimiser la distribution et l'ordonnancement des instructions de l'algorithme, ainsi que des communications interprocesseurs qui découlent de la distribution des instructions,
- d'autre part minimiser les ajouts d'instructions qui prennent les décisions d'allocation de ressources (mémoire, séquenceurs d'instructions et séquenceurs de communications) afin d'équilibrer leur charge, en particulier celles nécessaires à la gestion des communications (recopies mémoire, routage, synchronisation).

Pour que le gain apporté par l'optimisation de l'allocation des ressources (distribution et ordonnancement des calculs et des communications, dont découle la distribution des données dans les mémoires) ne soit pas pénalisé par le coût de l'optimisation elle-même, il faut que celle-ci soit faite avant l'exécution. Comme on dispose alors d'énormément plus de temps que pendant l'exécution, on peut faire des optimisations plus complexes, plus globales et donc probablement plus efficaces. Aux méthodes dites "dynamiques" d'allocation de ressources, qui nécessitent un exécutif représentant un volume de code et un temps d'exécution non négligeables pour prendre à l'exécution des décisions d'allocation, on préfèrera donc des méthodes plus "statiques" qui consistent à *synthétiser* un exécutif sur mesure, d'un surcoût bien moindre, à partir des décisions de distribution et d'ordonnancement prises avant l'exécution par l'heuristique d'optimisation.

1.3.4 Choix de la granularité

Ensuite, comme ce problème d'optimisation d'allocation de ressources a un espace des solutions à explorer variant exponentiellement avec le nombre de processeurs et d'instructions de l'algorithme, il faut en réduire la taille afin que la durée d'exécution de l'heuristique d'optimisation reste acceptable. Aussi, comme atomes ou "grains" indivisibles de distribution et d'ordonnancement, plutôt que de considérer des instructions individuelles, on considèrera des agrégats d'instructions préordonnancées (correspondant par exemple à des séquences d'instructions issues de la compilation séparée de sous-programmes FORTRAN ou de fonctions C) qu'on appellera par la suite indifféremment soit *macro-instructions*, soit plus généralement *opérations*, et des agrégats de cellules mémoire contiguës (correspondant par exemple à des tableaux ou à des structures C) qu'on appellera par la suite soit *macro-registres* soit plus généralement *dépendances (de données)*. Cette approche macroscopique présente de plus des avantages :

- **portabilité :** les opérations qui composent l'algorithme doivent être portables entre processeurs ayant des jeux d'instructions différents, non seulement parce que la durée de vie d'un algorithme dépasse presque toujours celle de sa première implantation, mais aussi parce que l'architecture peut être hétérogène (par exemple, le processeur qui gère l'interface avec l'utilisateur est souvent d'un type différent de ceux qui effectuent le gros des calculs),
- **modularité :** les algorithmiciens préfèrent concevoir leurs algorithmes en termes de structures de données (vecteurs, matrices, listes...) et d'opérations (filtres, transformées, opérations matricielles...) plus complexes que celles directement supportées par les jeux d'instructions,
- **surcoût par grain :** la portabilité et la modularité impliquent un surcoût d'interface pour chaque opération (lecture des opérandes et écriture des résultats en mémoire, et construction du contexte d'appel pour les opérations compilées séparément) ; de même, chaque communication implique un surcoût d'initialisation du contexte de la communication ; plus les grains sont petits, plus ils sont nombreux, et plus le surcoût total augmente,

4. Avec l'augmentation constante de la capacité mémoire et de la vitesse des processeurs des stations de travail, la pratique commune consiste plutôt à gaspiller ces ressources : ce gaspillage, au nom d'une prétendue économie sur les temps de développement, se traduit par une augmentation constante de la taille des logiciels et des systèmes d'exploitation, qui ne se traduit pas par une augmentation dans les mêmes proportions de leurs fonctionnalités, et encore moins de leurs performances, ou parfois même de leur maintenance.

– **caractérisation :** de plus en plus de processeurs utilisent des mémoires caches et des pipelines pour chercher, décoder et exécuter leurs instructions, et/ou pour lire leurs opérandes et stocker leurs résultats, ce qui induit des interactions, entre instructions successives, qui dépendent de détails architecturaux complexes et souvent non publiés ; la variation relative de durée d'exécution d'une séquence d'instructions est moindre que celle d'une instruction isolée, car ces interactions ont plus tendance à se compenser qu'à se cumuler.

Cependant, cette agrégation ne doit être ni excessive ni déséquilibrée: si quelques gros grains représentent à eux seuls la quasi-totalité du volume de calcul et/ou s'il y a trop peu de grains par rapport au nombre de séquenceurs d'instructions, l'heuristique a peu de choix pour équilibrer la charge des ressources; réciproquement, avec des grains plus petits, donc plus nombreux, l'heuristique a plus de choix pour équilibrer la charge des ressources, mais elle y met plus de temps, cumule plus d'erreurs, et le surcoût total augmente (car il y a un surcoût et une incertitude de durée d'exécution par grain, peu dépendants de la taille du grain). La pratique montre que le choix de la granularité a un impact important dans tous les problèmes d'optimisation d'allocation de ressources. La méthodologie "Adéquation Algorithme-Architecture" présentée au chapitre 2 apporte une aide au choix de la granularité.

Le choix de la granularité au niveau de l'architecture est moins délicat : il doit mettre en évidence le parallélisme disponible entre séquenceurs programmables. Ainsi, bien qu'un DSP[5] soit capable d'effectuer simultanément une multiplication, une addition, trois accès mémoire (lecture de deux données et d'une instruction), trois modifications de pointeurs mémoire (deux de données et un de programme), une décrémentation de compteur d'itérations et un test de fin d'itération, il n'est pas nécessaire de décrire le parallélisme entre les unités opératoires qui effectuent ces opérations, car ces unités opératoires ne sont pas séquençables indépendamment, elles sont commandées ensemble à chaque cycle instruction par un unique décodeur d'instructions, donc on les représentera par un seul "grain" séquentiel d'architecture les regroupant avec le séquenceur d'instructions. Par contre, un automate de communication interprocesseur est capable de séquencer, en parallèle avec le séquenceur d'instruction d'un processeur, des transferts entre la mémoire qu'il partage avec le processeur et le média de communication qu'il partage avec un (des) autre(s) automate(s) de communication, donc on le représentera par un grain séquentiel d'architecture différent de celui du processeur pour le compte duquel il effectue des communications. Ainsi, on décrira l'architecture en termes de séquenceurs d'opérations de calcul ou de transfert, que nous appellerons par la suite *opérateurs* (ressources séquentielles), et de *médias* (ressources partagées entre opérateurs) qui leur permettent de communiquer et de se synchroniser pour assurer la précédence entre production (écriture) et la consommation (lecture) des données communiquées.

1.3.5 Caractérisation et prédiction de performances

L'optimisation de l'allocation des ressources nécessite la comparaison des performances des différentes implantations possibles. Réaliser chaque implantation possible pour en mesurer les performances est inabordable : le nombre de solutions possibles est gigantesque, il se peut que l'architecture cible ne soit pas encore disponible au moment où l'on veut faire la comparaison, la durée nécessaire à la compilation et à l'exécution d'une implantation pour en mesurer les performances peut être longue, et surtout il faut choisir des conditions d'exécution qui couvrent tous les cas de figure, y compris le pire du point de vue des performances, afin de vérifier que ces dernières respectent les contraintes temps-réel. La comparaison par la mesure étant inabordable, il est nécessaire de construire un modèle prédictif des performances "pire cas" des implantations, qu'utilisera la fonction de coût de l'heuristique d'optimisation. Comme l'algorithme et l'architecture sont composés (de "grains" mis en relation par interconnexion), le modèle prédictif de performances doit être compositionnel, et pour permettre cette composition, il faut que l'algorithme et l'architecture soient spécifiés en termes de types cohérents de grains d'allocation.

La cohérence de types de grains d'allocation requiert d'une part que l'algorithme soit composé de grains *opérations* exécutables (pour chacun desquels il doit y avoir au moins un *opérateur*, grain de l'architecture, capable de l'exécuter seul, et il doit exister un code exécutable pour chaque type différent d'opérateur capable de l'exécuter) et que l'architecture soit composée de grains *opérateurs* (processeur ou circuit spécialisé) interconnectés, chacun capable de séquencer des opérations et des communications avec les opérateurs auxquels il est connecté (par l'intermédiaire de *médias* de communication). Cette cohérence permet une description de l'exécutif, et donc sa synthèse et l'évaluation de ses performances, en termes de composition parallèle (distribution) de séquences (ordonnancement) d'opérations de calcul et/ou de communication, synchronisées au niveau des accès aux tampons mémoire qu'elles partagent.

5. DSP : Digital Signal Processor, processeur de traitement du signal.

Le modèle de prédiction de performances doit être compositionnel et se baser sur les caractéristiques de chaque grain d'allocation : chaque opérateur doit être caractérisé (par mesure [3, 4] ou à défaut par estimation), en termes de durée d'exécution et de capacité mémoire, pour chaque opération qu'il est capable d'exécuter et/ou pour toute communication qu'il doit pouvoir supporter. La composition de ces caractéristiques est faite en fonction de la distribution et de l'ordonnancement des opérations et/ou des communications. La distribution implique une composition spatiale, parallèle, prenant en compte les arbitrages d'accès aux ressources partagées, et l'ordonnancement implique une composition temporelle, séquentielle, prenant en compte les synchronisations nécessaires aux communications.

1.3.6 Génération d'exécutifs

Tout modèle étant entâché d'erreurs dues aux nécessaires approximations simplificatrices, les dates de début et de fin d'exécution des opérations et des communications, calculées par le modèle de prédiction de performances lors de l'optimisation à partir des durées d'exécution, peuvent être suffisament différentes lors de l'exécution pour que leur ordre relatif (entre séquences parallèles) change. Pour garantir à l'exécution l'ordre relatif entre communications et opérations qui partagent les tampons mémoire des données communiquées, on ne peut donc se contenter des prédictions de dates d'exécution faites pendant l'optimisation, il faut imposer cet ordre relatif par des synchronisations explicites dans l'exécutif. En bref, la distribution et l'ordonnancement des opérations et des communications sont optimisés avant l'exécution, en fonction des durées d'exécution données, mais ils sont imposés à l'exécution indépendament des durées effectives d'exécution.

Toutes ces observations nous ont logiquement conduits à formaliser et modéliser l'algorithme et l'architecture en termes de graphes, et l'implantation et son optimisation en termes de transformations de graphes : c'est l'objet du chapitre 2. En plus des optimisations, ce formalisme permet aussi une génération automatique d'exécutifs distribués, par composition de macro-instructions d'un jeu formant un *noyau générique d'exécutif*, traduites par le macro-processeur "m4" (standard sous Unix, version GNU) au moyen d'un jeu correspondant de macros, extensible et facilement portable, aussi bien en assembleur qu'en langage de haut niveau : c'est l'objet du chapitre 3. Mais avant de passer à ces chapitres, il est nécessaire, pour la bonne compréhension des modèles d'algorithme, d'architecture et d'exécutif, de décrire plus en détail les caractéristiques structurales communes aux architectures parallèles, multicomposant, sur lesquelles doivent être implantées les noyaux génériques d'exécutif.

1.4 Synchronisation dans les architectures distribuées

L'indépendance de fonctionnement entre composants, fournie par leur parallélisme, est limitée par leur synchronisation, nécessaire à la réalisation d'une tâche commune.

1.4.1 Synchronisation centralisée

Par exemple, un processeur de traitement du signal, vu au niveau transfert de registre (RTL "Register Transfer Level"), est composé en général d'un multiplieur, d'un additionneur, de deux incrémenteurs d'adresses de données, d'un incrémenteur d'adresse programme et d'un décodeur d'instructions, qui opèrent en parallèle, mais qui sont fortement synchronisés par le séquenceur d'instructions : chacun effectue une opération (transfert entre registres à travers un circuit combinatoire) à chaque cycle instruction, l'ordre relatif de ces opérations est totalement déterminé, à la compilation, par l'ordre des instructions. Autre exemple, un processeur vectoriel SIMD [6] est capable d'effectuer simultanément sur plusieurs données une même opération, mais ces unités opératoires ne sont pas séquençables indépendamment, elles sont commandées ensemble à chaque cycle instruction par un unique séquenceur d'instructions.

Aussi, chaque fois qu'il faudra les distinguer, on qualifiera de *macro-opérations* les opérations de l'algorithme (composée chacune d'une séquence d'instructions), et de *micro-opérations* les opérations contrôlées par un séquenceur d'instructions (où l'exécution d'une instruction consiste en l'exécution de plusieurs micro-opérations, en parallèle et/ou en micro-séquence). De même, on qualifiera de *macro-registres* les zones mémoire stockant les opérandes ou les résultats des macro-opérations.

Les mêmes observations s'appliquent aux opérateurs séquenceurs de transferts mémoire (DMA "Direct Memory Access"), souvent présents dans les architectures pour supporter, en parallèle avec le séquencement des instructions, le séquencement des transferts de données à travers des média de communication inter-processeur. Dans ce cas, on qualifiera de *macro-transferts* ou de *macro-opérations de communication* le transfert d'un

6. SIMD : Single Instruction stream for Multiple Data streams.

macro-registre, et de *micro-transfert* ou *micro-opération de communication* le transfert d'une cellule mémoire ou chaque cycle d'accès à un média de communication.

1.4.2 Synchronisation distribuée

La synchronisation n'est pas aussi forte entre opérateurs (séquenceurs d'instructions et/ou de communications), car chacun peut séquencer ses opérations indépendamment des autres, sauf dans les deux cas suivants :

1. Lorsque deux opérateurs requièrent simultanément, pour leur micro-opération en cours, un accès à une même ressource (partagée), comme par exemple un bus mémoire, les deux accès doivent être séquentialisés, donc l'un des deux opérateurs doit, avant de commencer son accès, attendre que l'autre opérateur ait terminé le sien, ce qui rallonge d'autant la durée d'exécution de la micro-opération mise en attente ; ces interférences entre micro-opérations, et donc entre macro-opérations, dues aux arbitrages d'accès aux ressources partagées entre opérateurs, doivent être prises en compte dans le modèle prédictif de performances.

2. Lorsqu'une macro-opération consomme en donnée le résultat produit par une autre macro-opération exécutée par un autre opérateur, il faut que ce dernier termine l'exécution de la macro-opération productrice avant que l'autre opérateur ne commence l'exécution de la macro-opération consommatrice (par la suite, on qualifiera cette précédence de "tampon plein"), et de plus, comme les algorithmes réactifs sont par nature répétitifs, il faut que la macro-opération consommatrice soit terminée avant que la macro-opération productrice soit à nouveau exécutée lors de l'itération suivante de la séquence répétitive (par la suite, on qualifiera cette précédence de "tampon vide"), tout ceci afin que les données ne soient pas modifiées pendant leur utilisation.

Dans les deux cas, des opérations qui auraient pu être exécutées concurremment doivent être exécutées séquentiellement, mais leur ordre d'exécution est sans influence sur le résultat fonctionnel des opérations dans le premier cas, alors qu'il doit être imposé dans le second cas.

C'est la raison pour laquelle dans le premier cas il n'est pas nécessaire de spécifier les synchronisations, d'autant plus qu'elles doivent être faites au niveau de la micro-opération, "invisible" au niveau macroscopique (et même au niveau de l'instruction), et que leurs dates d'occurrence dépendent des durées relatives d'exécution des micro-opérations exécutées concurremment, raison sans doute pour laquelle elles sont généralement supportées par un arbitre matériel qui détermine, à l'exécution, l'ordre des micro-opérations.

Par contre dans le second cas, les synchronisations doivent être spécifiées au niveau macroscopique, insérées entre les macro-opérations (après la macro-opération productrice et avant la macro-opération consommatrice dans le cas d'une précédence tampon-plein, et réciproquement dans le cas d'une précédence tampon-vide). Comme il faut en général plusieurs instructions pour coder une synchronisation, on parlera par la suite de "macro-opérations de synchronisation".

Ce niveau "opérateur" de décomposition de l'architecture correspond à un grain adéquat pour le problème d'optimisation d'allocation de ressources : chaque opérateur séquence des macro-opérations de calcul et/ou de communication, et de synchronisation.

1.4.3 Synchronisation pour partage des séquenceurs d'instructions

En général les opérateurs de communication ont des capacités limitées de séquencement. Par exemple tous les DMA connus sont incapables de séquencement conditionnel : un séquenceur d'instructions est requis pour lire une donnée booléenne et pour effectuer un branchement conditionnel. Autre exemple, la plupart des DMA, s'ils sont capables de séquencer seuls la suite des micro-transferts, à des adresses mémoire régulièrement espacées, correspondant à un macro-transfert, sont incapabables de séquencer des macro-transferts : un séquenceur d'instructions est requis pour charger les registres du DMA qui paramètrent et initient le macro-transfert. Dernier exemple, une UART (Universal Asynchronous Receiver Transmitter), si elle est capable de séquencer seule les bits de "start", de donnée, et de "stop" d'un octet, est incapable de séquencer les octets d'un macro-transfert : un séquenceur d'instructions est requis non seulement pour transférer chaque octet entre le registre de sérialisation et la mémoire et pour incrémenter l'adresse mémoire et décrémenter le compte d'octets du macro-transfert, c'est-à-dire pour émuler un DMA, mais aussi pour charger les pseudo-registres (d'adresse et de compte) de ce DMA émulé.

Une séquence de macro-opérations de communication est donc supportée alternativement par un opérateur séquenceur de transferts et par un opérateur séquenceur d'instructions. Ce dernier doit donc partager son temps entre l'exécution des macro-opérations de calcul et le service des séquenceurs de transferts qui requièrent son

aide (il peut y en avoir plusieurs). Ce partage nécessite un arbitrage et une gestion simultanée du contexte (état d'avancement) de la séquence de calculs et des contextes des séquences de transferts. Comme l'expérience montre que les communications sont souvent critiques, il faut minimiser leur latence, et donc arbitrer l'allocation du séquenceur d'instructions en priorité au service des communications. Cependant, cette allocation prioritaire doit se limiter au strict nécessaire : pendant qu'un macro-transfert attend pour démarrer la fin d'une macro-opération de calcul, il faut que le séquenceur d'instructions soit alloué à la séquence de calculs pour avoir une opportunité d'exécuter la macro-opération de calcul dont la fin est attendue ; de même, pendant que l'opérateur de communication séquence ses micro-transferts, le séquenceur d'instruction doit être alloué à la séquence de calculs afin de tirer parti au mieux du parallélisme disponible entre opérateurs de calcul et de communication. Cette allocation ne doit donc avoir lieu que dès qu'un tampon mémoire est prêt à être transféré *et* dès que l'opérateur de communication est disponible pour le transfert (et juste pour la durée nécessaire à son relancement). En général ces deux événements ne sont pas simultanés et l'ordre de leurs occurences peut être quelconque, il faut donc les synchroniser : c'est à cela que servent les macro-opérations de synchronisation [7]. Ces deux événements sont générés par deux sources différentes : le premier par le séquenceur d'instructions et le second par le séquenceur de transferts. Les détails de leur génération et de leur synchronisation, nécessaires à une bonne compréhension de l'implantation des macro-opérations de synchronisation, sont décrits au chapitre 4.8.2.

7. Traditionnellement, le parallélisme et la synchronisation, entre calculs et communications, passent par le découpage des données en paquets recopiés dans des tampons mémoire alloués dynamiquement et organisés en files d'attente gérées sous interruption de fin de communication de paquet. Notre méthode est plus efficace, car elle évite les coûts d'allocation et de recopie en partageant, en exclusion mutuelle, les tampons mémoire de données, entre une séquence composée de macro-opérations de calcul et de synchronisation et une autre séquence composée de macro-opérations de communication et de synchronisation.

Chapitre 2

Méthodologie Adéquation Algorithme Architecture (AAA)

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 [5], 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 allouer les ressources matérielles de l'architecture aux opérations de l'algorithme, puis à compiler, charger, puis lancer l'exécution du programme correspondant sur le calculateur, avec dans le cas des processeurs le support d'un système d'exploitation ou d'un exécutif dont le surcoût ne doit pas être négligé.

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. Brièvement, car cela sera développé au chapitre 2.3.2, on va rechercher parmi toutes les implantations que l'on pourrait faire d'un algorithme sur une architecture donnée, une implantation particulière que l'on considérera comme optimisée en fonction d'un objectif que l'on s'est fixé.

2.1 Modèle d'algorithme

2.1.1 Modèle flot de contrôle et flot de données

De façon générale un algorithme peut être spécifié par un graphe ; on présente ci-dessous deux types de graphes : les graphes flot de contrôle et les graphes flot de données.

Dans la version organigramme d'un graphe flot de contrôle, les sommets du graphe sont des opérations qui consomment leurs données opérandes, et produisent leurs données résultats, dans des variables. Les arcs traduisent une relation d'ordre d'exécution "s'exécute avant" entre les opérations qu'ils relient. Dans la version "orientée automate", les sommets du graphe sont les états et les arcs définissent les transitions entre états, pendant lesquelles sont exécutées des opérations, qui elles aussi manipulent des variables. Dans les deux cas, un ordre total d'exécution à été imposé sur l'ensemble des opérations du graphe, ce qui correspond bien à la définition standard donnée plus haut d'un algorithme. Pour pouvoir exécuter des opérations en parallèle, il faut faire une analyse de dépendance des données communiquées entre opérations par l'intermédiaire des variables, afin de pouvoir décomposer l'algorithme en plusieurs graphes de contrôle (séquences d'opérations), composés en parallèle en établissant entre eux des communications au niveau des dépendances de données inter-séquences, comme dans le modèle CSP (Communicating Sequential Processes) de Hoare [6].

Un graphe flot de données est un hyper-graphe [7] orienté, où chaque sommet est une opération, et chaque arc est un transfert de données entre une opération productrice et une ou plusieurs (diffusion) opérations consommatrices. L'exécution de chaque opération et de chaque transfert de données est répétitive, d'où la notion de "flot". Chaque opération, à chacune de ses exécutions, consomme une donnée sur chacun de ses arcs d'entrée et les combine pour produire une donnée sur chacun de ses arcs de sortie. Une opération sans arc d'entrée (resp. de sortie) représente une interface d'entrée (capteur, resp. de sortie, actionneur) avec l'environnement physique. Lorsqu'une opération a besoin lors de sa n-ième exécution de consommer une donnée produite lors de la $(n-1)$-ième exécution d'une autre opération, il faut intercaler entre ces deux opérations une opération particulière appelée "retard" (le z^{-1} des traiteurs de signal), qui consomme une donnée sur son arc d'entrée **après** avoir produit sur son arc de sortie la donnée lue sur son arc d'entrée lors de son exécution précédente (une donnée initiale lors de sa première exécution). Sur chaque arc, chaque donnée doit être produite avant de pouvoir être consommée, donc les arcs traduisent une relation d'ordre d'exécution "s'exécute avant" entre les opérations, et en conséquence un graphe flot de données ne peut contenir de cycles que s'il y a au moins un retard dans chaque cycle. Ainsi un graphe flot de données n'impose qu'un ordre partiel sur l'exécution de ses opérations, et deux opérations qui ne sont pas en relation d'ordre peuvent être exécutées dans n'importe quel ordre, y compris en parallèle, si les ressources le permettent. De plus, la répétition implicite de l'exécution des opérations et des transferts de données autorise une autre forme de parallélisme "pipeline", où la n-ième donnée peut être produite sur un arc pendant que la $(n-1)$-ième est consommée, sur un autre processeur. Ces deux formes de parallélisme potentiel permettent de nombreuses implantations d'un même algorithme, qui consistent chacune à composer différemment des opérations en séquence et les séquences en parallèle, avec communications inter-séquences comme dans le cas précédent.

L'intérêt principal d'un graphe flot de données est la description explicite des dépendances de données, nécessaires à l'implantation parallèle de l'algorithme, alors que, dans le cas d'un graphe flot de contrôle, il faut extraire ces dépendances, implicitement décrites par le partage de variables entre opérations. Un autre intérêt du graphe flot de données est de localiser la mémoire d'état de l'algorithme dans les retards, contrairement au graphe flot de contrôle où la mémoire d'état n'est pas distinguée parmi les variables.

Il faut noter que pour les deux types de modèles, graphes flot de contrôle mis en parallèle, et graphes flot de données, on a étendu la notion initiale d'algorithme liée à un ordre total d'exécution sur les opérations, à un ordre partiel d'exécution. Dans la suite nous utiliserons systématiquement ce modèle étendu quand nous parlerons d'algorithme.

2.1.2 Prise en compte du temps, vérifications, simulations

Les langages Synchrones Esterel, Lustre, Signal, Statemate possèdent une sémantique tenant compte à la fois des aspects parallélisme et temporel [8, 9]. Alors qu'Esterel et Statemate sont des langages impératifs auxquels ont peut associer des graphes flot de contrôle, Lustre et Signal sont des langages déclaratifs auxquels ont peut associer des graphes flot de données.

Les langages Synchrones font l'hypothèse que les données produites par une opération apparaissent *simultanément* avec les données d'entrée qui ont déclenché l'opération, *c'est-à-dire sans attendre de nouvelles données d'entrée*. Cette notion de simultanéité est purement logique, elle se veut indépendante de toute implantation et

donc des durées physiques d'exécution des opérations, durées qui dépendent de l'implantation. Cette hypothèse permet de considérer que les calculs (sommets du graphe flot de données) et les transferts de données (arc du graphe flot de données) ont lieu de manière instantanée, leurs durées physiques ne sont pas considérées. Par transitivité appliquée à tous les sommets du graphe, les données produites par le graphe apparaissent simultanément avec les données y entrant. Ces dernières venant de l'environnement définissent des événements d'entrée ou stimuli. De même, les données produites pour l'environnement par le graphe définissent des événements de sortie ou réactions. Tout événement de sortie est associé à un événement d'entrée. Cela permet de définir un temps logique où seul compte l'ordre relatif des événements, indépendamment des durées physiques qui s'écoulent entre les événements. La notion de durée (logique) n'existe alors qu'au travers du comptage des événements.

Les compilateurs des langages Synchrones effectuent des vérifications sur la cohérence entre les événements produits en réaction aux événements qui les déclenchent. À ce niveau les vérifications ne portent que sur l'ordre des événements, la notion de durée physique liée à une horloge temps réel n'est pas prise en compte. Cela sera cependant fait plus tard lors de l'implantation et sera décrit au chapitre 2.3.2. Les compilateurs permettent de montrer par exemple que certains événements auront toujours lieu, ou bien se produiront après un certain nombre d'occurences d'un autre événement, ou bien que certains événement n'auront jamais lieu. Les raisonnements formels utilisés ici ne portent que sur des booléens, ils sont principalement basés sur des techniques de "Model Checking" utilisant le plus souvent des BDD (Binary Decision Diagram) [10]. Ces vérifications bien que limitées, éliminent un grand nombre d'erreurs logiques qui habituellement sont découvertes lors des tests en temps réel sur le prototype. Découvrir ces erreurs le plus tôt possible dans le processus de conception des applications temps réel embarquées est très important ; cela permet de diminuer la phase de tests temps réel très coûteuse que l'on estime parfois à 70% du temps de développement de ce type d'applications. On verra plus loin comment conserver ces propriétés lorsqu'on prendra en compte le temps physique au moment de l'implantation.

En plus des vérifications vues ci-dessus les compilateurs des langages Synchrones sont capables de générer un code exécutable séquentiel, généralement du C mais aussi du Fortran ou de l'Ada ou d'autres langages séquentiels. Ce code séquentiel est utilisé pour faire la simulation numérique et la vérification du comportement événementiel, en termes d'ordre sur les événements seulement, de l'algorithme ainsi spécifié.

2.2 Modèle d'architecture

2.2.1 Modèle multicomposant

Les modèles les plus classiquement utilisés pour spécifier des architectures parallèles ou distribuées sont les PRAM ("Parallel Random Access Machines") et les DRAM ("Distributed Random Access Machines") [11]. Le premier modèle représente un ensemble de processeurs communicant par mémoire partagée alors que le second correspond à un ensemble de processeurs communicant par mémoire distribuée avec passage de messages.

Afin d'exploiter au mieux les algorithmes d'optimisation utilisés lors de la recherche d'implantations optimisées, il est nécessaire d'utiliser des modèles d'architecture adaptés, qui sont des extensions de ceux présentés ci-dessus. En effet, ces derniers très généraux doivent être affinés quand on conçoit des systèmes embarqués dans lesquels il s'agit d'optimiser au maximum les ressources matérielles.

L'architecture d'un calculateur est généralement décrite de façon hiérarchique. On parle au plus haut niveau de baie, que l'on décrit comme un ensemble de cartes électroniques, elles-mêmes décrites en termes d'assemblage de composants programmables (processeurs) ou non programmables (circuits intégrés spécialisés), chaque composant peut à son tour se décrire en termes de composition d'automates. La notion d'automate est prise ici dans sa forme générique et correspond à une machine à états finie transformatrice, comportant des sorties. C'est en reliant les sorties de certains automates aux entrées d'autres automates qu'on les compose. On pourrait ainsi continuer à descendre dans le détail jusqu'aux transistors au risque d'aboutir à un modèle fort complexe. Le niveau consistant à choisir comme composant atomique non décomposable l'automate vu ci-dessus, permet une bonne précision dans le modèle tout en n'étant pas trop compliqué lorsqu'il s'agira de l'exploiter pour réaliser l'adéquation.

Une architecture multicomposant est donc un réseau de composants inter-connectés par des média de communication (liens, bus, mémoires partagées ...) dont le composant atomique est un automate. Elle peut se modéliser par un hyper-graphe non orienté, dont chaque sommet est un automate et chaque hyper-arc un média de communication. Un hyper-arc relie plusieurs sommets entre eux, contrairement à un arc simple qui ne relie que deux sommets. Il permet des communications bidirectionnelles.

Il y a deux types de sommets. Les opérateurs (ALU, FPU etc ...) qui séquencent des opérations et les transformateurs (DMA, convertisseur série/parallèle etc ...) qui séquencent des transferts de données entre les

mémoires de deux média de communication. Il existe des opérateurs dégénérés qui ne sont capables d'exécuter qu'un seul type d'opération, on verra plus loin au chapitre 2.3.1 qu'ils seront associés à des circuits intégrés spécialisés non programmables, ne réalisant qu'une seule fonction. Un média de communication comprend les fils conducteurs, une mémoire et un automate arbitre. Ce dernier réalise l'ordonnancement des accès aux média de communication (contrôle des ressources partagées) et la synchronisation des opérateurs et des transformateurs (mises en attente). La mémoire est de deux types, à accès aléatoire pour réaliser des communications asynchrones, ou à accès séquentiel (FIFO) pour réaliser des communications synchrones. L'ensemble des hyper-arcs définit sur les opérateurs et les transformateurs la relation "être connecté à".

Cette spécification est de type Macro-RTL, extension du modèle classique RTL (Register Transfert Level c'est-à-dire au niveau transfert de registres) [12]. Une macro-opération est une opération du graphe de l'algorithme (une séquence d'instructions) ; un macro-registre est une zone mémoire contiguë. Ce modèle encapsule les détails liés au jeu d'instructions, aux micro-programmes, au pipe-line, au cache, et lisse ainsi ces caractéristiques délicates à prendre en compte lors de l'optimisation. Il présente une complexité réduite adaptée aux algorithmes d'optimisation rapides tout en permettant des résultats d'optimisation relativement (mais suffisamment) précis.

2.2.2 Caractérisation d'architecture

Il s'agit de caractériser les composants et le réseau en fonction des contraintes temps réel et d'embarquabilité. Pour cela on associe à chaque opérateur et à chaque transformateur l'ensemble des opérations que chacun d'eux est capable de réaliser, et pour chaque opération sa durée, son occupation mémoire, la consommation associée à son exécution etc. Par exemple, l'opérateur unité centrale d'un processeur de traitement du signal est capable de réaliser, entre autres, une multiplication et une accumulation (instruction de base du processeur) en un cycle et une FFT (séquence d'instructions de base) en un certains nombre de cycles. De même, pour le DMA associé à un lien de communication d'un processeur de traitement du signal, on associera les opérations de transferts qu'il est capable de réaliser en fonction des types de données utilisés (un entier peut prendre moins de temps à être transféré qu'un réel, ou qu'un tableau d'entiers).

Les automates arbitres quant à eux jouent un rôle crucial, ils gèrent les accès aux ressources partagées (séquenceur, moteur de DMA etc...). Ils sont caractérisés par une matrice d'interférence qui décrit le ralentissement que subissent des opérateurs et/ou des transformateurs quand ils utilisent simultanément une même ressource. Les éléments de cette matrice servent à pondérer les valeurs brutes vues ci-dessus, associées aux opérateurs et aux transformateurs.

Cette caractérisation sera exploitée lors de l'optimisation comme on va le voir dans un chapitre suivant.

2.3 Modèle d'implantation

2.3.1 Distribution et ordonnancement

L'implantation d'un algorithme sur une architecture, consiste à réaliser, en tenant compte des contraintes, une distribution et un ordonnancement des opérations de l'algorithme sur l'architecture caractérisée comme indiqué dans le chapitre précédent. Il faut noter que ce qu'on appelle ici "distribution", est souvent appelé "placement" ou "répartition".

La distribution consiste tout d'abord à effectuer une partition du graphe de l'algorithme initial, en autant (ou moins) d'éléments de partition qu'il y a d'opérateurs dans le graphe de l'architecture. On verra plus loin comment ce graphe est obtenu s'il n'est pas imposé a priori. Il faut ensuite affecter chaque élément de partition, c'est-à-dire chaque sous-graphe correspondant du graphe de l'algorithme initial, à un opérateur du graphe de l'architecture. On ne peut affecter qu'un seul type d'opération à un opérateur dégénéré représentant un circuit intégré spécialisé non programmable. Puis à affecter les transferts de données du graphe de l'algorithme appartenant à des éléments de partition différents, à des routes. Ces dernières sont formées par une chaîne de transformateurs et de média de communication, pouvant se réduire à un seul média si on a une communication directe entre opérateurs. On obtient l'ensemble des routes d'un graphe d'architecture donné en calculant la fermeture transitive de la relation "être connecté à" définie au chapitre 2.2.1. Il peut bien sûr y avoir plusieurs routes parallèles, de longueurs (nombre d'éléments la constituant) différentes, reliant deux opérateurs.

L'ordonnancement consiste, pour chaque élément de partition, à linéariser (rendre total) l'ordre partiel correspondant au sous-graphe associé. Il se peut que celui-ci soit déjà un ordre total. Cette phase est nécessaire car l'opérateur auquel on affecte un sous-graphe du graphe de l'algorithme initial, est un automate séquentiel par définition, dont le rôle est d'exécuter séquentiellement des macro-opérations (dont chacune est une séquence d'instructions de base).

Étant donnés un graphe d'algorithme et un graphe d'architecture, on comprend aisément qu'il existe un nombre fini, mais qui peut être très grand, de distributions et d'ordonnancements possibles. En effet, on peut effectuer plusieurs partitions du graphe de l'algorithme, en fonction du nombre d'opérateurs, et pour chaque sous-graphe affecté à un opérateur il y a plusieurs linéarisations possibles de ce sous-graphe. La première chose à faire consiste à éliminer toutes les distributions et les ordonnancements qui ne conserveraient pas les propriétés montrées lors de la spécification avec les langages synchrones. Pour cela, il faut préserver la fermeture transitive du graphe de l'algorithme. L'ordre partiel associé au graphe transformé du graphe de l'algorithme après la distribution et l'ordonnancement doit être compatible avec l'ordre partiel du graphe de l'algorithme initial. Il s'agit maintenant d'expliquer comment parmi ces distributions et ordonnancements valides on va élire une distribution et un ordonnancement particuliers afin d'obtenir une implantation optimisée. Ce choix se fait en fonction des contraintes temps réel et d'embarquabilité.

2.3.2 Contraintes et optimisation

Il s'agit tout d'abord de respecter les contraintes temps réel qui sont de deux types : la latence et la cadence. Contrairement au chapitre 2.1.2 on prend maintenant en compte les durées physiques, que l'on doit mesurer par rapport à une horloge temps réel. La première contrainte concerne la durée d'exécution d'une réaction conduisant à produire un événement de sortie dû à l'arrivée d'un stimulus dans le système. La seconde concerne la durée qui s'écoule entre deux stimuli. Pour les aspects embarquabilité, il faut prendre en compte le nombre de ressources, c'est-à-dire le nombre d'opérateurs, de transformateurs et de média de communication.

Comme cela avait été souligné lors de l'introduction du chapitre 2, bien que l'on parle d'implantation optimisée, dans le cas général c'est une solution approchée que l'on recherche et non une solution optimale que l'on ne peut obtenir dans un temps raisonnable que pour des cas très simples. Dès que le nombre de sommets des graphes de l'algorithme et de l'architecture est de l'ordre de la dizaine, de telles solutions exactes ne sont pas humainement envisageables. On utilise alors des heuristiques que l'on désire à la fois rapides et donnant des résultats proches de la solution optimale. Ces deux caractéristiques sont bien sûr contradictoires. Dans le cas des applications de traitement du signal et des images, il est intéressant de choisir des heuristiques rapides afin de pouvoir essayer de nombreuses variantes d'implantation en fonction du coût et de la disponibilité des composants et de tester rapidement l'impact de l'ajout de nouvelles fonctionnalités. On peut ensuite chercher à améliorer les résultats de l'heuristique en restreignant, au moyen de contraintes de placement par exemple, l'espace des solutions qu'elle a à explorer, en particulier en coupant les "fausses pistes" menant à des minima locaux sans intérêt. On pourrait aussi, à partir d'une solution déjà bonne obtenue comme cela, utiliser des heuristiques plus lentes mais moins sensibles aux minima locaux, comme le "recuit simulé" par exemple.

Pour la latence, l'optimisation est basée sur des calculs de chemins critiques sur le graphe de l'algorithme, étiqueté par les durées des opérations et des transferts de données lorsqu'ils sont affectés respectivement aux opérateurs et aux routes. Les étiquettes sont déduites du modèle d'architecture caractérisé. Pour la cadence, l'optimisation est basée sur des recherches de boucles critiques identifiées par les retards sur le graphe de l'algorithme.

Les heuristiques basées sur des algorithmes "gloutons" sont très rapides car elles ne remettent pas en cause les résultats trouvés dans des étapes ultérieures à l'étape courante [13]. Elles sont aussi dites sans retour arrière.

2.4 Heuristique d'adéquation

2.4.1 Principes

Voici très brièvement les principes d'un exemple d'heuristique gloutonne dite de type "ordonnancement de liste" simple et performante [14]. La distribution et l'ordonnancement sont faits en même temps, on va tenter de construire un optimum global à partir d'optima locaux de la façon suivante. On part d'une liste d'opérations candidates, initialisée aux opérations sans prédécesseurs (les sommets d'entrée du graphe de l'algorithme, ainsi que ceux qui n'ont pour prédécesseur que des retards), et pour chacun de ces candidats on va déterminer à l'aide d'une fonction de coût celui des opérateurs sur lequel il sera affecté. Une fonction de coût simple revient à évaluer, lorsqu'on affecte une opération à un opérateur, de combien on allonge le chemin critique du graphe de l'algorithme étiqueté comme vu plus haut, tout en exploitant la marge d'ordonnancement de l'opération considérée. Cette dernière correspond à la différence entre sa date de début d'exécution au plus tôt et sa date de début d'exécution au plus tard. Bien sûr on prend aussi en compte le coût des transferts inter-partitions qui sont affectés à la route la plus courte, si plusieurs routes sont possibles. Cette première

phase conduit à déterminer pour chaque opération candidate le meilleur opérateur. On rappelle encore que le coût de l'affectation d'une opération à un opérateur et les coûts des transferts de données induits sont bien sûr calculés grâce à la caractérisation de l'architecture. Il peut y avoir plusieurs types d'opérateurs capables de réaliser la même opération, mais à des coûts différents. Par ailleurs, certains opérateurs ne peuvent réaliser que certaines opérations. On obtient ainsi un ensemble de couples (candidat, meilleur opérateur). On choisit maintenant parmi eux celui qu'il est le plus urgent d'affecter, c'est-à-dire celui qui a le moins de marge ou qui allonge le plus le chemin critique.

Dans le cas de l'optimisation de la cadence, on va considérer les retards et les boucles critiques qui leurs sont attachées pour évaluer les possibilités de "pipe-line" en vue de faire du "retiming" en déplaçant des retards affectés à des mémoires.

Jusqu'à présent on a fait l'hypothèse que le nombre total d'opérateurs, de transformateurs et de médias était donné a priori. On recherche dans ce cas à exploiter au mieux les ressources dont on dispose. Le problème peut être généralisé au cas où le nombre de ressources n'est pas donné a priori. Pour cela on se ramène tout d'abord au cas précédent en calculant la borne maximale des ressources. Dans le cas d'une architecture homogène (un seul type d'opérateur, un seul type de transformateur et un seul type de média), on peut calculer le nombre d'opérateurs au delà duquel il serait inutile d'en rajouter car cela ne conduirait plus à de l'accélération. Cette borne correspond à l'exploitation de tout le parallélisme potentiel de calcul, hors parallélisme pipe-line, lié à la spécification de l'algorithme d'application. Cette borne se calcule en prenant l'entier immédiatement supérieur au rapport entre la durée d'exécution monoprocesseur calculée en additionnant la durée de chaque opération du graphe de l'algorithme, et la durée du chemin critique calculée sans prendre en compte les durées de transferts de données. Cette valeur représente l'accélération maximale que l'on pourra jamais obtenir avec cet algorithme, toujours sans considérer les aspects pipe-line que l'on traite séparément à l'aide de calculs de boucles critiques. L'accélération effective sera celle obtenue en prenant le chemin critique réel après distribution et ordonnancement. Il prend en compte les durées des transferts de données affectés aux routes. Il est évident que l'accélération effective ne peut être supérieure à l'accélération maximale.

On peut tenter d'améliorer la solution obtenue de cette façon en diminuant itérativement le nombre d'opérateurs, de transformateurs et de médias et en effectuant de nouveau une distribution et un ordonnancement et en vérifiant si la contrainte de latence est toujours vérifiée.

2.4.2 Prévision de comportement temps réel

Les calculs de dates effectués lors de l'optimisation de la latence et de la cadence, permettent d'obtenir les dates de début et de fin de l'exécution des opérations et des transferts de données implantés sur l'architecture. Ceci permet de tracer un diagramme temporel décrivant une prévision de comportement temps réel de l'algorithme sur l'architecture. Ce point est important car il permet de faire l'évaluation des performances d'une application en simulant son comportement temps réel sans l'exécuter réellement. Il suffit d'avoir modélisé et caractérisé les composants que l'on veut utiliser pour construire son architecture. Cela permet par exemple d'évaluer, sans posséder réellement la nouvelle version d'un processeur, ce qu'il pourrait apporter en termes d'amélioration de performances.

2.5 Génération d'exécutifs distribués temps réel

Dès qu'une distribution et un ordonnancement ont été déterminés, il est assez simple de générer automatiquement des exécutifs distribués temps réel, principalement statiques avec une partie dynamique uniquement quand cela est inévitable (calcul des booléens de conditionnement à l'exécution) [15]. Ces exécutifs sont générés en installant un système de communication inter-processeur sans "deadlock" puisque l'on peut faire de la prévention d'interblocage[1]. Il faut encore insister ici sur le fait que dans les applications embarquées de traitement du signal et des images, l'exécutif qui supportera l'exécution distribuée temps réel doit être le moins coûteux possible. C'est pourquoi on évitera autant que faire se peut les exécutifs dynamiques, qui s'ils semblent faciles à mettre en œuvre pour l'utilisateur, conduisent à une sur-couche logicielle système, avec changements de contexte, dont le surcoût n'est pas négligeable. Avec l'approche préconisée, il suffit lorsqu'on construit l'exécutif, de conserver par un mécanisme simple d'exclusion mutuelle les propriétés d'ordre, montrées avec les langages Synchrones sur

1. Les interblocages ne peuvent provenir que de cycles de dépendance entre opérations, qui sont détectés au niveau du graphe de l'algorithme. L'heuristique de distribution et d'ordonnancement ne faisant que renforcer l'ordre partiel (sans cycle de dépendance) entre les opérations de l'algorithme, le graphe de l'algorithme implanté est également sans cycle de dépendance, donc l'exécutif généré est sans interblocage.

la spécification algorithmique, qui ont été conservées lors du choix de la distribution et de l'ordonnancement optimisés. Le processus de génération d'exécutif est parfaitement systématique : il correspond à ce que fait habituellement l'utilisateur à la main dans une approche classique après avoir effectué de nombreux tests en temps réel sur l'architecture réelle. Les exécutifs peuvent être totalement générés sur mesure en langage de haut niveau ou en assembleur, en faisant appel à des fonctions utilisateurs C compilées séparément si nécessaire. Ils peuvent aussi faire appel à des primitives d'un noyau d'exécutif distribué temps réel standard résident tels que Virtuoso, Lynx, Osek etc. Cependant il faut être conscient que cela augmente le surcoût des exécutifs.

Afin d'évaluer les performances temps réel des implantations réalisées, on peut générer les exécutifs avec des macro-opérations de chronométrage insérées entre les macro-opérations de calcul. Le chronométrage s'effectue en trois phases : sur chaque processeur on mesure des dates de début et de fin des opérations et des transferts à l'aide de son horloge temps réel, puis en fin d'exécution de l'application on recale les horloges temps réel des différents processeurs, enfin on collecte les mesures mémorisées sur chaque processeur et on les transfère sur le processeur hôte qui possède des moyens de stockage de masse. Ces mesures sont ensuite analysées afin de les comparer à celles calculées lors de la prédiction de comportement temps réel. Ceci permet d'une part d'obtenir la première fois les durées de calcul et de communication qui permettent de caractériser le modèle d'architecture, puis les fois suivantes d'évaluer l'écart entre les modèles d'architecture utilisés et l'architecture réelle.

Lorsqu'on a suffisamment confiance en ces modèles, il est très facile de réaliser des variantes d'implantation en vue d'introduire de nouvelles fonctionnalités dans l'application, en modifiant l'algorithme et le nombre de composants de l'architecture, puis en effectuant une implantation optimisée et en générant l'exécutif. Cela est habituellement beaucoup plus difficile avec une approche classique n'utilisant pas l'approche Adéquation Algorithme Architecture puisqu'il faut réaliser puis tester un nouveau prototype complet pour chaque variante d'implantation.

2.6 Logiciel SynDEx

SynDEx [16, 17] est un logiciel de CAO niveau système, supportant la méthodologie AAA, pour le prototypage rapide et pour optimiser la mise en œuvre d'applications distribuées temps réel embarquées. C'est un logiciel graphique interactif qui offre les services suivants :

- spécification et vérification d'un algorithme d'application saisi sous la forme d'un graphe flot de données conditionné (ou interface avec les langages Synchrones tels que SIGNAL) ;
- spécification d'un graphe d'architecture multicomposant (processeurs et/ou composants spécialisés) ;
- heuristique pour la distribution et l'ordonnancement de l'algorithme d'application sur l'architecture, avec optimisation du temps de réponse ;
- visualisation de la prédiction des performances temps réel pour le dimensionnement de l'architecture ;
- génération des exécutifs distribués temps réel, sans interblocage et principalement statiques, avec mesure optionnelle des performances temps réel. Ceux-ci sont construits, avec un surcoût minimal, à partir d'un noyau d'exécutif dépendant du processeur cible. Actuellement des noyaux d'exécutifs sont fournis pour : SHARC, TMS320C40, i80386, MC68332, i80C196 et stations de travail Unix ou Linux. Des noyaux pour d'autres processeurs sont facilement portés à partir des noyaux existants.

Puisque les exécutifs distribués sont générés automatiquement, leur codage et leur mise au point sont éliminés, réduisant de manière importante le cycle de développement.

Chapitre 3

Modèle d'exécutif

3.1 Introduction

Sur une architecture programmable, l'exécutif est la partie du logiciel qui gère les ressources matérielles pour les allouer aux besoins de l'algorithme d'application, ou à des besoins annexes tels le chronométrage. Les ressources à allouer sont, sur une architecture multi-composants vue au niveau macroscopique :

- les espaces mémoires des processeurs, pour le code et pour les données,
- le temps des séquenceurs d'instructions, un par opérateur, partagé comme on l'a vu au chapitre 1.4.3 entre une séquence de calcul et des séquences de communication (une pour chacun des média de communication accessible par l'opérateur),
- le temps des média de communication, et plus précisément pour chaque média le temps des séquenceurs de transferts qui partagent le média.

Les exécutifs traditionnels étant conçus séparément des applications qu'ils supportent, ils fournissent une plétore de services pour tenter de répondre aux besoins variés des concepteurs d'applications, qui utilisent des méthodes variées de conception et d'implantation de leurs algorithmes. Certains concepteurs ne veulent consacrer qu'une minimum de leur temps à l'implantation de leur algorithme, aussi attendent-ils de l'exécutif qu'il leur fournisse des services d'allocation automatique des ressources, au prix de surcoûts non seulement d'allocation dynamique (migration de processus pour équilibrage de charge des processeurs, relocation des exécutables au chargement, ramasse-miettes ou "garbage collector", multitâche en temps partagé ou "time-slicing", paquetage et routage des communications ...) mais aussi d'auto-protection de l'exécutif contre d'éventuelles erreurs de programmation (vérification de la cohérence des requêtes d'allocation, procédures d'exception en cas d'incohérence ou d'insuffisance de ressource disponible, chiens de garde de détection de perte de messages ou d'interblocage ...). D'autres concepteurs veulent tirer parti au maximum des performances de l'architecture, aussi attendent-ils de l'exécutif qu'il leur fournisse des services d'allocation dits "de bas niveau" (possibilité d'écrire des programmes d'interruption à faible temps de réponse, contrôle du gestionnaire mémoire et du cache pour accélérer les temps d'accès, contrôle de la politique d'ordonnancement du gestionnaire multitâche ...) pour réduire les surcoûts cités ci-dessus, mais au prix d'un temps de mise au point plus important.

Grâce à la méthodologie Adéquation Algorithme Architecture (AAA) présentée au chapitre précédent, et au logiciel SynDEx [16, 17] qui la supporte, il n'est plus nécessaire de penser les exécutifs en ces termes, car un maximum de décisions et de vérifications d'allocation de ressources sont prises par l'heuristique d'optimisation, avant l'exécution, donc ces décisions peuvent être traduites automatiquement, sans que l'utilisateur n'ait à l'écrire ni à le déboguer, en un code distribué qui, sur chaque processeur, séquence les macro-opérations de calcul, alloue les macro-registres nécessaires au transfert des données entre macro-opérations, et séquence les macro-opérations de communications inter-processeurs en utilisant les mêmes macro-registres que la séquence de calcul avec des macro-opérations de synchronisation pour assurer leur accès en exclusion mutuelle. *Ce code distribué est en fait un exécutif généré sur mesure pour l'algorithme et l'architecture.* La suffisance des ressources ayant été vérifiée avant l'exécution par l'heuristique d'optimisation, et le code étant généré automatiquement par le logiciel SynDEx, l'exécutif n'a plus à effectuer ces vérifications à l'exécution ni à se protéger contre d'éventuelles erreurs de programmation, donc son surcoût, autant en espace code qu'en temps d'exécution, est moindre que dans le cas traditionnel. Et surtout, le temps que consacre traditionnellement l'utilisateur à écrire et

à déboguer le code distribué de son application, peut être plus utilement consacré à optimiser son implantation (en interagissant avec l'heuristique d'optimisation pour l'aider à trouver de meilleurs résultats si possible ou pour découvrir les goulots d'étranglement de son application, ce qui peut l'amener à adapter la taille des grains de son algorithme, et à minimiser les ressources de son architecture).

Dans le cadre de la méthodologie AAA, l'exécutif n'est donc pas conçu comme un "noyau résident" compilé et chargé indépendamment de l'application, mais plutôt comme l'ossature, la partie intégrante de l'application, générée sur mesure en fonction de l'algorithme et de l'architecture, à partir d'une bibliothèque générique de *macros d'allocation de ressources*, que nous appelons "noyau générique d'exécutif". L'exécutif est compilé et/ou lié avec les macro-opérations spécifiques à l'application (insérées en ligne dans l'exécutif ou compilées séparément), pour produire un code binaire chargeable et exécutable directement sur l'architecture matérielle, ou avec le support d'un système d'exploitation si son usage est incontournable.

Pour que le générateur d'exécutifs s'adapte facilement à différents types de processeurs cibles (de stations de travail avec un système d'exploitation, de traitement du signal nus, sans système d'exploitation, ou même des microcontrôleurs 8 bits), il a été scindé en deux parties :

- La première partie traduit la distribution et l'ordonnancement produits par l'heuristique d'optimisation en un macro-code intermédiaire, comprenant pour chaque processeur des appels de macros d'allocation mémoire, une séquence d'appels de macros de calcul et autant de séquences d'appels de macros de communication que de médias de communication accessibles au processeur. Ce macro-code est générique, c'est-à-dire indépendant du langage cible préféré pour chaque type de processeur, aussi est-il intégré au logiciel SynDEx.

- La seconde partie traduit le macro-code intermédiaire en code source compilable par la chaîne de compilation spécifique à chaque type de processeur cible. Cette seconde partie est basée sur le macro-processeur m4 (standard sous Unix, version GNU), complété pour chaque type de processeur cible par un fichier de définitions de macros spécifiques à ce type de processeur. Nous appelons "noyau générique d'exécutif" le jeu de macros, extensible et portable entre différents langages cibles (de haut niveau ou assembleur). Pour chaque nouveau type de processeur, si l'on est moins concerné par la performance de l'exécutif que par le temps de portage des macros, on peut par exemple adapter le jeu de macros générant du C, déjà développé pour stations de travail Unix, sinon on peut redéfinir les macros pour qu'elles génèrent de l'assembleur, en s'inspirant des jeux de macros déjà développés pour quelques processeurs populaires.

Le reste de ce chapitre est consacré à la description de la structure du macrocode intermédiaire généré.

3.2 Structure du macrocode intermédiaire

Le code intermédiaire généré pour l'exécutif distribué d'une application est constitué :

- d'un fichier source pour chaque processeur, macro-codant l'exécutif dédié à ce processeur, qui sera traduit en source compilable pour ce processeur,

- et d'un fichier source macro-codant la topologie de l'architecture, qui sera traduit en makefile pour automatiser les opérations de compilation.

La structure du code intermédiaire généré pour chaque processeur est directement issue de la distribution et de l'ordonnancement des calculs de l'algorithme et des communications interprocesseur qui en découlent.

Dans le cadre des systèmes réactifs, les algorithmes sont par nature itératifs : un ensemble d'opérations de calcul est exécuté répétitivement jusqu'à ce qu'il soit décidé de mettre fin à l'application. La distribution et l'ordonnancement de cette itération sur plusieurs processeurs se traduit sur chaque processeur par *une séquence itérative d'opérations de calcul.*

De même, la distribution et l'ordonnancement sur plusieurs médias de communication interprocesseur, des transferts de données entre opérations exécutées sur des processeurs différents, se traduit sur chaque processeur par *une séquence itérative d'émissions et/ou de réceptions pour chacun des médias connecté au processeur.*

Sur chaque processeur, la séquence de calculs et les séquences de transferts *sont exécutées en parallèle et synchronisées* par l'intermédiaire de sémaphores qui imposent les alternances entre écriture et lecture des tampons mémoire partagés entre opérations exécutées dans des séquences parallèles. L'ordre partiel d'exécution des opérations, spécifié par les arcs dépendances de données du graphe de l'algorithme, est ainsi imposé dans le code généré, soit statiquement par chaque séquence, soit dynamiquement par l'intermédiaire des synchronisations (et transitivement des transferts interprocesseurs).

3.3 Structure du macrocode pour chaque processeur

Pour chaque processeur, le code intermédiaire comprend dans l'ordre :

1. Un commentaire d'estampillage de la version du générateur et de la date de génération du code, contenant également le nom de l'application et du processeur auquel le code est destiné, et le commentaire du concepteur décrivant succintement l'application.
2. Une macro `processor_` prenant en argument le nom que le concepteur a donné au processeur dans son graphe d'architecture, qui sert à insérer un préambule assembleur et/ou tout code généré dans une "diversion" (zone de stockage temporaire) pendant le macroprocessing des macros du noyau générique qui précède le macroprocessing des macros suivantes du macrocode intermédiaire.
3. Une liste de macros de chargement arborescent des programmes.
4. Une liste de macros d'allocation mémoire : déclarations de tampons de données, de sémaphores, et optionnellement de tampon de chronométrage.
5. Une liste de macros pour chaque séquence de communications.
6. Une liste de macros pour la séquence de calculs.
7. Une macro `end_` sans argument, qui sert à insérer un épilogue assembleur et/ou tout code généré dans une "diversion" pendant le macroprocessing des macros précédentes du macrocode intermédiaire.

Pour un exemple concret de macrocode généré, le lecteur peut se reporter au chapitre 5.8 pour un exécutif monoprocesseur, sans séquences de communication, et au chapitre 5.9 pour un exécutif biprocesseur, avec une séquence de communication sur chacun des deux processeurs.

3.4 Chargement arborescent des programmes

L'exécutif doit tout d'abord supporter le chargement initial des mémoires des processeurs. Habituellement, un seul processeur "hôte" est équipé de mémoire de masse non volatile, disque ou EPROM, contenant les programmes à charger sur les autres processeurs. Nous ne nous préoccuperons pas ici du chargement du processeur hôte, mais uniquement de celui des autres processeurs, effectué à partir du hôte.

On supposera que l'hôte démarre ("boot") à partir de sa mémoire de masse, charge éventuellement un système d'exploitation pour gérer des entrées-sorties standard ("stdio" : clavier, écran, disque), et au besoin requiert la saisie par un utilisateur d'une commande de lancement pour terminer son propre chargement. C'est à partir du point d'entrée du `main` du hôte que l'on s'intéresse ici à générer l'exécutif.

On supposera également que le démarrage ("reset") de chacun des autres processeurs est commandé par le processeur hôte, et que SynDEx choisit un arbre de couverture du graphe d'interconnexion des processeurs, ayant pour racine le processeur hôte (d'où son identification sous SynDEx par le nom "root").

Si l'hôte a un accès direct à la mémoire programme de ses descendants dans l'arbre, alors il chargera leur programme avant de commander leur démarrage. Sinon, on supposera que chaque processeur a une mémoire non volatile de démarrage contenant un programme de chargement à travers une liaison physique de communication, auquel cas l'hôte commandera le démarrage de tous les processeurs et leur transmettra leur programme, au besoin par l'entremise des processeurs intermédiaires dans l'arbre.

Ainsi, la phase initiale du programme du hôte consiste à extraire de sa mémoire de masse le programme de chaque processeur et à le lui transmettre, au besoin par l'entremise des processeurs intermédiaires dans l'arbre. La phase initiale du programme de chacun des autres processeurs consiste à recevoir de son ascendant dans l'arbre et à transmettre à chacun de ses descendant, le programme qui lui est destiné. Les processeurs feuilles de l'arbre n'ont donc rien à faire pendant cette phase initiale puisqu'ils n'ont pas de descendant dans l'arbre de couverture du graphe de processeurs.

Cette arborescence est codée, au début du source de chaque processeur, par des macros désignant, pour l'ascendant et pour chaque descendant du processeur, le média qui l'y connecte.

3.5 Allocation mémoire

Chaque tampon mémoire, utilisé pour stocker une dépendance de donnée (c'est-à-dire un résultat intermédiaire entre deux opérations d'entrée-sortie, de calcul ou de transfert), est déclaré par une macro d'allocation mémoire. L'allocation est statique, c'est-à-dire effectuée lors de la compilation, l'adresse et la taille de la zone mémoire allouée restent invariantes au cours de l'exécution. Le même nom de tampon est passé en argument :

- de la macro d'allocation,
- de la macro qui code l'opération à l'origine de la dépendance de donnée, qui à l'exécution produira un résultat intermédiaire et l'écrira dans le tampon,
- de chacune des macros qui codent les opérations aux extrémités de la dépendance de donnée (il peut y en avoir plusieurs en cas de diffusion), qui chacune à l'exécution lira le résultat intermédiaire dans le tampon.

3.6 Séquences de calcul et de communications

Chaque séquence itérative est codée par une liste de macros (une pour chaque opération d'entrée-sortie, de calcul, de transfert, de synchronisation ou de contrôle structuré de répétition ou d'exécution conditionnelle), composée de trois phases : initialisation, itération, et finalisation. Dans le cas d'une séquence de calculs :

1. la phase d'initialisation commence par la macro `main_ini_` marquant le point d'entrée après le "boot" du processeur, suivie de macros d'initialisation des constantes et des retards, de macros d'initialisation des interfaces avec l'environnement, d'une macro `spawn_` de lancement pour chaque séquence de communication exécutée sur le même processeur, et optionnellement d'une macro `Chrono_ini_` d'initialisation du chronométrage ;
2. la phase itérative comprend des macros d'opérations de calcul et d'interface avec l'environnement exécutées sur le processeur, dont certaines sont précédées et/ou suivies de macros de synchronisation, le tout encadré par deux macros de contrôle itératif `while_` et `endwhile_` ; optionnellement, des macros de chronométrage sont intercalées entre les macros de calcul et d'interface ;
3. la phase de finalisation comprend des macros de désactivation des interfaces avec l'environnement, optionnellement une macro `Chrono_end_` de collecte des chronométrages, et se termine par la macro `main_end_` marquant la fin de la séquence.

Dans le cas d'une séquence de transferts :

1. la phase d'initialisation commence par la macro `thread_` étiquetant le point d'entrée de cette séquence de transferts (à chaque macro `thread_` correspond une macro `spawn_` de lancement dans la phase d'initialisation de la séquence de calculs), suivie d'une macro d'initialisation du coprocesseur de transferts (DMA), suivie de macros de synchronisations initiales ;
2. la phase itérative comprend des macros d'émission ou de réception, chacune précédée et/ou suivie de macros de synchronisation, le tout encadré par deux macros de contrôle itératif `while_` et `endwhile_` ;
3. la phase de finalisation comprend une macro de désactivation du DMA.

Pour un exemple concret de macrocode généré, le lecteur peut se reporter au chapitre 5.9 pour un exécutif biprocesseur, avec une séquence de communication sur chacun des deux processeurs.

3.7 Implantation des dépendances de données

De manière générale, chaque opération de calcul est codée par un appel de macro prenant en arguments les noms (c'est-à-dire les adresses) des tampons mémoire où l'opération doit lire ses opérandes et écrire ses résultats, dans la mémoire du processeur qui l'exécute.

Comme on ne sait pas en détail quand, pendant sa durée d'exécution, l'opération accède à ses tampons mémoire (cela dépend de son codage, que l'on désire conserver encapsulé), le pire cas consiste à considérer que

chacun de ses tampons mémoire doit être disponible, en lecture pour un tampon d'entrée ou en écriture pour un tampon de sortie, pendant toute la durée d'exécution de l'opération.

Pour qu'une dépendance de donnée entre deux opérations soit correctement implantée, il faut que l'écriture de la donnée dans le tampon correspondant, par l'opération productrice, précède la lecture de cette même donnée par l'opération consommatrice, donc il faut que la fin de l'opération productrice précède le début de l'opération consommatrice. L'exécution des opérations étant itératives, il faut aussi s'assurer qu'une donnée produite soit consommée avant d'être écrasée par la donnée produite par la même opération lors de l'itération suivante.

Il doit donc y avoir alternance entre écriture (par une et une seule opération productrice) et lecture(s) (par une ou plusieurs opérations consommatrices) d'un même tampon.

Implantation des dépendances de données intra-processeur

Lorsque les deux opérations sont exécutées par le même processeur, il suffit que l'exécution de l'opération productrice soit séquencée dans la même itération avant celle de l'opération consommatrice, et que la donnée soit communiquée entre les deux par l'intermédiaire du même tampon (dans la mémoire du processeur) dont l'adresse est passée en argument des deux opérations. L'exécution séquentielle des deux opérations garantit l'alternance énoncée ci-dessus entre écriture et lecture(s) du tampon partagé, sans qu'il y ait besoin d'autre moyen de synchronisation.

Implantation des dépendances de données inter-processeur

Lorsque les deux opérations sont exécutées par des processeurs différents, il faut d'une part rendre accessible, au processeur exécutant l'opération consommatrice, le résultat de l'opération productrice exécutée par l'autre processeur, et d'autre part imposer l'alternance entre écriture et lecture(s) énoncée ci-dessus, par des moyens de synchronisation qui dépendent de la nature (indexable ou séquentielle) du média de communication inter-processeur. L'implantation des dépendances de données interprocesseur est décrite dans chacun des deux cas (média indexable et média séquentiel) dans les sections suivantes.

3.7.1 Communication et synchronisation par média RAM

Un média RAM est une mémoire indexable "à accès aléatoire" (Random Access Memory), c'est-à-dire permettant d'y lire les données dans un ordre différent de celui où elles y ont été écrites ; cette catégorie comprend les mémoires partagées, statiques ou dynamiques, à bus d'accès unique ou multiple. Cette différence possible d'ordre entre écritures et lectures permet un décalage temporel quelconque entre l'écriture d'une donnée et sa lecture plus tard : pour cette raison, les communications par média RAM sont souvent dites "asynchrones", qualificatif que nous éviterons d'employer ici d'une part pour sa confusion possible avec les qualificatifs "synchrone" ou "asynchrone" relatifs aux protocoles matériels d'échange des données sur les bus connectant mémoires et processeurs, et d'autre part parce qu'il est question ici uniquement de communications *synchronisées*.

Par rapport à un tampon mémoire partagé entre deux processeurs, appelons "producteur" l'opération qui y écrit, exécutée par l'un des deux processeurs, et "consommateur" l'opération qui y lit, exécutée par l'autre processeur. Il ne faut pas qu'une donnée soit lue dans un tampon avant d'y avoir été écrite, ni qu'elle y soit écrasée par une nouvelle donnée avant d'y avoir éte lue par tous les consommateurs qui l'utilisent (il peut y en avoir plusieurs en cas de diffusion). Il faut donc garantir la synchronisation entre producteur et consommateur(s), c'est-à-dire :

- d'une part que l'exécution du producteur se termine avant le début de l'exécution du consommateur : appelons cette précédence *tampon-plein*,
- d'autre part que l'exécution du consommateur se termine avant le début de l'exécution suivante du producteur lors de l'itération suivante : appelons cette précédence *tampon-vide*.

Ce shéma de synchronisation est très général, il se transpose directement au niveau matériel dans le cas des circuits dits "asynchrones", où les précédences tampon-plein et tampon-vide sont habituellement dénommées respectivement "data set ready" ou "strobe" pour la première, et "data acknowledge" ou "data request" pour la seconde.

Une précédence se projette de manière différente sur les deux séquences qui se synchronisent. Appelons `Pre` la macro de synchronisation correspondant à la projection côté prédécesseur et `Suc` la macro de synchronisation correspondant à la projection côté successeur. `Suc` ne doit pas se terminer avant la fin du `Pre` correspondant, par contre `Pre` peut se terminer avant la fin du `Suc` correspondant. Donc `Suc` est *bloquante* (si elle est exécutée

plus tôt elle doit attendre l'exécution de `Pre`) alors que `Pre` est *passante* (si elle est exécutée plus tôt, elle peut se terminer sans attendre).

En termes de sémaphores, `Pre` et `Suc` correspondent respectivement aux opérations indivisibles `V` de libération et `P` de prise du sémaphore, aussi souvent appelées respectivement `Signal` et `Wait` dans les exécutifs traditionnels.

La mise en correspondance d'un `Pre` et d'un `Suc` se fait en passant aux deux macros le même argument identifiant la précédence, et qui est dans la plupart des implantations un index dans un tableau de sémaphores. Ce tableau est alloué et initialisé par une macro spécifique, générée avec les autres macros d'allocation mémoire, avant les séquences de macros, afin que les sémaphores soient alloués avant d'être référencés.

En pratique, la structure du code intermédiaire généré est la suivante pour chaque opération accédant à des tampons partagés par plusieurs séquences :

- Pour chaque tampon de sortie partagé, la macro de l'opération est suivie d'un `Pre` tampon-plein, et précédée d'un `Suc` tampon-vide. Pour chaque `Suc` tampon-vide, il y a un `Pre` tampon-vide initial, dans la phase initiale de la séquence de communication dont la phase itérative utilise le même sémaphore.
- Pour chaque tampon d'entrée partagé, la macro de l'opération est précédée d'un `Suc` tampon-plein, et suivie d'un `Pre` tampon-vide.

Dans le cas où plusieurs consommateurs utilisent la même donnée, il faut un `Suc` tampon-plein avant chacun d'eux et un `Pre` tampon-vide après chacun. Dans le cas où plusieurs consommateurs utilisant la même donnée sont exécutés dans la même séquence, il suffit d'un seul `Suc` tampon-plein avant le premier dans la séquence et d'un seul `Pre` tampon-vide après le dernier. Une analyse des précédences au niveau global peut également permettre d'éliminer les précédences tampon-vide qui incluent une chaîne causale de précédences tampon-plein.

3.7.2 Communication et synchronisation par média SAM

Un média SAM est une mémoire à accès séquentiel (Sequential Access Memory), c'est-à-dire imposant matériellement que les données y soient lues dans l'ordre où elles y ont été écrites ; cette catégorie comprend les mémoires FIFO (First-In First-Out), et plus généralement toute liaison série ou parallèle, point-à-point ou multipoint, permettant le transfert de données en mode FIFO entre deux ports de communication. Cette identité d'ordre entre écritures et lectures impose d'une part que la i-ième écriture soit terminée avant que la i-ième lecture puisse commencer, et d'autre part, pour une mémoire FIFO de n cellules, que la i-ième lecture soit terminée avant que la $(i+n)$-ième écriture puisse commencer. Pour une communication par média SAM d'un macro-registre de taille supérieure à n cellules, il y a donc forcément recouvrement temporel entre son écriture et sa lecture (la lecture de la première cellule débute forcément avant la fin de l'écriture de la dernière) : pour cette raison, les communications par média SAM sont souvent dites "synchrones", qualificatif que nous éviterons d'employer ici pour les mêmes raisons que pour les communications par média RAM. Il est préférable de dire que les communications par média SAM sont "matériellement synchronisées", ou plus généralement qu'elles sont soumises à un contrôle de flux[1].

La communication directe par média SAM, où un macro-registre serait directement écrit (resp. lu) dans la FIFO par une macro-opération productrice (resp. consommatrice), poserait plusieurs problèmes :

- Le codage de chaque macro-opération dépendrait de la nature des médias, indexables ou séquentiels, dans lesquels elle devrait lire ses macro-registres opérandes ou écrire ses macro-registres résultats. Il faudrait donc soit avoir plusieurs versions de chaque macro-opération (mais il en faudrait 2^n pour une opération prenant n arguments !), soit ne passer au codage qu'après avoir décidé de la distribution des macro-opérations sur les opérateurs (c'est une pratique malheureusement courante, mais qui nécessite un recodage à chaque modification de la distribution).
- Alors qu'une même donnée peut être lue plusieurs fois dans un média RAM, elle ne peut l'être qu'une seule fois dans un média SAM, donc si une macro-opération productrice doit diffuser le même résultat à plusieurs macro-opérations consommatrices, elle devra l'écrire plusieurs fois, non seulement une fois pour chaque média RAM, mais aussi et surtout une fois pour chacune des macro-opérations consommatrices qui liront la donnée dans un média SAM. Le codage des macro-opérations dépendrait donc également de la connectivité du graphe de dépendance de données entre macro-opérations.

1. Par exemple, dans le cas du TMS320C40 les deux tampons FIFOs de réception et d'émission sont limités à un total de 16 mots de 32 bits et le contrôle de flux est assuré par des signaux de synchronisation transmis parallèlement aux signaux de données. Par contre, dans le cas d'une liaison série sans signaux matériels de synchronisation, le contrôle de flux doit être assuré par l'envoi en sens inverse de données de synchronisation ; c'est ce qui est réalisé dans le cas du Transputer, où la transmission en sens inverse de bits de sychronisation est supportée directement par les DMA pour assurer le contrôle de flux.

– Le décalage limité, entre écritures et lectures dans la mémoire FIFO, imposerait un couplage entre les exécutions des macro-opérations productrice et consommatrice(s), qui aurait souvent pour effet de provoquer des attentes de chaque coté (dues à des rythmes différents d'écriture et de lecture) et donc de ralentir l'exécution.

Pour toutes ces raisons, on préfère découpler calculs et "communications par média SAM" en intercalant un coprocesseur DMA, qui communique et se synchronise avec l'opérateur de calcul par l'intermédiaire d'un média RAM partagé, et effecture les transferts entre le média RAM et le média SAM, pendant que l'opérateur effectue d'autres calculs. Ainsi chaque macro-opération peut être codée indépendamment de la distribution et de la connectivité du graphe de dépendance de données, et les instructions qui la composent peuvent être exécutées indépendamment de (sans couplage avec) celles des macro-opérations avec lesquelles elle est en relation de dépendance de données, donc sa durée d'exécution sera moindre et plus facilement caractérisable. Par ailleurs, comme le DMA effectue uniquement des transferts entre deux médias, il peut tirer le maximum de bande passante des deux, c'est-à-dire saturer le média le plus lent (habituellement le média SAM).

Donc, dans le cas d'une dépendance de donnée par l'intermédiaire d'un média SAM de communication, il faut que la valeur produite (disponible à la fin de l'exécution de l'opération productrice) soit transférée depuis la mémoire du processeur qui a exécuté l'opération productrice, vers la mémoire du processeur qui doit exécuter l'opération consommatrice (il doit l'exécuter après la fin du transfert, car la donnée doit être disponible avant le début de l'exécution de l'opération consommatrice).

Les acteurs du transfert entre mémoires sont les opérateurs DMA connectés au média SAM de communication. Chaque opération de transfert doit donc être "projetée" sur chaque processeur, sous la forme soit d'une *émission*, soit d'une *réception* (soit encore d'une attente de la fin du transfert dans le cas d'un DMA ni émetteur ni récepteur quand le média de communication est partagé par plus de deux DMA).

Pour que les séquences de calcul soient synchronisées entre elles, il suffit donc qu'elles soient synchronisées localement avec les séquences de transfert par l'intermédiaire desquelles elles communiquent, car ces séquences de transfert sont matériellement synchronisées entre elles par l'intermédiaire du média SAM qu'elles partagent. Si les deux processeurs ne sont pas directement connectés, le transfert se fait à travers plusieurs médias avec l'aide de processeurs intermédiaires de routage.

Sur un même processeur, la séquence d'opérations de calcul communique avec une séquence de transferts (ou, dans le cas d'un processeur de routage, deux séquences de transferts communiquent directement entre elles) par l'intermédiaire de la mémoire du processeur, média RAM qu'elles partagent. C'est dans cette mémoire partagée qu'une opération de calcul lit ses opérandes et écrit ses résultats, et que sont lues les données émises et écrites les données reçues. Pour ne pas payer un surcoût inutile de recopie mémoire (comme c'est le cas avec la plupart des exécutifs traditionnels, et comme c'était aussi le cas avec l'exécutif SynDEx v3), les données émises sont lues dans le tampon même où elles ont été écrites par l'opération qui les a produites, et réciproquement les données reçues sont écrites dans le tampon même où elles seront lues par l'opération qui les consommera ; comme la lecture d'un tampon dans une mémoire à accès aléatoire (média RAM) n'en change pas le contenu, plusieurs opérations peuvent consommer la même donnée dans le même tampon.

La synchronisation entre séquence de calculs et séquence de communications, pour assurer une alternance entre écriture et lecture(s) lors de leurs accès aux tampons partagés, correspond donc au cas de la communication par média RAM décrit dans la section précédente, où le producteur d'un tampon peut être soit une opération de calcul qui y écrit son résultat, soit une opération de réception qui y écrit les données reçues, et où le consommateur d'un tampon peut être soit une opération de calcul qui y lit son opérande, soit une opération d'émission qui y lit les données à émettre. Il existe cependant une différence entre les deux cas :

– Dans le cas de la section précédente, chaque séquence est exécutée par un processeur différent, possédant chacun son propre séquenceur d'instructions.

– Dans le cas de la section présente, la séquence de communications est exécutée en grande partie par le DMA, qui séquence les micro-transferts de chaque macro de communication (cf. chap1.4), mais elle requiert aussi, pour séquencer les macros de communication et de synchronisation, le séquenceur d'instructions du processeur, qui est sinon alloué à la séquence de calculs ; c'est donc le même séquenceur d'instructions qui est partagé pour exécuter les macros de synchronisation dans ce cas.

En conséquence, les macros de synchronisation `Pre` et `Suc` ne seront pas implantées de la même manière dans les deux cas, comme on le verra en détails au chapitre 4.8.2.

Chapitre 4

Spécification du noyau générique d'exécutif SynDEx v4

Après le chapitre précédent qui décrit les principes de fonctionnement et la structure du macrocode intermédiaire des exécutifs générés, ce chapitre spécifie la syntaxe et la sémantique des macros du jeu de macros constituant le noyau générique des exécutifs SynDEx v4.

Des exemples génériques de codage et d'utilisation des macros illustrent les spécifications pour faciliter la compréhension des noyaux d'exécutifs déjà développés et le travail du programmeur d'un nouveau noyau générique spécifique à un nouveau type de processeur. Ces exemples utilisent pour langage cible un pseudo-assembleur très fortement inspiré du langage C, avec en plus des pseudo-instructions d'appel et de retour de sous-programme (`call(label)` et `returnFromCall`) pour décrire les transferts de contrôle entre séquences ("changements de contexte").

4.1 Introduction au macro-processeur m4

Un macro-processeur est un programme qui consomme en entrée une chaîne de caractères (texte source), la traite séquentiellement en susbstituant chaque sous-chaîne qu'il reconnait ("appel de macro") par une chaîne correspondante de substitution ("définition de macro") qu'il traite à nouveau jusqu'à ce qu'il n'y ait plus de substitution possible, et qui produit en sortie la chaîne de caractères traitée.

Dans un macro-processeur, il y a un dictionnaire de "macros" qui associe chaque nom de macro à reconnaître avec sa définition. Un appel de macro peut comprendre, juste après le nom de la macro, une liste de sous-chaînes arguments qui sont substituées, pendant le processus de substitution de l'appel de la macro par sa définition, aux paramètres formels trouvés dans la définition de la macro. Un certain nombre de macros sont prédéfinies au démarrage du macro-processeur dont au moins une qui permet de définir de nouvelles macros.

Pour m4, le macro-processeur standard des systèmes d'exploitation Unix :

- Toute chaîne encadrée par un caractère 'backquote à gauche et un caractère 'quote à droite (et pouvant contenir des backquotes et des quotes balancés) est substituée directement, sans nouvelle tentative de substitution, par la même chaîne sans le premier caractère backquote ni le dernier caractère quote.
  Par exemple, `'hello''world'` est substituée par `hello''world`.

- Les noms de macros ne peuvent être constitués que des caractères alphabétiques, numériques et du caractère "_" (underscore) et ne doivent pas commencer par un caractère numérique (expression régulière `[_A-Za-z][_0-9A-Za-z]*`).
  Par exemple `foo X1 _1z` sont trois noms de macros possibles.

- Les sous-chaînes arguments d'un appel de macro sont séparées par des virgules et leur liste est encadrée entre parenthèses (la parenthèse ouvrante doit être le premier caractère qui suit le nom de la macro).
  Par exemple, `foo(un,(2))` appelle la macro `foo` avec deux arguments `un` et `(2)`.

- La macro `define(name,subs)` est substituée par une chaîne vide, mais a pour effet de bord de définir une nouvelle macro de nom `name` et de définition `subs`. Pendant une substitution, `$n` sera substitué par le *n*-ième argument de la macro en cours de susbtitution (et `$0` par le nom de cette macro). Par exemple, `define('add','$0q $1+$2')add(un,(2))` est substitué par `addq un+(2)`.

– La macro `dnl` est substituée, ainsi que les caractères qui la suivent jusqu'à y compris le premier caractère de fin de ligne suivant, par une chaîne vide (utile pour commenter et formater les sources).

Pour en savoir plus sur les autre macros prédéfinies, consulter la documentation du macro-processeur m4, dans sa version GNU qui offre en plus du standard des macros manipulant des "expressions régulières".

Les exemples de définition et de substitution des macros, qui illustrent les spécifications des macros, sont imprimés en caractères de style `teletype` et chaque appel de macro dont la substitution n'est pas vide est suivi de sa substitution, sur une ligne séparée commençant par un caractère "|" qui ne fait pas partie de la substitution.

4.2 Règles de nommage des macros

Pour éviter des conflits de noms entre macros générées, on a suivi les règles de nommage suivantes :

1. Les noms fournis par le concepteur de l'application (pour identifier les sommets du graphe de l'algorithme et de l'architecture, leurs ports, et la macro à générer pour chaque opération) sont constitués d'une chaîne de caractères alphanumériques avec initiale alphabétique (comme dans la plupart des autres langages, expression régulière `[A-Za-z][0-9A-Za-z]*`), mais sans caractère "underscore" qui est réservé pour constituer des noms sans conflit avec les premiers.

2. Le nom identifiant le tampon mémoire d'une connexion est constitué en concaténant par un "underscore" le nom identifiant le sommet et le nom identifiant le port de sortie à l'origine de la connexion.

3. Les macros d'initialisation, d'itération et de finalisation d'une opération d'entrée (sans port d'entrée) ou de sortie (sans port de sortie) sont identifiées en concaténant au nom fourni par le concepteur un "underscore", un suffixe les différenciant, et un second "underscore".

4. Les macros du noyau générique d'exécutif sont identifiées par un nom suffixé par un "underscore".

5. Les commentaires sont encadrés par des accolades et le mot clé `comment` comme dans l'exemple suivant :

   ```
   comment{Ceci est un commentaire}comment
   ```

4.3 Structure du jeu de macros

Le jeu de macros, que nous appelons aussi "noyau générique d'exécutif", est conceptuellement structuré en plusieurs familles de macros :

1. macros de chargement arborescent des programmes
2. macros de gestion de la mémoire de données
3. macros de calcul et d'entrée-sortie
4. macros de contrôle structuré conditionnel et itératif
5. macros de lancement et de synchronisation des séquences
6. macros de transfert intermédia
7. macros de chronométrage

Chacune de ces familles fait l'objet d'une des sections suivantes.

4.4 Macros de chargement arborescent des programmes

Trois macros sont utilisées pour décrire les relations d'ascendance et de descendance de l'arbre de couverture du graphe de processeurs calculé par le générateur d'exécutif SynDEx, que deux autres macros utilisent, l'une avant l'exécution de l'algorithme d'application pour charger les programmes, et l'autre après l'exécution pour collecter les chronométrages effectués sur chaque processeur, si cette option de génération d'exécutif a été choisie.

Chaque processeur a un seul ascendant (sauf le processeur à la racine de l'arbre de couverture, qui n'en a pas, et qui est désigné par le concepteur de l'application en le nommant "root") et peut avoir plusieurs descendants (ou aucun pour les processeurs feuilles de l'arbre de couverture). Le processeur ascendant est désigné par la macro `tree_up_(link)` qui prend en argument un entier identifiant le média par l'intermédiaire duquel le processeur et son ascendant sont connectés (cet entier correspond au numéro d'ordre, dans la liste des ports déclarés par l'utilisateur pour ce processeur, du port connecté à ce média; pour le processeur "root", l'argument passé est `-1`). Chaque processeur descendant est désigné par une macro `tree_dn_(link)` qui prend en argument un entier identifiant le média par l'intermédiaire duquel le processeur et son descendant sont connectés. La fin de la liste des processeurs descendants (qui peut être vide pour un processeur feuille de l'arbre de couverture) est marquée par la macro `tree_dn_end`.

La macro `main_ini_` marque le point d'entrée du programme principal (c'est-à-dire en C de la fonction `main`) exécuté après le "boot" du processeur qui consiste à charger sa mémoire programme avec le code reçu de son processeur ascendant (ce code de "boot", exécuté en premier au démarrage du processeur, est forcément résidant sur chaque processeur). Cette macro doit générer le code permettant le chargement du programme de chaque descendant avec le code reçu toujours de l'ascendant (ce code de chargement doit être cohérent avec celui de "boot").

La macro `main_end_` marque la fin du programme principal et doit générer un code soit de retour à l'état initial (prêt pour un nouveau "boot"), soit d'arrêt du processeur (par exemple une instruction assembleur "idle", ou une boucle infinie)

Exemple de macros de chargement arborescent des programmes

Voici un exemple d'utilisation de ces macros (sans substitution car celle-ci est trop dépendante du type de processeur) pour un processeur qui serait connecté à son ascendant par son média 3, et à deux descendants par ses médias 0 et 2 :

```
tree_up_(3) tree_dn_(0) tree_dn_(2) tree_dn_end

main_ini_
... corps du programme principal
main_end_
```

Voici un exemple de définition des macros `main_ini_` et `main_end_` dans le cas d'une génération d'exécutif C monoprocesseur :

```
define(‘main_ini_’, ‘int main(int argc, char *argv[]){’)
define(‘main_end_’, ‘return 0;} /* main end */’)

main_ini_ comment{...}comment main_end_
| int main(int argc, char *argv[]){ /* ... */ return 0;} /* main end */
```

4.5 Macros de gestion de la mémoire de données

L'exécutif doit supporter l'allocation, l'initialisation et la copie de macro-registres, zones de mémoire utilisées pour stocker les données passées entre opérations de calcul, d'entrée/sortie et de communication. La même zone mémoire peut dans certains cas être allouée à des macro-registres différents, il faut donc que l'exécutif supporte aussi la réallocation (aussi appelée renommage ou "aliasing") de zones mémoire déjà allouées. Enfin, l'exécutif doit aussi supporter les "fenêtres glissantes" (macro-registre à décalage mémorisant, en plus du résultat de l'itération courante d'une opération, ses résultats des itérations précédentes), très utilisées pour le filtrage en traitement du signal.

L'allocation mémoire est statique, c'est-à-dire que l'adresse de chaque macro-registre est fixée à la compilation et reste invariante pendant l'exécution. Cette adresse est représentée par une étiquette symbolique, composée

par concaténation par un "underscore" du nom du sommet du graphe de l'algorithme et du nom du port de ce sommet à l'origine de la connexion que le macro-registre implante (chaque connexion est une dépendance de donnée qui a une origine unique, sommet producteur de la donnée, et une ou plusieurs extrémités, sommets consommateurs de la donnée).

On ne fait pas de distinction entre un type scalaire et les types tableaux dérivés: tous sont représentés par une adresse de base symbolique et par un nombre d'éléments (un seul pour le type scalaire, le produit des dimensions pour les tableaux à plus d'une dimension), donc on ne parlera par la suite plus que de tableaux.

Pour chaque type de donnée (il y a quatre types définis en SynDEx: `logical integer real dpreal`), il faut définir cinq macros pour:

- l'allocation
- le renommage
- l'initialisation
- la copie
- le décalage des fenêtres glissantes

4.5.1 Allocation mémoire

Chaque macro d'allocation mémoire doit être nommée comme le type des éléments du tableau, suffixé d'un caractère *underscore*, et doit prendre 2 ou 3 arguments:

1. une étiquette symbolique (unique, non encore utilisée) représentant l'adresse de base de la zone mémoire allouée
2. le nombre entier d'éléments à allouer contigument
3. s'il est présent, le troisième argument doit être un entier identifiant un espace mémoire particulier

La macro doit associer à l'étiquette la valeur du pointeur d'allocation de la zone mémoire données (celle désignée par le troisième argument ou une zone par défaut), puis déplacer ce pointeur d'allocation d'un nombre de cellules égal au produit du nombre d'éléments à allouer par le nombre de cellules mémoires nécessaire pour stocker un élément du type.

Exemple de macros d'allocation mémoire

Voici un exemple de définition des macros d'allocation mémoire générant du code C, avec un exemple d'appel et de code généré pour un tampon de type `[4,5]real` nommé `buf`:

```
define(‘logical_’, ‘int $1[$2];’)
define(‘integer_’, ‘int $1[$2];’)
define(‘real_’,    ‘float $1[$2];’)
define(‘dpreal_’,  ‘double $1[$2];’)

real_(buf, 20)
| float buf[20];
```

Dans le cas de macros générant du code assembleur, on utilisera les directives d'assemblage qui allouent une zone mémoire non initialisée.

4.5.2 Renommage mémoire

Chaque macro de renommage mémoire doit être nommée comme le type des éléments du tableau suffixé par `_alias_` et doit prendre 2 ou 3 arguments:

1. une nouvelle étiquette symbolisant l'adresse de base de la zone mémoire renommée
2. l'étiquette de base d'un macro-registre déjà alloué
3. l'indice désignant l'élément à ré-étiqueter dans le macro-registre déjà alloué, indice optionnel et nul par défaut

La macro doit associer à la nouvelle étiquette l'adresse de l'élément désigné.

Exemple de macros de renommage mémoire

Voici un exemple de définition des macros de renommage mémoire générant du code C :

```
define(‘cell_alias_’, ‘define(‘$1’, ifelse($3,,$2,($2+$3)))’)
define(‘logical_alias_’, ‘cell_alias_($@)’)
define(‘integer_alias_’, ‘cell_alias_($@)’)
define(‘real_alias_’,    ‘cell_alias_($@)’)
define(‘dpreal_alias_’,  ‘cell_alias_($@)’)

real_alias_(buf_last_,buf,19)
buf_last_[0]=3;
| (buf+19)[0]=3;
```

Les quatre macros ont le même comportement (factorisé par la macro `cell_alias_`) qui consiste à définir une macro qui remplacera chaque occurence du nom de la nouvelle étiquette soit par l'étiquette du macro-registre déjà alloué, si le troisième argument est absent, soit par la somme de l'étiquette du macro-registre déjà alloué et de l'indice (ce qui donne une adresse car, comme on l'a vu, toutes les étiquettes représentent des tableaux).

Dans le cas de macros générant de l'assembleur, on utilisera une directive d'équivalence entre la nouvelle étiquette et l'adresse correspondant à la somme de l'étiquette du macro-registre déjà alloué et du produit de l'indice par la taille (nombre de cellules mémoire) du type des éléments du tableau.

4.5.3 Initialisation mémoire

L'initialisation mémoire n'est nécessaire que pour les zones mémoires utilisées pour contenir soit une constante, soit une valeur calculée lors d'une itération précédente (retards et fenêtres glissantes). Les initialisations mémoire sont effectuées pendant la phase d'initialisation du programme principal de chaque processeur, avant les initialisations des entrées/sorties et avant le lancement des séquences de communication. Juste après la dernière macro d'initialisation mémoire, est générée une macro `data_ini_end` qui peut être utilisée pour marquer la fin d'une table de données d'initialisations.

Chaque macro d'initialisation mémoire doit être nommée comme le type des éléments, suffixé par `_ini_` et prend un nombre d'arguments variable en fonction du nombre d'éléments à initialiser :

1. la valeur littérale d'initialisation, obligatoire
2. une étiquette mémoire, obligatoire
3. l'indice (base zéro) du premier élément à initialiser, optionnel et nul par défaut
4. une liste d'itérateurs composés chacun de deux arguments dont le premier est le nombre d'itérations et le second l'incrément d'indice entre deux itérations, optionnel et égal à un par défaut pour le dernier itérateur

La macro doit initialiser un nombre d'éléments égal au produit des nombres d'itérations (égal à un si la liste d'itérateurs est vide). Chaque itération d'un itérateur est constituée d'abord soit, pour le dernier itérateur dans la liste des arguments, du stockage de la valeur d'initialisation dans l'élément d'indice courant, soit de l'ensemble des itérations de l'itérateur suivant dans la liste des arguments, puis d'une post-incrémentation de l'indice courant par l'incrément de l'itérateur. Par exemple, soit à initialiser le port `buf` (de l'exemple précédent d'allocation mémoire) par la constante SynDEx suivante :

```
[4,5]real ?’[{to 4,to 5}:0.0, {in 2 to 4, to 5 step 2}:1.0, [2,3]:1.5]’
```

Cette initialisation spécifie qu'après avoir mis tous les éléments à zéro, les éléments des colonnes impaires des trois dernières lignes doivent être initialisés à un, et enfin que l'élément de la troisième colonne de la seconde ligne doit être initialisé à un et demi :

1. pour la valeur d'initialisation `0.0` on pourrait écrire naïvement :
   `real_ini_(0.0, buf, 0, 4,0, 5,1)`
   mais on peut réduire les deux itérateurs en un seul :
   `real_ini_(0.0, buf, 0, 20)`
   car l'incrément d'indice entre deux lignes est nul, donc il suffit d'un seul itérateur, et l'incrément d'indice entre deux colonnes est 1, valeur par défaut

2. pour la valeur d'initialisation `1.0` on ne peut qu'écrire :
   `real_ini_(1.0, buf, 5, 3,-1, 3,2)`
   car le premier élément à initialiser est celui de la seconde ligne (indice 5) et 3 incréments de 2 réclament un incrément supplémentaire de -1 par ligne

3. pour la valeur d'initialisation `1.5` il n'y a qu'un seul élément à initialiser, le troisième de la seconde ligne, d'indice 7 :
   `real_ini_(1.5, buf, 7)`

Pour traiter leurs listes d'arguments itérateurs, les macros m4 d'initialisation mémoire sont définies de manière récursive.

Exemple de macros d'initialisation mémoire

Voici un exemple de définition de la macro d'initialisation `real_ini_` qui génére du code C, avec le code généré pour les trois appels de macros de l'exemple précédent :

```
define(‘real_ini_’, ‘ifelse($#,3,‘  $2[$3]=$1;’, ‘dnl $# = nb arguments
  {int i=$3;define(‘cell_init_’, ‘$2[i]=$1;’)
  cell_iter_(shift(shift(shift($@)))) dnl $@ = tous les arguments
  }’)’)

define(‘cell_iter_’, ‘{int n=$1; while(n--)ifelse(
  eval($#<=2),1,‘ cell_init_’, ‘
  cell_iter_(shift(shift($@)))’) dnl shift(arg1,autresArgs) -> autresArgs
  i+=ifelse($2,,1,$2);}’)

real_ini_(0.0, buf, 0, 20)
|  {int i=0;
|  {int n=20; while(n--) buf[i]=0.0;
|  i+=1;}
|  }

real_ini_(1.0, buf, 5, 3,-1, 3,2)
|  {int i=5;
|  {int n=3; while(n--)
|  {int n=3; while(n--) buf[i]=1.0;
|  i+=2;}
|  i+=-1;}
|  }

real_ini_(1.5, buf, 7)
|  buf[7]=1.5;
```

Dans le cas de macros générant du code assembleur, plutôt que de générer des instructions pour chaque initialisation, on économisera un espace mémoire non négligeable en ne stockant que les arguments de la macro, dans une zone mémoire initialisée (par exemple directement dans la zone code), avec un marqueur de fin de liste d'itérateurs pour chaque macro d'initialisation, et un marqueur de fin de liste d'initialisations, et on aura un seul code d'initialisation parcourant cette liste de listes jusqu'à trouver le marqueur de fin de liste d'initialisations. Chaque macro d'initialisation stocke d'abord l'adresse du premier élément à initialiser (afin que le marqueur de fin de liste d'initialisations soit simplement une adresse non valide, du genre 0 comme NULL), puis la valeur d'initialisation (précédée d'un identificateur de type si les types n'ont pas tous la même taille mémoire), puis un compte et un incrément pour chaque itérateur, puis un compte nul pour marquer la fin de la liste d'itérateurs. L'algorithme du code d'initialisation devra donc être le suivant :

– initialiser, sur le début de la zone mémoire contenant les données d'initialisation, un pointeur de lecture L (post-incrémenté après chaque lecture)

nextInit: lire dans le pointeur A l'adresse du premier élément à initialiser et terminer l'algorithme d'initialisations si cette adresse est nulle

readValue: lire dans le registre V la valeur d'initialisation (après avoir lu le code du type de cette valeur si les types n'ont pas tous la même taille mémoire)

- lire dans les registres C et I le compte et l'incrément du premier itérateur, et si C est non nul, passer à l'étape suivante, sinon stocker la valeur V à l'adresse pointée par A et reprendre en nextInit
- stocker la valeur courante du pointeur de lecture L dans B, base de la liste d'itérateurs

pushIter: empiler C et I puis lire dans C et I le compte et l'incrément de l'itérateur suivant et si C est non nul, recommencer l'étape courante, sinon passer à l'étape suivante

writeLoop: reculer le pointeur de lecture L d'un itérateur, dépiler C et I et répéter C fois :
écrire V à l'adresse pointée par A puis ajouter I à A

popIter: si L est égal à B, lire les itérateurs jusqu'à trouver celui de compte nul et reprendre en nextInit, sinon passer à l'étape suivante

- reculer le pointeur de lecture L d'un itérateur, dépiler C et I, décrémenter C et ajouter I à A, et si C est non nul, reprendre en pushIter, sinon reprendre en popIter

Si les types n'ont pas tous la même taille mémoire, alors l'étape writeLoop doit prendre en compte le type lu à l'étape readValue. Pour cela, les macros d'initialisation mémoire peuvent stocker le type de la valeur d'initialisation sous la forme d'une étiquette de sous-programme, lue avec la valeur d'initialisation à l'étape readValue et appelée dans l'étape writeLoop, afin que les instructions et les registres utilisés pour stocker la valeur d'initialisation puissent être différents pour chaque type.

4.5.4 Copie mémoire

Chaque macro de copie mémoire doit être nommée comme le type des éléments du tableau, suffixé par `_copy_` et doit prendre 3 arguments :

1. l'étiquette d'un macro-registre destination
2. l'étiquette d'un macro-registre source
3. le nombre d'éléments à copier de la source vers la destination

Exemple de macros de copie mémoire

Voici un exemple de définition des macros de copie mémoire générant du code C :

```
define(‘logical_copy_’, ‘memcpy($1,$2,$3*sizeof(int));’)
define(‘integer_copy_’, ‘memcpy($1,$2,$3*sizeof(int));’)
define(‘real_copy_’,    ‘memcpy($1,$2,$3*sizeof(float));’)
define(‘dpreal_copy_’,  ‘memcpy($1,$2,$3*sizeof(double));’)

real_copy_(dest,srce,5)
| memcpy(dest,srce,5*sizeof(float));
```

Dans le cas de macros générant du code assembleur, on distinguera le cas d'un seul élément transféré, ne requérant aucun code itératif, du cas général nécessitant un registre compteur et deux registres d'adresse pour supporter l'itération.

4.5.5 Fenêtres glissantes

En traitement du signal, on a souvent besoin de l'historique d'un signal, limité à ses n dernières valeurs des n dernières itérations *stimulus-calcul-réaction*. L'instruction `memory` de SynDEx permet de spécifier de telles *fenêtres glissantes*, appelées aussi *retards* lorsque seule la valeur la plus ancienne est référencée.

Une fenêtre glissante est un tableau de n macro-registres dont le premier, d'indice 0, est le plus ancien (celui de la $(n-1)$-ème dernière itération), et le dernier, d'indice $n-1$, est le plus récent (celui de l'itération courante). Chaque macro-registre est lui-même un tableau, dont la dimension d est le produit des dimensions

du type SynDEx utilisé pour le port d'entrée de l'instruction `memory` ($d = 1$ pour un type scalaire). Donc pour une fenêtre glissante, il faudra allouer un tableau de nd éléments de ce type.

Les fenêtres glissantes peuvent être implantées de deux manières :

- en décalant, de la taille d'un macro-registre, un indice de base, ce qui ne coûte qu'une mise à jour de cet indice de base, modulo n, mais nécessite, pour accéder aux macro-registres de la fenêtre glissante, un indiçage, également modulo n, relatif à l'indice de base
- en décalant, de la taille d'un macro-registre, le contenu du tableau, ce qui coûte un transfert de $n - 1$ macro-registres, mais ne nécessite, pour accéder aux macro-registres de la fenêtre glissante, qu'un indiçage absolu qui permet de traiter la fenêtre glissante comme un simple tableau

C'est la seconde implantation qui est utilisée actuellement. La première implantation donnera de meilleures performances, surtout pour des fenêtres glissantes de taille importante comme on peut en rencontrer par exemple en traitement d'image, mais elle nécessitera l'introduction d'une nouvelle syntaxe SynDEx pour distinguer les opérations (et plus particulièrement leurs ports d'entrée) qui utilisent des *tableaux circulaires*.

Chaque macro de décalage de fenêtre glissante doit être nommée comme le type des éléments du tableau, suffixé par `_window_` et doit prendre 3 arguments :

1. la taille d'un macro-registre de la fenêtre glissante, en nombre d'éléments du type de base
2. la taille de la fenêtre glissante, en nombre de macros-registres
3. l'étiquette de base de la fenêtre glissante

Exemple de macros de décalage de fenêtres glissantes

Voici un exemple de définition des macros de décalage pour fenêtres glissantes, générant du code C, ainsi qu'un exemple de code généré :

```
define(‘cell_window_’, ‘
  { int i; for(i=0; i<eval(($2-1)*$1); i++) $3[i]=$3[$1+i]; }’)
define(‘logical_window_’, ‘cell_window_($@)’)
define(‘integer_window_’, ‘cell_window_($@)’)
define(‘real_window_’,    ‘cell_window_($@)’)
define(‘dpreal_window_’,  ‘cell_window_($@)’)

real_window_(4,5,buf)
|  { int i; for(i=0; i<16; i++) buf[i]=buf[4+i]; }
```

Note : la fonction C `memcpy` ne peut pas être utilisée comme pour les macros de transfert mémoire parce que son comportement n'est pas garanti si les zones mémoire source et destination se recouvrent, ce qui est le cas des fenêtres glissantes.

4.6 Macros de calcul et d'entrée-sortie

Les macros de calcul et d'entrée-sortie sont spécifiques à l'application et donc spécifiées par le concepteur de l'application, alors que toutes les autres macros forment un *noyau générique d'exécutif*, jeu de macros indépendant de l'algorithme et de l'architecture de l'application, dont le codage est indépendant de l'algorithme mais dépend du type de processeur cible. Bien sûr, pour alléger le travail de spécification du concepteur d'application, il est souhaitable que le noyau générique d'exécutif soit fourni avec des macros de calcul et d'entrée-sortie de base, telles par exemple des macros de calcul logique (négation, conjonction, disjonction) et arithmétique (addition, soustraction, produit, division, comparaisons) sur les quatre types de données scalaires, ainsi que des macros de calcul et d'entrée-sortie du domaine applicatif, telles par exemple des macros de traitement du signal.

4.6.1 Macro-opérations de calcul

Pour chaque sommet opération de calcul du graphe de l'algorithme, le concepteur d'application spécifie un nom identifiant le sommet, un nom de macro à générer dans le code intermédiaire de l'exécutif, et une liste de

types et de noms de ports. Un appel à cette macro est généré dans le code intermédiaire du processeur qui a été alloué pour exécuter l'opération. Les arguments passés à cet appel de macro correspondent en nombre, position et type à la liste des ports de l'opération. Le nom de chaque argument est celui du tampon mémoire alloué pour le macro-registre implantant la connexion au port correspondant (cf chapitre 4.5).

Une macro-opération de calcul doit être purement combinatoire, sans aucun effet de bord : sa définition (substituée par le macroprocesseur à l'appel de la macro) doit générer une séquence d'instructions dont l'exécution ne doit faire aucun accès "caché" à des variables statiques globales ou locales (terminologie C), elle ne doit accéder qu'aux tampons mémoire dont les noms sont passés en argument de l'appel de la macro, pour y lire ses paramètres et y écrire ses résultats. Seuls les tampons correspondant aux ports de sortie (y compris ceux d'entrée et sortie) peuvent être accédés en écriture, les tampons correspondant aux ports d'entrée seule ne doivent être accédés qu'en lecture.

Ces contraintes sur le codage des définitions des macro-opérations de calcul sont nécessaires pour mettre en évidence les dépendances de données entre macro-opérations (en interdisant celles cachées par effet de bord), afin qu'on puisse générer pour chaque dépendance de données interprocesseur le code de la communication interprocesseur nécessaire au transfert de la donnée, et le code de la synchronisation entre producteur et consommateur(s) de la donnée. Grâce à cette synchronisation, le code intermédiaire généré garantit que, pendant l'exécution des instructions d'une macro-opération, ses tampons d'entrée resteront inchangés et que ses tampons de sorties ne seront accédés par aucune autre macro-opération.

La séquence d'instructions codant la combinatoire de la macro-opération peut être soit insérée en ligne, en substitution de l'appel de la macro, soit compilée séparément, auquel cas l'appel de la macro est substitué par une séquence d'instructions d'appel au code compilé séparément. Dans tous les cas, la durée d'exécution d'une macro-opération doit être bornée, sinon il sera impossible d'espérer respecter des contraintes temps réel.

Pour simplifier le codage des macros générant une séquence d'instructions d'appel à un code compilé séparément en C, le noyau générique d'exécutif est fourni avec des macros génériques de construction de l'appel. Rappelons qu'en C, les arguments sont passés :

- soit par valeur, pour les arguments d'entrée de type scalaire, avec les macros :
  `arg_logical_ arg_integer_ arg_real_ arg_dpreal_`
- soit par référence, pour les arguments de type tableau et pour les arguments de sortie (ou d'entrée et sortie) de type scalaire, avec la macro `arg_addr_`
- dans l'ordre inverse de la liste d'appel, avant l'emploi de la macro `call_args_` d'appel du point d'entrée du code compilé séparément

Exemples de macros de calcul

Voici quelques exemples de définitions de macros arithmétiques et logiques, générant en ligne des expressions C, ainsi qu'un exemple de code généré :

```
define('less','$3[0]=$1[0]<$2[0];')
define('add', '$3[0]=$1[0]+$2[0];')
define('addVec', '{ int i=$1; while(i--) $4[i]=$2[i]+$3[i]; }')

add(arg1, arg2, sum)
| sum[0]=arg1[0]+arg2[0];

addVec(10, v1, v2, vsum)
| { int i=10; while(i--) vsum[i]=v1[i]+v2[i]; }
```

Rappelons que tous les macro-registres sont des tableaux, y compris les "scalaires" qui sont des tableaux à un seul élément, ce qui explique l'indexation par `[0]` dans les exemples ci-dessus.

Voici un exemple de définition de macro pour une fonction C compilée séparément, ainsi que le code assembleur i80386/32bits généré :

```
# void fft(int order, float *inputArray, float *outputArray);
define('fft','arg_addr_($3)arg_addr_($2)arg_integer_($1)call_args_('fft')')

fft(order, timeSamples, freqSamples)
```

```
|  push OFFSET freqSamples
|  push OFFSET timeSamples
|  push DWORD PTR order
|  call _fft
|  add  SP,12 ; cleanup stack
```

4.6.2 Macro-opérations d'entrée-sortie

Un sommet source du graphe de l'algorithme, c'est-à-dire sans port d'entrée, représente un capteur, c'est une interface d'entrée de l'algorithme produisant des données obtenues par acquisition de l'état courant d'une variable de l'environnement. De même, un sommet puits du graphe de l'algorithme, c'est-à-dire sans port de sortie, représente un actionneur, c'est une interface de sortie de l'algorithme consommant des données représentant la valeur courante d'une variable de commande de l'environnement. Dans les deux cas, le sommet doit donc être contraint à être exécuté par le processeur attaché matériellement au capteur ou à l'actionneur. Cette contrainte permet de relâcher partiellement les contraintes citées au paragraphe précédent sur le codage des définitions des macro-opérations de calcul.

Une macro-opération d'entrée ou de sortie peut effectuer des effets de bord par l'intermédiaire de "variables statiques", qui sont le plus souvent des registres matériels d'entrée-sortie, mais qui peuvent aussi être des variables statiques locales (par exemple pour mémoriser un état de l'entrée-sortie), ou même globales (par exemple partagées avec une routine d'interruption, à condition que la macro-opération d'entrée-sortie soit contrainte à être exécutée sur le processeur qui exécute la routine d'interruption). Par contre, les dépendances de données entre macro-opérations de tous types doivent toujours impérativement s'effectuer par l'intermédiaire des tampons passés en argument des macros, les dépendances de données par effet de bord sont autant prohibées pour tous les types de macro-opérations.

Pour chaque sommet source ou puits du graphe de l'algorithme, le concepteur doit fournir un nom de macro (comme pour un sommet de calcul) et trois macros dont les noms auront pour même préfixe le nom de macro:

1. une macro d'initialisation, suffixée de `_ini_` et sans argument, censée générer tout code nécessaire d'initialisation (avant la première itération) de l'interface correspondant au sommet
2. une macro d'itération, suffixée de `_get_` pour un sommet source ou de `_put_` pour un sommet puits, prenant en argument la liste des ports du sommet (comme pour une macro de calcul)
3. une macro de finalisation, suffixée de `_end_` et sans argument, censée générer tout code nécessaire de finalisation (après la dernière itération) de l'interface correspondant au sommet

Exemples de macros d'entrée-sortie

Voici des exemples de définitions de macros d'entrée-sortie appelant des opérations d'entrée-sortie compilées séparément en C qui accèdent à des fichiers séquentiels simulant les flots d'entrée-sortie:

```
/* separately compiled C source */
#include <stdio.h>

/* (function gensig "calls" gensig "dt" 360 "i/o" real !sig) */
FILE *gensig_file_;
void gensig_ini_(void){ gensig_file_=fopen("gensig.dat","r"); }
void gensig_get_(float *sig){ fscanf(gensig_file_,"%e",sig); }
void gensig_end_(void){ fclose(gensig_file_); }

/* (function visu "calls" visu "dt" 360 "i/o" real ?er) */
FILE *visu_file_;
void visu_ini_(void){ visu_file_=fopen("visu.dat","w"); }
void visu_put_(float er){ fprintf(visu_file_,"%f\n",er); }
void visu_end_(void){ fclose(visu_file_); }

######## macros m4 ########
define(‘gensig_ini_’,‘call_args_(‘$0’)’)
define(‘gensig_get_’,‘arg_addr_($1)call_args_(‘$0’)’)
```

```
define(‘gensig_end_’,‘call_args_(‘$0’)’)
define(‘visu_ini_’,‘call_args_(‘$0’)’)
define(‘visu_put_’,‘arg_real_($1)call_args_(‘$0’)’)
define(‘visu_end_’,‘call_args_(‘$0’)’)

real_(gensig_sig,1)
| float gensig_sig[1];

gensig_ini_
| gensig_ini_();

gensig_get_(gensig_sig)
| gensig_get_(gensig_sig_);

visu_put_(gensig_sig);
| visu_put_(gensig_sig[0]);

visu_end_
| visu_end_();
```

4.7 Macros de contrôle conditionnel et itératif

L'exécutif doit supporter le séquencement des opérations de chacune des séquences de calcul ou de communication. En plus du séquencement implicite, dans l'ordre textuel du code généré, il faut coder explicitement le bouclage (sauts arrière) supportant l'itération (repliement/factorisation), le conditionnement (sauts avant), et le lancement des séquences de communication à partir de la séquence de calcul (le lancement des séquences de calcul, une seule sur chaque processeur, fait l'objet du paragraphe "Chargement des mémoires programme").

Pour les sauts avant, les noms des macros sont `if_ ifnot_ else_ endif_` et sont structurées de telle manière que chaque `if_` ou `ifnot_` soit balancé en aval soit par un `else_` soit par un `endif_`, et qu'un `else_` soit balancé en aval par un `endif_`. Pour les sauts arrière, nécessaires au contrôle itératif, les noms des macros sont `while_ endwhile_` et sont structurées de telle manière qu'un `while_` soit balancé en aval par un `endwhile_`[1].

Les macros `if_ ifnot_ while_ endwhile_` doivent prendre un argument, l'étiquette mémoire du macro-registre de type `logical` à tester, `else_` et `endif_` ne prennent aucun argument. La macro `endwhile_` prend en argument le même booléen que la macro `while_` correspondante, afin que le test du booléen et le saut conditionnel puissent être générés soit en début de boucle (par `while_` avec `endwhile_` générant en fin de boucle un saut inconditionnel en début de boucle), soit en fin de boucle (par `endwhile_` avec `while_` générant en début de boucle un saut inconditionnel sur le test en fin de boucle, ce qui économise un saut inconditionnel à chaque itération).

Les macros `hmul_ hadd_` sont générées pour les sommets d'intersection (conjonction) et d'union (disjonction) de booléens de conditionnement (les "horloges" du langage SIGNAL). Elles prennent en argument d'abord le nom du tampon booléen résultat, puis deux listes quotées de booléens correspondant chacune à une branche dans la hiérarchie des booléens de conditionnement.

Exemple de macros de séquencement

Voici un exemple de définition des macros de séquencement générant du code C, avec des exemples de code généré:

```
define(‘if_’, ‘if($1[0] != 0){’)
define(‘ifnot_’, ‘if($1[0] == 0){’)
define(‘else_’, ‘}else{’)
define(‘endif_’, ‘}’)
define(‘while_’, ‘while($1[0] != 0){’)
define(‘endwhile_’, ‘} /* end while */’)
define(‘h_term_’, ‘$1[0]ifelse($#,1,,‘‘&&’h_term_(shift($@))’)’)
```

1. les itérations dénombrées du genre boucles DO en FORTRAN ne sont pas supportées en version 4.

```
define(‘hmul_’, ‘$1[0]=(h_term_($2))&&(h_term_($3));’)
define(‘hadd_’, ‘$1[0]=(h_term_($2))||(h_term_($3));’)

if_(bool) /* bool true */ else_ /* bool false */ endif_
| if(bool[0] != 0){ /* bool true */ }else{ /* bool false */ }

while_(bool) /* loop body */ endwhile_(bool)
| while(bool[0] != 0){ /* loop body */ } /* end while */

hmul_(bool, ‘h1,h2’, ‘h3,h4’)
| bool[0]=(h1[0]&&h2[0])&&(h3[0]&&h4[0]);

hadd_(bool, ‘h1,h2’, ‘h3,h4’)
| bool[0]=(h1[0]&&h2[0])||(h3[0]&&h4[0]);
```

Dans le cas de macros générant du code assembleur, il faut générer des étiquettes de saut. Pour cela on utilise la capacité du macro-processeur `m4` à empiler des définitions de macros, ce qui permet de supporter l'imbrication des structures de contrôle. Voici un exemple de définition des macros de séquencement générant du pseudo assembleur, avec un exemple de code généré:

```
define(‘labelnumber’,0) dnl label counter for generating unique labels
define(‘pushlabel’,    ‘dnl make a new unique label based on $1
  pushdef(‘$1’, $1‘_’labelnumber‘_’)dnl
  define(‘labelnumber’,incr(labelnumber))’)
define(‘poplabel’, ‘popdef(‘$1’)’)

define(‘if_’, ‘pushlabel(‘forward_’)
  if($1[0] == 0) goto forward_;’)

define(‘ifnot_’, ‘pushlabel(‘forward_’)
  if($1[0] != 0) goto forward_;’)

define(‘else_’,
 ‘pushdef(‘swap_’, forward_)poplabel(‘forward_’)pushlabel(‘forward_’)
  goto forward_;
swap_: popdef(‘swap_’)’)

define(‘endif_’, ‘
forward_: poplabel(‘forward_’)’)

define(‘while_’, ‘pushlabel(‘backward_’)pushlabel(‘forward_’)
  goto forward_;
backward_:’)

define(‘endwhile_’, ‘
forward_:
  if($1[0]!=0) goto backward_; poplabel(‘forward_’)poplabel(‘backward_’)’)

define(‘h_term_’, ‘
  tmp = $1[0]; ifelse($#,1,,‘dnl
  if(tmp == 0) goto forward_; h_term_(shift($@))’)’)

dnl hmul_(int, ‘int,...’, ‘int,...’)
define(‘hmul_’, ‘{ register int tmp; pushlabel(‘forward_’)h_term_($2)dnl
  if(tmp == 0) goto forward_; h_term_($3)
forward_: poplabel(‘forward_’)
  $1[0] = tmp; }’)
```

```
dnl hadd_(int, 'int,...', 'int,...')
define('hadd_', '{ register int tmp; pushlabel('forward_')h_term_($2)dnl
pushdef('swap_', forward_)poplabel('forward_')pushlabel('forward_')dnl
  if(tmp != 0) goto forward_;
swap_: popdef('swap_')h_term_($3)
forward_: poplabel('forward_')
  $1[0] = tmp; }')

if_(bool) /* bool true */ else_ /* bool false */ endif_
|  if(bool[0] == 0) goto forward__0_; /* bool true */
|  goto forward__1_;
|forward__0_:  /* bool false */
|forward__1_:

while_(bool) /* loop body */ endwhile_(bool)
|  goto forward__3_;
|backward__2_: /* loop body */
|forward__3_:
|  if(bool[0] != 0) goto backward__2_;

hmul_(bool, 'h1,h2', 'h3,h4')
|{ register int tmp;
|  tmp = h1[0];   if(tmp == 0) goto forward__4_;
|  tmp = h2[0];   if(tmp == 0) goto forward__4_;
|  tmp = h3[0];   if(tmp == 0) goto forward__4_;
|  tmp = h4[0];
|forward__4_:
|  bool[0] = tmp; }

hadd_(bool, 'h1,h2', 'h3,h4')
|{ register int tmp;
|  tmp = h1[0];   if(tmp == 0) goto forward__5_;
|  tmp = h2[0];   if(tmp != 0) goto forward__6_;
|forward__5_:
|  tmp = h3[0];   if(tmp == 0) goto forward__6_;
|  tmp = h4[0];
|forward__6_:
|  bool[0] = tmp; }
```

4.8 Macros de lancement et de synchronisation

4.8.1 Macros de lancement des séquences de communication

Pour le lancement des séquences de communication, les noms des macros sont `spawn_ thread_` qui prennent chacune pour seul argument un entier identifiant la séquence à lancer en (pseudo)parallèle avec la séquence principale de calcul.

Pour la plupart des processeurs, chaque macro `spawn_` utilisée dans la phase d'initialisation de la séquence de calcul, génère une instruction `call` sautant à l'étiquette définie par ailleurs par la macro `thread_` au début de la séquence de communication correspondante. Le retour par une instruction `returnFromCall` à la séquence de calcul se fait dès que la séquence de communication exécute une macro-opération (de transfert ou de synchronisation) qui suspend son déroulement.

4.8.2 Macros de synchronisation inter-séquences

Comme pour la plupart des processeurs actuels la séquence de calcul et les séquences de communication exécutées sur un même processeur partagent l'unique séquenceur du processeur (cf chapitre 1.4.3), il faut un mécanisme d'arbitrage de l'allocation du séquenceur. Comme par ailleurs les liaisons de communication sont

généralement des ressources "lentes", il faut les utiliser au maximum, donc l'arbitrage doit allouer le séquenceur d'instructions à une séquence de communication prioritairement par rapport à la séquence d'opérations de calcul. Par contre, il ne faut pas qu'une séquence de communication "gaspille" inutilement le temps du séquenceur d'instructions.

Tous les opérateurs de communication sont conçus au niveau matériel pour être capables de requérir le séquenceur d'instructions en fin de transfert. Cette réquisition déclenche une sauvegarde automatique du contexte du séquenceur d'instructions (au moins son pointeur d'instruction) et initialise celui-ci pour exécuter le "programme d'interruption" associé à l'opérateur de communication (le mécanisme de sélection du programme d'interruption en fonction de la source de la requête varie d'un processeur à l'autre). Grâce à cela, il n'est pas nécessaire d'allouer le séquenceur d'instructions à une séquence de communication pendant ses périodes d'attente soit d'une fin de transfert, soit d'une synchronisation. Ainsi la séquence de calcul ne sera interrompue que juste le temps nécessaire soit à la programmation des registres de configuration de l'opérateur de communication, soit à l'exécution des étapes de synchronisation entre séquence de communication et séquence de calcul.

Dans les deux cas, les paramètres de l'étape de communication (identification de l'opérateur de communication et adresse et taille de la zone mémoire à transférer, ou bien adresse du sémaphore de synchronisation) sont déterminés lors de la compilation, donc inclus sous forme de constantes littérales dans le code de l'étape de communication. Ainsi, le contexte de la séquence de communication, à sauvegarder pendant ses périodes d'attente, est limité à l'adresse de la première instruction de l'étape de communication suivant celle en attente de complétion. Par ailleurs, le contexte de la séquence de calcul, à sauvegarder pendant ses interruptions par les séquences de communications, est limité aux quelques registres utilisés pendant les étapes de communication par les fonctions de synchronisation et de programmation des DMA, donc pendant la durée de l'interruption, il suffit de sauvegarder ces quelques registres sur la pile d'appel de sous-programmes de la séquence de calcul. Les coûts de ces changements de contexte réduits (de l'ordre d'une demi-douzaine d'instructions pour un TMS320C40) sont considérablement inférieurs à ceux des exécutifs multitâche classiques qui requièrent une sauvegarde et une restitution de tous les registres du processeurs lors d'un changement de contexte entre deux tâches (de l'ordre d'une centaine d'instructions pour un TMS320C40).

La dissymétrie d'arbitrage du séquenceur entre la séquence de calcul et celle de communication implique une dissymétrie de l'implantation des macros de synchronisation `Pre` et `Suc` pour chaque séquence. En effet, une étape d'attente `Suc`, si elle peut côté calcul attendre activement (c'est-à-dire en utilisant le séquenceur d'instructions, qui sera requis sur interruption par un événement de fin d'étape de transfert), doit côté communication attendre passivement (c'est-à-dire sans utiliser le séquenceur d'instructions, sinon la séquence de calcul n'aura jamais l'opportunité d'exécuter le `Pre` correspondant). Pour cette raison, on nommera par la suite `Suc0_` les macros d'attente côté calcul et `Suc1_` celles côté communication (0 et 1 pour "basse" et "haute" priorités d'arbitrage) et on nommera `Pre0_` et `Pre1_` les macros de synchronisation correspondant respectivement aux `Suc0_` et `Suc1_`.

4.8.3 Précédences communication(`Pre0_`)-calcul(`Suc0_`)

C'est le cas le plus simple. Comme la séquence de communication est exécutée sur interruption de la séquence de calcul, `Suc0_` côté calcul peut attendre en scrutant activement l'état de son sémaphore, car le processeur n'a rien d'autre de mieux à faire en attendant l'interruption de fin de transfert, qui requerra le séquenceur d'instructions pour exécuter dans la séquence de communication l'étape `Pre0_` attendue. Celle-ci n'a qu'à forcer dans l'état activé le sémaphore testé par `Suc0_` pour conditionner la répétition de sa boucle d'attente. Lorsque la séquence de communication exécute l'instruction de retour d'interruption (après avoir soit programmé les registres du DMA soit s'être mise en attente inactive par une étape `Suc1_`, cf sections suivantes), la séquence de calcul reprend l'exécution interrompue de `Suc0_`. Avant de se terminer, celle-ci doit remettre le sémaphore dans l'état inactivé afin que sa prochaine exécution le trouve dans son état initial.

Exemple de macros Pre0_/Suc0_

Voici un exemple de définition des macros de précédence avec attente active, générant du pseudo-assembleur :

```
define(‘semaphores_’, ‘dnl allocate $1 semaphores
  volatile static int sem_[$1] = {0}; /* all initially locked */’)

define(‘Pre0_’, ‘dnl signal $1-th semaphore
  sem_[$1] = 1;   /* unlock */’)

define(‘Suc0_’, ‘dnl wait on $1-th semaphore
```

```
  while(sem_[$1] == 0); /* until unlocked on interrupt */
  sem_[$1] = 0;   /* relock */')

semaphores_(12)
|  volatile static int sem_[12] = {0}; /* all initially locked */

Pre0_(7)
|  sem_[7] = 1;   /* unlock */

Suc0_(7)
|  while(sem_[7] == 0); /* until unlocked on interrupt */
|  sem_[7] = 0;   /* relock */
```

4.8.4 Précédences calcul(Pre1_)-communication(Suc1_)

Ce cas est plus complexe. Comme la séquence de communication est exécutée sur interruption de la séquence de calcul, Suc1_ côté communication ne peut attendre en scrutant activement l'état de son sémaphore, elle doit rendre (par retour d'interruption) le séquenceur à la séquence de calcul.

Dans le cas où la séquence de calcul exécute un Pre1_ avant que le Suc1_ correspondant soit exécuté par la séquence de communication, il suffit que Pre1_ change l'état du semaphore pour signaler son passage et que Suc1_ reconnaisse ensuite cet état pour continuer sans s'arrêter après avoir remis le sémaphore dans son état initial. Par contre, dans le cas où la séquence de communication exécute un Suc1_ avant que la séquence de calcul ait exécuté le Pre1_ correspondant, comme la séquence de communication est exécutée en interruption de la séquence de calcul, on ne peut se permettre une attente active comme pour les précédences Pre0_/Suc0_, donc Suc1_ doit sauvegarder le contexte de la séquence de communication et restituer celui de la séquence de calcul interrompue. Plus tard, l'exécution du Pre1_ correspondant "auto"-interromp la séquence de calcul en sauvegardant son contexte et en restituant celui de la séquence de communication. Comme les contextes des séquences sont alors limités à leur pointeur d'instruction et comme celui de la séquence de communication est connu à la compilation (l'adresse de l'instruction après Suc1_), pour changer de contexte il suffit d'une instruction d'appel de sous-programme dans Pre1_ et d'une instruction de retour de sous-programme dans Suc1_, dont l'exécution doit être conditionnée par l'état du sémaphore après sa modification.

Il est nécessaire que la lecture, le test et la modification du sémaphore soient effectués en exclusion mutuelle entre Pre1_ et Suc1_, donc interruption inhibées car l'exécution de Suc1_ peut être déclenchée par une interruption générée par le DMA en fin de transfert.

Exemple de macros Pre1_/Suc1_

Voici un exemple de définition de macros de précédence avec attente inactive, générant du pseudo-assembleur :

```
define(‘Pre1_’, ‘dnl signal $1-th semaphore
  if((sem_[$1] ^= 1) == 0) /* read/toggle/write/test must be indivisible */
    call(succ_$1_); /* global label declared by ‘Suc1_($1)’ */')

define(‘Suc1_’, ‘dnl wait on $1-th semaphore
  if((sem_[$1] ^= 1) == 1) /* read/toggle/write/test must be indivisible */
    returnFromCall; /* into DMAx_interrupt or after latest ‘Pre1_’ */
succ_$1_: /* global label for ‘Pre1_($1)’ */')
```

La condition d'exécution du `call` de Pre1_ étant l'opposée de celle du `returnFromCall` de Suc1_, ces deux macros peuvent être simplifiées de la manière suivante :

```
define(‘Pre1_’, ‘dnl signal $1-th semaphore
  call(succ_$1_); /* call global label declared by ‘Suc1_($1)’ */')

define(‘Suc1_’, ‘dnl wait on $1-th semaphore
succ_$1_: /* global label for ‘Pre1_($1)’ */
  if((sem_[$1] ^= 1) == 1) /* read/toggle/write/test must be indivisible */
    returnFromCall; /* into DMAx_interrupt or after latest ‘Pre1_’ */')
```

```
Pre1_(5)
|  call(succ_5_); /* call global label declared by Suc1_(5) */

Suc1_(5)
|succ_5_: /* global label for Pre1_(5) */
|  if((sem_[5] ^= 1) == 1) /* read/toggle/write/test must be indivisible */
|    returnFromCall; /* into DMAx_interrupt or after latest Pre1_ */
```

4.8.5 Précédences communication(Pre1_)-communication(Suc1_)

Dans le cas d'un routage d'une communication, les deux séquences de communication (l'une réceptrice et l'autre émettrice) se synchronisent directement sans passer par l'intermédiaire de la séquence de calcul. Comme les deux séquences de communication sont exécutées sous interruption, il faut alors utiliser le couple `Pre1_/Suc1_`.

4.8.6 Précédences calcul(Pre0_)-calcul(Suc0_)

Dans le cas de communication par mémoire partagée directement entre deux séquences de calcul disposant chacune de son propre séquenceur, chacune peut attendre "activement", donc il faut utiliser le couple `Pre0_/Suc0_`.

4.9 Macros de transfert inter-média

Le rôle de ces macros est de transférer une zone mémoire entre deux médias de communication, au besoin en changeant le format des données transférées si les formats des deux médias de communication diffèrent (par exemple dans le cas des Transputers, des TMS320C40 ou des SHARC, 32 bits coté bus mémoire et 1 bit coté lien de Transputer, ou 8 bits coté lien de TMS320C40, ou 4 bits coté lien du SHARC). Comme la taille de la zone mémoire à transférer (son nombre total de bits) est le plus souvent plus importante que la largeur des médias de communication (nombre de bits qu'ils sont capables de transférer simultanément), il est nécessaire de décomposer le transfert modulo la largeur du média de communication, donc d'effectuer une transformation spatio-temporelle dont la durée d'exécution augmente avec la taille de la zone mémoire à transférer. Les macros de transfert inter-média (une pour chaque sens de transfert) permettent d'encapsuler les détails d'implantation particuliers à chaque type de support matériel des transformatages.

Dans le cas des liens de Transputer, de TMS320C40 ou de SHARC, le plus gros du travail est supporté directement au niveau matériel. Comme il y a changement de format, c'est le média dont la bande passante (nombre de bits par seconde) est la plus faible qui détermine le rythme du transfert. Pour n'occuper qu'une fraction de la bande passante de l'autre média plus rapide, une mémoire intermédiaire est utilisée (mémoires FIFO sur TMS320C40, registre à décalage sur SHARC et Transputer). L'automate du DMA, programmable par l'intermédiaire de registres accessibles par le séquenceur d'instructions, séquence les micro-transferts entre le bus mémoire 32 bits et la mémoire intermédiaire. Un autre automate, synchronisé avec celui du DMA dans le rapport des bandes passantes des deux médias, séquence les micro-transferts entre la mémoire intermédiaire et le média lent.

Dans ces cas où le séquencement est supporté par DMA, les macros de transfert inter-média se réduisent à programmer les registres du DMA, à sauvegarder l'adresse de l'instruction qui suit l'appel de la macro de transfert, et à restaurer le contexte de la séquence de calcul. Le programme d'interruption de fin de transfert doit de son coté sauvegarder le contexte de la séquence de calcul (simplement sur la pile du `main`) et reprendre l'exécution de la séquence de communication à l'adresse sauvegardée.

Dans les cas où il n'y a pas de DMA pour automatiser les micro-transferts coté bus mémoire, comme par exemple dans le cas d'un micro-contrôleur pilotant une interface série du genre RS232, le séquencement des micro-transferts entre le bus mémoire et les registres intermédiaires de sérialisation doit être supporté par le séquenceur d'instructions, donc en interrompant la séquence de calcul à chaque micro-transfert. Les macros de transfert inter-média doivent alors sauvegarder les paramètres de la communication qu'elles reçoivent en arguments (adresse et taille de la zone mémoire à transferer) et armer un programme d'interruption qui, appelé à la fin de chaque micro-transfert, incrémente l'adresse courante, décrémente la taille restant à transférer et relance un micro-transfert, et ce jusqu'à ce que la taille restante devienne nulle, après quoi la séquence de

communication est reprise à l'instruction suivant l'appel de la macro de transfert inter-média, comme dans le cas avec DMA.

Dans une architecture hétérogène, il peut y avoir plusieurs types de transformateurs. Il est donc nécessaire de nommer les fonctions de gestion des transformateurs en les préfixant par le nom du type du transformateur. Pour chaque type de transformateur, il faut pouvoir :

- programmer les parties invariantes d'un transformateur (mises en place de vecteurs d'interruptions, démasquage des interruptions ...), nom de macro suffixé par `_ini_`
- programmer le transformateur dans un état inactif en fin d'exécution d'un programme distribué, nom de macro suffixé par `_end_`
- opérer un transfert en *émission*, c'est-à-dire d'une zone mémoire contigüe vers un lien, nom de macro infixé par `_o_` ("out") et suffixé par le nom du type de donnée (voir paragraphe suivant)
- opèrer un transfert en *réception*, c'est-à-dire d'un lien vers une zone mémoire contigüe, nom de macro infixé par `_i_` ("in") et suffixé par le nom du type de donnée
- ignorer un transfert, dans le cas de médias de communication partagés par plus de deux transformateurs, pour les étapes de communication où le transformateur n'est ni émetteur ni récepteur, nom de macro infixé par `_n_` ("nop") et suffixé par le nom du type de donnée

Dans une architecture hétérogène, chaque type de donnée peut avoir des représentations mémoire différentes pour les différents types de composants de l'architecture. Il est donc nécessaire de définir des macros de transfert inter-média différentes pour chaque type de donnée afin de supporter lors du transfert les conversions entre représentations mémoires différentes. Par contre, comme tous les éléments d'un tableau ont la même taille et sont contigus en mémoire, il suffit d'une seule macro de transfert inter-média pour tous les types tableau d'un même type scalaire, macro qui prend en argument le nombre total d'éléments à transférer (un seul pour le type scalaire lui-même, sinon le produit des dimensions du tableau).

Exemple de macros de transfert inter-média

Voici un exemple de définition de macros générant du pseudo-assembleur pour un DMA de transfert entre un média RAM (méroire partagée avec le processeur) et un média SAM de communication point-à-point (PPL pour "Processor to Processor Link", multipoint serait PBL pour "Processor to Bus Link") :

```
typedef struct{     /* typical DMA channel registers structure */
  int   control;    /* set to START_DMA_OUTPUT or to START_DMA_INPUT */
  int   counter;    /* transfer size in address units */
  char *address;    /* transfer start memory address */
  void(*suspend)(); /* address of instruction suspended during transfer */
} DMAchannels[NUMBER_OF_DMA_CHANNELS];
#define DMAchannel (DMAchannels)BASE_ADDRESS_OF_MEMORY_MAPPED_DMA_REGISTERS

define(‘PPL_ini_’, ‘dnl initialize $1-th communication sequence
DMA$1_interrupt:    /* typical DMA end-of-transfer interrupt routine */
  saveRegistersUsedDuringInterrupt();
  call(DMAchannel[$1].suspend); /* resume suspended instruction */
  restoreSavedRegisters();
  returnFromInterrupt;
DMA$1_sequence:     /* entry point of DMA$1 communication sequence */
  enableDMAinterrupt($1); /* and other DMA$1 initializations */’)

define(‘spawn_’, ‘dnl start communication sequence $1 in parallel with main
  call(DMA$1_sequence);   /* during main initializations */’)

define(‘PPL_o_integer’, ‘dnl thru link $1 send $2 integers from address $3
  DMAchannel[$1].counter = $2*sizeof(int); /* transfer size */
  DMAchannel[$1].address = (char*)$3; /* first source address */
  DMAchannel[$1].suspend = resume$3;  /* global label declared hereunder */
```

```
  DMAchannel[$1].control = START_DMA_OUTPUT;   /* activates DMA */
  returnFromCall; /* of DMA$1_interrupt or of 'Pre1_' */
resume$3: /* address of instruction to be resumed by DMA$1_interrupt */')

define('PPL_i_integer', 'dnl thru link $1 recv $2 integers from address $3
  DMAchannel[$1].counter = $2*sizeof(int); /* transfer size */
  DMAchannel[$1].address = (char*)$3; /* first destination address */
  DMAchannel[$1].suspend = resume$3;  /* global label declared hereunder */
  DMAchannel[$1].control = START_DMA_INPUT;    /* activates DMA */
  returnFromCall; /* of DMA$1_interrupt or of 'Pre1_' */
resume$3: /* address of instruction to be resumed by DMA$1_interrupt */')

define('PPL_end_', 'dnl terminate $1-th communication sequence
  disableDMAinterrupt(); /* and other DMA$1 finalizations */
  returnFromCall; /* of DMA$1_interrupt or of Pre1 */')
```

4.10 Macros de chronométrage

La génération des macros de chronométrage est une option paramétrable du générateur d'exécutifs. Le chronométrage consiste à mesurer les performances temps-réel de l'exécutif, il n'est pas nécessaire à son bon fonctionnement. Ces mesures permettent :

- de caractériser, pour chaque nouvelle architecture matérielle, les macro-opérations de synchronisation et de communication du noyau générique d'exécutif,
- de caractériser chaque nouvelle macro-opération utilisée dans de nouveaux algorithmes,
- de vérifier pour chaque implantation d'un algorithme sur une architecture, que les performances prédites par SynDEx, calculées à partir des caractéristiques mesurées, correspondent à la réalité.

Le chronométrage d'applications temps réel embarquées distribuées présente plusieurs difficultés :

- Chaque processeur ayant le plus souvent sa propre horloge, il faut considérer que la notion du temps est locale à chaque processeur et donc mesurer (par l'intermédiaire de communications interprocesseurs qui prennent du temps, problème classique de relativité) les décalages entre horloges pour pouvoir reconstituer une notion globale du temps.
- Les contraintes temps réel et d'embarquabilité incitent à minimiser les surcoûts de chronométrage, et donc à reporter en phase de finalisation, après la dernière itération temps réel, les surcoûts de mesures de décalages entre horloges et ceux de collecte des chronométrages, et pour limiter les surcoûts de mémorisation avant collecte, à ne conserver que les chronométrages des quelques dernières itérations.

C'est pourquoi le chronométrage s'effectue en quatre phases.

4.10.1 Initialisation

Une première phase d'initialisation du tampon circulaire de stockage des mesures est effectuée par la macro `Chrono_ini_` à la fin de la phase d'initialisations de la séquence de calculs.

L'allocation du tampon est faite par la macro `Chronos_` (appelée en fin de la liste des macros d'allocation mémoire) qui prend en argument un entier spécifiant le nombre de mesures que doit pouvoir mémoriser le tampon circulaire. Chaque mesure comprend deux entiers, l'un étant une étiquette identifiant le point de mesure (entre deux étapes de la séquence de calcul), l'autre étant une date lue sur l'horloge temps réel locale du processeur.

Pour le cas où le nombre de mesures effectuées serait inférieur à la capacité du tampon circulaire, chaque point de mesure est identifié avec une étiquette non nulle et on initialise le tampon avec des étiquettes nulles. Ainsi lors de la collecte, on pourra ignorer la partie du tampon qui contient des étiquettes nulles.

4.10.2 Mesure des dates

Une seconde phase, de mesure, indépendante sur chaque processeur, est effectuée pendant le déroulement de son programme. Des macros de chronométrage `Chrono_lap_` sont insérées entre les étapes de la séquence de calcul. Chaque point de mesure est identifié par une étiquette différente, passée en argument de la macro, qui est enregistrée, avec la date mesurée sur l'horloge temps-réel du processeur, dans un tampon circulaire dimensionné de manière à retenir les mesures des quelques dernières itérations (réactions) du programme.

La lecture d'une date sur l'horloge temps réel, ainsi que son stockage dans le tampon circulaire, ne sont pas gratuits : la durée d'exécution d'une macro opération de chronométrage doit être caractérisée (mesurée pour chaque type de processeur) pour pouvoir être retranchée de chaque intervalle entre deux dates si l'on désire avoir des mesures précises de durées des macro-opérations. Pour mesurer la durée d'exécution d'une macro-opération de chronométrage, il suffit d'en exécuter deux contigües, ou mieux plusieurs pour pouvoir calculer une durée moyenne et se faire un idée des variations s'il y en a.

Cette seconde phase de mesure des dates pose un problème délicat : il faut définir une condition d'arrêt pour passer de la seconde à la troisième phase. La difficulté consiste à arrêter tous les processeurs lors de la même itération afin qu'ils soient ensuite tous synchronisés pour les deux dernières phases de chronométrage. Pour les programmes sans itération, comme c'est le cas dans certains tests, il n'y a aucun problème. Pour les programmes avec itération, on peut :

- soit fixer à la compilation un nombre d'itérations commun à tous les processeurs, solution la plus simple utilisée actuellement, réalisée en conditionnant les macros de contrôle itératif `while_` et `endwhile_` par l'existence de la macro `NBITERATIONS` :
  - si la macro `NBITERATIONS` est définie, les macros `while_` et `endwhile_` génèrent, pour chaque séquence de calcul ou de communication, un code de contrôle imposant un nombre d'itérations fixé par la valeur de la macro `NBITERATIONS`,
  - sinon, les macros `while_` et `endwhile_` génèrent un code de contrôle itératif conditionnel dont la condition d'arrêt est fixée par la variable booléenne globale `_IOok_` initialisée à *vrai* et qui peut être forcée à $faux$ par les macro-opérations d'entrée-sortie; cette solution temporaire ne convient qu'en monoprocesseur, car elle ne permet d'arrêter qu'un seul processeur.
- soit laisser chaque processeur déterminer une condition d'arrêt, mais si l'on ne veut pas restreindre le parallélisme pipeline potentiel, il faut plusieurs itérations pour diffuser aux autres processeurs la condition d'arrêt, mais sur chaque processeur, le nombre d'itérations à effectuer entre la réception de la condition d'arrêt et le passage en troisième phase de chronométrage doit être diminué du nombre d'itérations nécessaire au transfert de la condition d'arrêt depuis le processeur qui l'a produite.

Le problème mérite d'être pris en compte au niveau de la spécification de l'algorithme (chose qui ne peut être faite dans le modèle actuel qui définit un signal comme une suite **sans fin** d'événements), sans quoi il semble difficile de prendre en compte, dans l'heuristique de distribution et d'ordonnancement, les communications diffusant la condition d'arrêt.

4.10.3 Mesure des décalages entre horloges

Une troisième phase est effectuée après la terminaison des calculs, par la première partie de la macro `Chrono_end_` générée en fin de phase de finalisation de chaque séquence de calcul.

Cette phase consiste à mesurer les décalages entre les horloges des processeurs, afin de rendre comparables entre elles les dates mesurées sur des processeurs différents. Le processeur hôte (à la racine de l'arbre de couverture du graphe d'interconnection des processeurs qui est utilisé pour le chargement arborescent des programmes, voir chapitre 4.4, puis récursivement chacun de ses descendants), effectue une communication aller-retour avec chacun de ses descendants dans l'arbre. En prenant la précaution pendant cette phase d'éviter toute interférence entre calculs et communications, on peut considérer que la date de renvoi du message, mesurée par le descendant, correspond à la moyenne des dates d'emission et de réception du message, mesurées par l'ascendant. Comme les branches de l'arbre sont séparées, ce processus de mesure des décalages entre les horloges des processeurs peut être effectué en parallèle dans chaque sous-arbre sans qu'il y ait d'interférence qui puisse perturber la précision de la mesure.

Soient :

- t le temps de l'horloge locale

- t_1 la date d'émission du message aller
- t_2 la date de réception du message retour
- t' le temps de l'horloge du processeur parent
- τ' la date de réception/renvoi sur le processeur parent

La date $\tau = (t_1 + t_2)/2$ correspond à τ', ce qui donne la relation $t - \tau = t' - \tau'$ d'où l'on tire $t' = t + (\tau' - \tau)$. Il faut donc ajouter $\Delta t = \tau' - \tau$ aux dates mesurées sur l'horloge locale pour les rendre comparables à celles mesurées sur l'horloge du processeur parent dans l'arbre de couverture du graphe des processeurs.

En pratique, les dates étant des entiers non signés, il faut utiliser l'arithmétique modulo [2] et calculer $\tau = t_1 + (t_2 - t_1)/2$, ce qui donne $\Delta t = \tau' - (t_1 + (t_2 - t_1)/2)$, ou encore, pour la commodité de l'implantation du calcul, $\Delta t = (t_2 - t_1)/2 - (t_1 - \tau')$.

On décompose donc cette phase en deux fonctions complémentaires. Côté parent, la fonction `ChronoSync` envoie un premier message pour signaler que le processeur est prêt à faire la mesure, puis attend de recevoir t_1 pour mesurer τ' et renvoyer immédiatement $t_1 - \tau'$. Côté descendant, la fonction `ChronoDiff` attend le message "prêt" de son parent, puis mesure t_1 qu'il envoie immédiatement, puis attend de recevoir $t_1 - \tau'$ pour mesurer t_2 qu'il retranche alors à t_1 et divise le résultat par deux et y retranche $t_1 - \tau'$ pour obtenir Δt qu'il retourne en résultat. Ces deux fonctions sont de préférence définies en assembleur pour leur donner le maximum d'efficacité afin que la précision des mesures soit la meilleure possible.

4.10.4 Collecte des chronométrages

La quatrième et dernière phase est effectuée après la terminaison des calculs en seconde partie de la macro `Chrono_end_` générée en fin de phase de finalisation de chaque séquence de calcul.

Cette phase consiste à collecter les résultats des chronométrages effectués en seconde phase sur chaque processeur. Les mesures sont collectées en remontant l'arbre de couverture du graphe d'interconnexion des processeurs qui est utilisé pour le chargement arborescent des programmes, voir chapitre 4.4. Chaque processeur envoie d'abord ses propres mesures à son ascendant dans l'arbre, sauf le hôte, à la racine de l'arbre, qui n'a pas d'ascendant, et qui stocke les résultats dans sa mémoire de masse, d'abord les siens puis ceux reçus de ses descendants. Puis chaque processeur retransmet à son ascendant les résultats qu'il reçoit de ses descendants. Lorsqu'il n'a plus de mesures à transmettre à son ascendant, il lui envoie un message "vide" (contenant une mesure avec l'étiquette nulle réservée à cet effet) qui permet au processeur ascendant de passer à son descendant suivant. La collecte se termine lorsque le hôte n'a plus de descendant suivant.

2. car le bit de poids fort du résultat de l'addition $t_1 + t_2$ est perdu et ne peut être correctement restitué par la division par deux, signée par défaut, alors que la durée $t_2 - t_1$ est correctement signée (sauf si sa magnitude est supérieure à la moitié de la dynamique du compteur de l'horloge, soit environ 200 secondes pour un timer 32 bits à 10 MHz comme celui du TMS320C40, cas que nous considérons exclus), donc la division par deux fournit un résultat correct qui peut être additionné à t_1 sans problème en utilisant l'arithmétique modulo.

Chapitre 5

Suite de tests pour mise au point d'un noyau générique d'exécutif

Ce chapitre décrit la démarche habituellement suivie pour tester et mettre au point progressivement les macros d'un nouveau noyau générique d'exécutif. Cette démarche se décompose en une série de tests permettant chacun de valider quelques macros supplémentaires, ou des contextes supplémentaires d'utilisation de ces macros.

Pour chaque macro, il est d'abord conseillé une expérimentation manuelle avec le macroprocesseur m4 en mode interactif, qui permet de cerner rapidement les erreurs évidentes de macro-substitution. Pour chaque test, il est ensuite conseillé, avant compilation et exécution, d'inspecter le code généré par substitution des macros par le macroprocesseur m4. Cette inspection devrait, si nécessaire, encourager à faire des efforts de formatage et de documentation en ligne des macros autant que du code qu'elles génèrent, afin d'améliorer la lisibilité du code généré, nécessaire à une bonne traçabilité des éventuelles erreurs, surtout si on les découvre plus tard.

L'expérience acquise au cours du développement de plusieurs noyaux génériques d'exécutifs pour des types variés de processeurs (de traitement du signal, des microcontrôleurs, des postes de travail) montre que le développement et le test d'un nouveau noyau est réalisable par un jeune programmeur en peu de temps (de quelques jours à quelques semaines) pourvu que soient disponibles une bonne documentation, décrivant le processeur et ses périphériques de communication et d'entrée-sortie, et un exemple de noyau développé pour un processeur semblable.

5.1 Test des macros de démarrage

Le tout premier test consiste à générer un programme minimum, monoprocesseur, du genre du classique "Hello world!", où la macro `printf_` peut être remplacée par toute autre macro effectuant un effet de bord observable montrant que le programme a été correctement chargé, lancé, exécuté, et terminé.

On ne teste pas ici le chargement arborescent des programmes de plusieurs processeurs, car la macro `tree_dn_` n'est pas utilisée dans ce test, mais les macros `tree_up_` et `tree_dn_end` sont nécessaires car elles génèrent un code (habituellement une structure de données simple du genre tableau) qui spécifie ici que la macro `main_ini_` n'a aucun chargement arborescent à effectuer.

On ne teste pas ici non plus de macro d'initialisation des tampons de données, mais la macro `data_ini_end` est nécessaire car elle génère un code qui spécifie pour la macro `main_ini_` la fin des initialisations des tampons de données.

```
comment{Test processor_ main_ini_ data_ini_end main_end_ end_}comment
processor_(root)
  tree_up_(-1)
  tree_dn_end

main_ini_
  data_ini_end
  printf_(‘Hello world!’)
main_end_

end_
```

Pour un noyau générique d'exécutif monoprocesseur générant du C, ce macrocode serait substitué par le code C suivant :

```
/* Test processor_ main_ini_ data_ini_end main_end_ end_ */
#include<stdio.h>

int main(int argc, char *argv[]){
  /* End of data init. */
  printf("Hello world!\n");
  return 0;
} /* end of main */
```

5.2 Test des macros d'allocation mémoire

Il s'agit ici de tester l'allocation de tableaux d'un (scalaire) ou plusieurs éléments, le renommage d'un élément (*type*_alias_), puis les initialisations d'un scalaire, d'une zone contigüe d'un tableau ou d'éléments régulièrement espacés d'un tableau, puis enfin les transferts entre tableaux et les fenêtres glissantes sur tableaux. La vérification du bon fonctionnement de ces macros peut se faire soit par inspection directe du source généré, soit en traçant son exécution à l'aide d'un débogueur.

Ce test peut être adapté pour tester les macros relatives à chacun des trois autre types de données.

```
comment{Test type_ type_alias_ type_ini_ type_copy_ type_window_}comment
processor_(root)
  tree_up_(-1)
  tree_dn_end

  logical_(_IOok_,1)             comment{un scalaire booleen}comment
  logical_(bools,8)              comment{tableau de 8 elements}comment
  logical_alias_(bool_6,bools,6) comment{bool_6==bools+6}comment

main_ini_
  logical_ini_(true,_IOok_,0)    comment{_IOok_[0]:=true}comment
  logical_ini_(false,bools,0,7)  comment{bools[0..7]:=false}comment
  logical_ini_(true,bools,1,2,3) comment{bools{1,4,7}:=true}comment
  data_ini_end   comment{false,true,f,f,t,f,f,t}comment
  logical_copy_(bools,_IOok_,1)  comment{bools[0]:=true}comment
  logical_copy_(bool_6,bools,2)  comment{[6..7]:=[0..2]}comment
  logical_window_(2,4,bools)     comment{[0..5]:=[2..7]}comment
  printf_('Ok')  comment{f,f,t,f,t,t,t,t}comment
main_end_

end_
```

Pour un noyau générique d'exécutif monoprocesseur générant du C, ce macrocode serait substitué par le code C suivant :

```
/* Test type_ type_alias_ type_ini_ type_copy_ type_window_ */
#include<stdio.h>

  int _IOok_[1];             /* un scalaire booleen */
  int bools[8];              /* tableau de 8 elements */
   /* bool_6==bools+6 */

int main(int argc, char *argv[]){
  _IOok_[0]=1;          /* _IOok_[0]:=true */
  {int i=0; {int n=7; while(n--){
  bools[i]=0; i+=1;}} } /* bools[0..7]:=false */
  {int i=1; {int n=2; while(n--){
  bools[i]=1; i+=3;}} } /* bools{1,4,7}:=true */
  /* End of data init. */   /* false,true,f,f,t,f,f,t */
  bools[0]=_IOok_[0];   /* bools[0]:=true */
  {int i=2; while(i--) (bools+6)[i]=bools[i];}   /* [6..7]:=[0..2] */
  {int i; for(i=0;i<6;i++) bools[i]=bools[2+i];} /* [0..5]:=[2..7] */
  printf("Ok\n");  /* f,f,t,f,t,t,t,t */
  return 0;
} /* end of main */
```

5.3 Test des macros de lancement des séquences de communication

Il s'agit ici de tester le lancement d'une séquence de communication (dont la priorité d'exécution est supérieure à celle de la séquence de calcul, donc qui interromp le déroulement de cette dernière) effectué par la macro `spawn_` en correspondance avec la macro `thread_`, puis la terminaison de la séquence de communication effectuée par la macro `PPL_end_` (qui reprend le déroulement de la séquence de calcul).

La macro `PPL_ini_` est nécessaire car elle effectue normalement une initialisation du média de communication, balancée par la finalisation effectuée par la macro `PPL_end_` (la finalisation effectue les opérations inverses de l'initialisation).

```
comment{Test spawn_ thread_ PPL_ini_ PPL_end_}comment
processor_(root)
  tree_up_(-1)
  tree_dn_end

thread_(0)
  PPL_ini_(0)
  printf_('2: thread(0)')
  PPL_end_(0)

main_ini_
  data_ini_end
  printf_('1: spawn(0)')
  spawn_(0)
  printf_('3: end.')
main_end_

end_
```

L'exécution du code obtenu, après substitution et compilation du macrocode de ce test, devrait afficher les messages suivants, dans cet ordre:

```
1: spawn(0)
2: thread(0)
3: end.
```

5.4 Test des macros de synchronisation

On vérifie d'abord le bon fonctionnement des quatres macros de synchronisation, indépendamment de toute communication. Chaque macro de synchronisation est testée dans les deux cas, "première arrivée" et "dernière arrivée", sauf pour `Suc0_` car la fin de son attente active ne peut être provoquée que sur interruption.

```
comment{Test Pre0_ Suc0_ Pre1_ Suc1_}comment
processor_(root)
  tree_up_(-1) tree_dn_end

  semaphores_(3)

thread_(0)
  PPL_ini_(0)
  printf_(‘3: Suc1(0); 2nd: continue’)
  Suc1_(0)    comment{derniere arrivee}comment
  printf_(‘4: Pre0(1); 1st: continue’)
  Pre0_(1)    comment{premiere arrivee}comment
  printf_(‘5: Suc1(2); 1st: switch to main’)
  Suc1_(2)    comment{premiere arrivee}comment
  printf_(‘8:        ; comm end’)
  PPL_end_(0)

main_ini_
  data_ini_end
  printf_(‘1: Pre1(0); 1st: continue’)
  Pre1_(0)    comment{premiere arrivee}comment
  printf_(‘2: spawn(0); start comm’)
  spawn_(0)
  printf_(‘6: Suc0(1); 2nd: continue’)
  Suc0_(1)    comment{derniere arrivee}comment
  printf_(‘7: Pre1(2); 2nd: switch to comm’)
  Pre1_(2)    comment{derniere arrivee}comment
  printf_(‘9:        ; main end.’)
main_end_
end_
```

L'exécution du code obtenu, après substitution et compilation du macrocode de ce test, devrait afficher les messages suivants, dans cet ordre:

```
1: Pre1(0); 1st: continue
2: spawn(0); start comm
3: Suc1(0); 2nd: continue
4: Pre0(1); 1st: continue
5: Suc1(2); 1st: switch to main
6: Suc0(1); 2nd: continue
7: Pre1(2); 2nd: switch to comm
8:        ; comm end
9:        ; main end.
```

5.5 Test monoprocesseur des macros de transfert intermédia

Ce test est supposé précédé de tests de programmation de l'interface matérielle de transfert, très spécifiques à cette interface matérielle, permettant de s'assurer que l'on sait utiliser correctement les bits de contrôle et d'état de l'interface matérielle, que l'on sait transférer correctement une zone mémoire de taille quelconque, et enfin que l'on sait provoquer et intercepter une interruption de fin de transfert.

Pour simplifier ces premiers tests, il faut éviter d'avoir à suivre le déroulement des événements à la fois sur les deux processeurs communiquant. Pour cela, il suffit de reboucler aussi simplement que possible une interface émettrice sur une interface réceptrice, soit en raccordant directement un port émetteur à un port récepteur, sans passer par l'intermédiaire d'un second processeur, si l'interface matérielle le permet ("loopback"), soit en programmant le second processeur au plus simple pour qu'il renvoie immédiatement tout ce qu'il reçoit. Dans le second cas, ces tests préliminaires de communication interprocesseur peuvent par la même occasion servir de test de la fonctionnalité "chargement arborescent des programmes" (macros `tree_up_ tree_dn_ main_ini_`).

Le présent test peut être adapté pour les autres types et d'autres tailles de données.

```
comment{Test PPL_o_integer PPL_i_integer}comment
processor_(root)
  tree_up_(-1) tree_dn_end

  integer_(ping,1)
  integer_(pong,1)
  semaphores_(2)

thread_(1) comment{link 1 is assumed to be connected to link 4}comment
  PPL_ini_(1)
  Suc1_(0)
  PPL_o_integer(1,1,ping) comment{send ping contents}comment
  PPL_end_(1)

thread_(4) comment{link 4 is assumed to be connected to link 1}comment
  PPL_ini_(4)
  PPL_i_integer(4,1,pong) comment{receive data into pong}comment
  Pre0_(1)
  PPL_end_(4)

main_ini_
  integer_ini_(12345678,ping) comment{setup ping}comment
  integer_ini_(0,pong)        comment{clear pong}comment
  data_ini_end
  printf_('1: spawn(1) -> PPL_ini(1) Suc1(0) ...')
  spawn_(1)
  printf_('2: spawn(4) -> PPL_ini(4) PPL_i_int(4,1,pong) ...')
  spawn_(4)
  printf_('3: Pre1(0) -> PPL_o_int(1,1,ping) ... PPL_end(1)')
  Pre1_(0)
  printf_('4: Suc0(1) ... PPL_i_int interrupt -> Pre0(1) PPL_end(4)')
  Suc0_(1)
  arg_integer_(pong) printf_('5: received: %d (12345678 expected).')
main_end_
end_
```

5.6 Test multiprocesseur des macros de transfert intermédia

On peut maintenant tester complètement un transfert de données aller et retour entre deux séquences de "calculs". Dans ce test, on suppose que le processeur `root` est connecté par son lien 0 au lien 3 du processeur `echo`.

Macro-exécutif du processeur root

Le processeur `root` commence par transférer le code du processeur `echo` (c'est la macro `main_ini_` qui s'en charge, à travers le lien 0 déclaré par la macro `tree_dn_`), puis envoie le contenu de son tampon `ping`, puis reçoit l'écho dans son tampon `pong` qu'il affiche pour contrôle.

```
processor_(root)
  tree_up_(-1)
  tree_dn_(0) comment{root link 0 connected to link 3 of echo}comment
  tree_dn_end

  integer_(ping,1)
  integer_(pong,1)
  semaphores_(2)

thread_(0)
  PPL_ini_(0)
  Suc1_(0)
  PPL_o_integer(0,1,ping)
  PPL_i_integer(0,1,pong)
  Pre0_(1)
  PPL_end_(0)

main_ini_
  integer_ini_(12345678,ping) comment{setup ping}comment
  integer_ini_(0,pong)        comment{clear pong}comment
  data_ini_end
  printf_('1: spawn(0) -> PPL_ini(0) Suc1(0) ...')
  spawn_(0)
  printf_('2: Pre1(0) -> PPL_o_int(0,1,ping) ... PPL_i_int(0,1,pong) ...')
  Pre1_(0)
  printf_('3: Suc0(1) ... PPL_i_int interrupt -> Pre0(1) PPL_end(0)')
  Suc0_(1)
  arg_integer_(pong) printf_('4: received: %d (12345678 expected).')
  printf_('Ok')
main_end_
end_
```

Macro-exécutif du processeur echo

Le processeur `echo` reçoit son code du processeur `root` (par son lien 3 déclaré par la macro `tree_up_`), puis attend la réception d'une donnée dans son tampon `ping`, qu'il copie dans son tampon `pong` (initialisés tous deux à zéro pour s'assurer qu'il y a bien eu transfert), puis envoie le contenu de son tampon `pong`, et enfin termine son exécution.

```
processor_(echo)
  tree_up_(3) comment{echo link 3 connected to link 0 of root}comment
  tree_dn_end

  integer_(ping,1)
  integer_(pong,1)
  semaphores_(2)

thread_(3)
  PPL_ini_(3)
  PPL_i_integer(3,1,ping)
  Pre0_(0)
  Suc1_(1)
  PPL_o_integer(3,1,pong)
  PPL_end_(3)

main_ini_
  integer_ini_(0,ping)
  integer_ini_(0,pong)
  data_ini_end
  spawn_(3)
  Suc0_(0)
  integer_copy_(pong,ping,1) comment{computation ersatz}comment
  Pre1_(1)
main_end_
end_
```

L'exécution du code obtenu, après substitution et compilation des macrocodes de ce test, devrait afficher les messages suivants, dans cet ordre:

```
1: spawn(0) -> PPL_ini(0) Suc1(0) ...
2: Pre1(0) -> PPL_o_int(0,1,ping) ... PPL_i_int(0,1,pong) ...
3: Suc0(1) ... PPL_i_int interrupt -> Pre0(1) PPL_end(0)
4: received: 12345678 (12345678 expected).
```

5.7 Tests des macros de chronométrage

On commence d'abord par un test monoprocesseur des fonctionnalités d'allocation (macro `Chronos_(n)`) et d'initialisation (macro `Chrono_ini_`) du tampon circulaire de stockage des mesures, et des fonctionnalités de mesure (macro `Chrono_lap_(label)`) et de stockage final (macro `Chrono_end_`) des dates, mais qui ne teste pas les fonctionnalités multiprocesseur de mesure des décalages entre horloges et de collecte arborescente des chronométrages.

Programme de test monoprocesseur

Dans ce test, une suite de macros `Chrono_lap_` contigües est utilisée afin de mesurer (assez précisément, par moyennage) la durée d'exécution parasite d'une mesure de date.

```
comment{Test Chonos_ Chrono_ini_ Chrono_lap_ Chrono_end_}comment
processor_(root)
  tree_up_(-1) tree_dn_end

  Chronos_(10)

main_ini_
  data_ini_end
  Chrono_ini_
  Chrono_lap_(1) comment{mesure des durees de chronometrage}comment
  Chrono_lap_(2)
  Chrono_lap_(3)
  Chrono_lap_(4)
  Chrono_lap_(5)
  Chrono_lap_(6)
  Chrono_lap_(7)
  Chrono_lap_(8) comment{chronometrage de l'operation printf}comment
  printf_('Ici Root.')
  Chrono_lap_(9)
  Chrono_end_
main_end_
end_
```

L'exécution du code obtenu, après substitution et compilation du macrocode de ce test, devrait afficher des messages analogues aux suivants, où les dates sont purement imaginaires, et les parenthèses sont des commentaires manuels :

```
label date duree=date-datePrec
  1    123   123     (duree de l'initialisation)
  2    133    10
  3    144    11
  4    154    10
  5    165    11     (duree moyenne d'un Chrono_lap_ = 10.5)
  6    175    10
  7    186    11
  8    196    10
  9   3600  3404     (duree du printf)
```

Pour tester la "récursivité" arborescente de la collecte, il faut avoir un arbre à au moins deux niveaux avec au moins deux processeurs à chaque niveau, ce qui ferait au moins 5 processeurs. Comme on n'en dispose que de 4 sur une carte mère TDMB410, on utilise l'arbre suivant à deux niveaux : P1-root-P2-P3.

Programme pour le processeur root

```
processor_(root) comment{Arbre: (root 0(3 P1) 2(4 P2 0(3 P3)))}comment
  tree_up_(-1) tree_dn_(0) tree_dn_(2) tree_dn_end

  integer_(buf,1)
  Chronos_(8)
  semaphores_(6)

thread_(0) comment{communications avec P1}comment
  PPL_ini_(0)
  Suc1_(0)               comment{buf vide}comment
  PPL_i_integer(0,1,buf) comment{1 de P1}comment
  Pre0_(1)               comment{buf plein}comment
  PPL_end_(0)

thread_(2) comment{communications avec P2}comment
  PPL_ini_(2)
  Suc1_(2)               comment{buf vide}comment
  PPL_i_integer(2,8,buf) comment{2 de P2}comment
  Pre0_(3)
  Suc1_(4)               comment{buf vide}comment
  PPL_i_integer(2,8,buf) comment{3 de P3 via P2}comment
  Pre0_(5)
  PPL_end_(2)

main_ini_ comment{root labels = 100+n}comment
  integer_ini_(0,buf) comment{buffer reception}comment
  spawn_(0)           comment{communications avec P1}comment
  spawn_(2)           comment{communications avec P2}comment
  Chrono_ini_
  Chrono_lap_(100)    comment{chronometrage de l'operation printf}comment
  printf_('Ici root')
  Chrono_lap_(101)    comment{date 101 - date 100 = duree printf}comment
  Pre1_(0)            comment{buf vide}comment
  Suc0_(1)            comment{attente reception 1 de P1}comment
  Chrono_lap_(102)    comment{date 102 - date 101 = duree attente}comment
  arg_integer_(buf) printf_('Ici P%d')
  Chrono_lap_(103)
  Pre1_(2)            comment{buf vide}comment
  Suc0_(3)            comment{attente reception 2 de P2}comment
  Chrono_lap_(104)
  arg_integer_(buf) printf_('Ici P%d')
  Chrono_lap_(105)
  Pre1_(4)            comment{buf vide}comment
  Suc0_(5)            comment{attente reception 3 de P3}comment
  Chrono_lap_(106)
  arg_integer_(buf) printf_('Ici P%d.')
  Chrono_lap_(107)
  Chrono_end_
main_end_
end_
```

Programme pour le processeur P1

```
processor_(P1) comment{Arbre: (root 0(3 P1) 2(4 P2 0(3 P3)))}comment
  tree_up_(3) tree_dn_end

  integer_(buf,1)
  Chronos_(2)
  semaphores_(2)

thread_(3)  comment{communications avec root}comment
  PPL_ini_(3)
  Suc1_(0)               comment{buf plein}comment
  PPL_o_integer(3,1,buf) comment{1 pour root}comment
  Pre0_(1)               comment{buf vide}comment
  PPL_end_(3)

main_ini_  comment{P1 labels = 200+n}comment
  integer_ini_(1,buf)
  data_ini_end
  spawn_(3)        comment{communications avec root}comment
  Chrono_ini_
  Chrono_lap_(200)
  Pre1_(0)         comment{buf plein}comment
  Suc0_(1)         comment{buf vide}comment
  Chrono_lap_(201)
  Chrono_end_
main_end_
end_
```

Programme pour le processeur P2

```
processor_(P2) comment{Arbre: (root 0(3 P1) 2(4 P2 0(3 P3)))}comment
  tree_up_(4) tree_dn_(0) tree_dn_end

  integer_(buf,1)
  Chronos_(4)
  semaphores_(6)

thread_(4)    comment{communications avec root}comment
  PPL_ini_(4)
  Suc1_(0)              comment{buf plein}comment
  PPL_o_integer(4,1,buf)comment{2 pour root}comment
  Pre0_(1)              comment{buf vide}comment
  Suc1_(4)              comment{buf plein}comment
  PPL_o_integer(4,1,buf)comment{3 de P3 pour root}comment
  Pre0_(5)              comment{buf vide}comment
  PPL_end_(4)

thread_(0)    comment{communications avec P3}comment
  PPL_ini_(0)
  Suc1_(2)              comment{buf vide}comment
  PPL_i_integer(0,1,buf)comment{3 de P3}comment
  Pre0_(3)              comment{buf plein}comment
  PPL_end_(0)

main_ini_     comment{P2 labels = 300+n}comment
  integer_ini_(2,buf)
  data_ini_end
  spawn_(4)        comment{communications avec root}comment
  spawn_(0)        comment{communications avec P3}comment
  Chrono_ini_
  Chrono_lap_(300)
  Pre1_(0)         comment{buf plein}comment
  Suc0_(1)         comment{buf vide}comment
  Chrono_lap_(301)
  Pre1_(2)         comment{buf vide}comment
  Suc0_(3)         comment{buf plein}comment
  Chrono_lap_(302)
  Pre1_(4)         comment{buf plein}comment
  Suc0_(5)         comment{buf vide}comment
  Chrono_lap_(303)
  Chrono_end_
main_end_
end_
```

Programme pour le processeur P3

```
processor_(P3) comment{Arbre: (root 0(3 P1) 2(4 P2 0(3 P3)))}comment
  tree_up_(3) tree_dn_end

  integer_(buf,1)
  Chronos_(2)
  semaphores_(2)

thread_(3)     comment{communications avec P2}comment
  PPL_ini_(3)
  Suc1_(0)               comment{buf plein}comment
  PPL_o_integer(3,1,buf) comment{3 pour root via P2}comment
  Pre0_(1)               comment{buf vide}comment
  PPL_end_(3)

main_ini_      comment{P3 labels = 400+n}comment
  integer_ini_(3,buf)
  data_ini_end
  spawn_(3)
  Chrono_ini_
  Chrono_lap_(400)
  Pre1_(0)         comment{buf plein}comment
  Suc0_(1)         comment{buf vide}comment
  Chrono_lap_(401)
  Chrono_end_
main_end_
end_
```

L'exécution du code obtenu, après substitution et compilation des macrocodes de ce test, devrait afficher des messages analogues aux suivants, où les dates sont purement imaginaires mais plausibles, et où les parenthèses sont des commentaires manuels :

```
Ici root
Ici P1
Ici P2
Ici P3.
label date duree=date-datePrec
 100   150   150     (root init)
 101  1000   850     (root duree printf 'Ici root')
 102  1100   100     (root attente reception 1 de P1)
 103  2000   900     (root duree printf 'Ici P1')
 104  2100   100     (root attente reception 2 de P2)
 105  3000   900     (root duree printf 'Ici P2')
 106  3100   100     (root attente reception 3 de P3)
 107  4000   900     (root duree printf 'Ici P3')
 200   150 -3850     (P1 init, duree reelle = 150)
 201  1050   900     (P1 attente fin emission 1 pour root)
 300   150  -900     (P2 init, duree reelle = 150)
 301  2050  1900     (P2 attente fin emission 2 pour root)
 302  2150   100     (P2 attente reception 3 de P3)
 303  3050   900     (P2 attente fin emission 3 pour root)
 400   150 -2900     (P3 init, duree reelle = 150)
 401  2100  1950     (P3 attente fin emission 3 pour root via P2)
```

5.8 Test pour l'application EGA monoprocesseur

L'application "EGA" est un exemple simple de traitement du signal (un algorithme d'égalisation adaptative), de petite taille mais complet, qui se prête bien aux démonstrations et aux tests de génération d'exécutifs. Cette application est livrée avec le logiciel SynDEx, accompagnée de plusieurs implantations (mono et biprocesseur, pour différents types de processeurs), avec tous les fichiers sources nécessaires à leurs réalisation.

Sont reproduits ici les sources m4 du macrocode généré pour :

- une implantation monoprocesseur, où certains commentaires ont été ajoutés dans un but didactique
- une implantation biprocesseur "EGA2C" sans chronométrage
- une implantation biprocesseur "EGA2C" avec macros de chronométrage

Ces sources permettront au lecteur de se faire une idée plus concrète du macrocode généré pour une application complète avec entrées/sorties, calculs et communications interprocesseurs, répétés itérativement.

Fichier source m4 pour mono-processeur

```
comment{SynDEx v4.3 INRIA 1998/7/23-1:42:15}comment
comment{application ega: real equalizer}comment
comment{ega1.m4 source file for monoprocessor}comment

processor_(root)
  tree_up_(-1)
  tree_dn_end

comment{constants}comment
  real_(gain_1,1) comment{scalar constant}comment
  real_(filt_1,9) comment{vector constant [{to 9}:0.0, [5]:1.0]}comment
comment{initialized data (memories)}comment
  integer_(_IOok_,1) comment{I/O correct flag}comment
  real_(adap_coeff,9) comment{vector delay init [{to 9}:0.0]}comment
  real_(wind_o,9) comment{slidding window init [{to 8}:0.0]}comment
  real_alias_(gensig_sig,wind_o,8) comment{last/newest wind_o item}comment
comment{uninitialized data (temporary buffers)}comment
  real_(filt_o,1)
  real_alias_(gain_o,filt_o,0) comment{temporary buffer used twice}comment
  real_(filta_o,1)
  real_(sub_er,1)

main_ini_
  integer_ini_(1,_IOok_)
  real_ini_(0.1,gain_1)
  real_ini_(0.0,filt_1,0,9)
  real_ini_(1.0,filt_1,4)
  real_ini_(0.0,adap_coeff,0,9)
  real_ini_(0.0,wind_o,0,8)
  data_ini_end
  gensig_ini_()
  visu_ini_()
while_(_IOok_)  comment{_____main_loop_start_____}comment
  gensig_get_(gensig_sig)
  realDotProduct(9, filt_1, wind_o, filt_o)
  realDotProduct(9, adap_coeff, wind_o, filta_o)
  realSub(filta_o, filt_o, sub_er)
  visu_put_(sub_er)
  realMul(sub_er, gain_1, gain_o)
  realEqualizer(9, gain_o, wind_o, adap_coeff)
  real_window_(1,9, wind_o)
endwhile_(_IOok_)  comment{_____main_loop_end_____}comment
  gensig_end_()
  visu_end_()
main_end_
end_
```

Fichier source des macros d'entrée-sortie définies en C

```
divert(-1)
dnl SynDEx v4 INRIA
dnl application ega2c: real equalizer on TMS320C40
dnl ega2c.m4h source file for I/O macros

divert(1)dnl
#include<stdio.h>
divert(-1)

dnl extern int _IOok_[1];

divert(1)dnl
FILE *gensig_file_;
divert(-1)
def(‘gensig_ini_’,
 ‘if(_IOok_[0]) _IOok_[0]=((gensig_file_=fopen("gensig.dat","r"))!=NULL);’)
def(‘gensig_end_’,
 ‘fclose(gensig_file_);’)
def(‘gensig_get_’, dnl args:(float *sig)
 ‘if(_IOok_[0]) _IOok_[0]=((fscanf(gensig_file_, "%e", $1)!= EOF));’)

divert(1)dnl
FILE *visu_file_;
divert(-1)
def(‘visu_ini_’,
 ‘if(_IOok_[0]) _IOok_[0]=((visu_file_=fopen("visu.dat","w"))!=NULL);’)
def(‘visu_end_’,
 ‘fclose(visu_file_);’)
def(‘visu_put_’, dnl args:(float er)
 ‘if(_IOok_[0]) fprintf(visu_file_, "%f\n", $1[0]);’)
divert‘’dnl
```

Toutes les autres macros font partie du noyau générique d'exécutif, y compris les macros `realSub` et `realMul` qui font partie des macros standard de calcul arithmétique et logique, et les macros `realDotProduct` et `realEqualizer` qui font partie des macros standard de calcul de traitement du signal, dont voici les définitions dans le cas présent d'une génération de code C.

```
dnl realSub(s1,s2,dst) ; dst=s1-s2
define(‘realSub’, ‘$3[0]=$1[0]-$2[0];’)

dnl realMul(s1,s2,dst) ; dst=s1*s2
define(‘realMul’, ‘$3[0]=$1[0]*$2[0];’)

dnl realDotProduct(size,i1,i2,res) ; res=sum[i=0..size-1]{i1[i]*i2[i]}
def(‘realDotProduct’, dnl size=$1, i1[$1]@$2, i2[$1]@$3, res[1]@$4
 ‘{ int i=$1[0]; float accu=0.0;
    while(i--) accu+=$2[i]*$3[i]; $4[0]=accu; }’)

dnl realEqualizer(size,err,win,coeff) ; coeff[size]-=win[size]*err
def(‘realEqualizer’, dnl size=$1, err[1]@$2, win[$1]@$3, coeff[$1]@$4
 ‘{ int i=$1[0]; while(i--) $4[i]-=$3[i]*$2[0]; }’)
```

Fichier source C monoprocesseur obtenu après substitution

```
/* SynDEx v4.3 INRIA 1998/7/23-1:42:15 */
/* application ega: real equalizer */
/* ega1.m4 source file for monoprocessor */
#include<stdio.h>
FILE *gensig_file_;
FILE *visu_file_;

/* constants */
  float gain_1[1]; /* scalar constant */
  float filt_1[9]; /* vector constant [{to 9}:0.0, [5]:1.0] */
/* initialized data (memories) */
  int _IOok_[1]; /* I/O correct flag */
  float adap_coeff[9]; /* vector delay init [{to 9}:0.0] */
  float wind_o[9]; /* slidding window init [{to 8}:0.0] */
   /* last/newest wind_o item */
/* uninitialized data (temporary buffers) */
  float filt_o[1];
   /* temporary buffer used twice */
  float filta_o[1];
  float sub_er[1];

int main(int argc, char *argv[]){
  _IOok_[0]=1;
  gain_1[0]=0.1;
  {int i=0; {int n=9; while(n--){
  filt_1[i]=0.0; i+=1;}} }
  filt_1[4]=1.0;
  {int i=0; {int n=9; while(n--){
  adap_coeff[i]=0.0; i+=1;}} }
  {int i=0; {int n=8; while(n--){
  wind_o[i]=0.0; i+=1;}} }
  /* End of data initializations. */
  if(_IOok_[0]) _IOok_[0]=((gensig_file_=fopen("gensig.dat","r"))!=NULL);
  if(_IOok_[0]) _IOok_[0]=((visu_file_=fopen("visu.dat","w"))!=NULL);
while(_IOok_[0]){  /* _____main_loop_start_____ */
  if(_IOok_[0]) _IOok_[0]=((fscanf(gensig_file_, "%e", (wind_o+8))!= EOF));
  { int i=9[0]; float accu=0.0;
  while(i--) accu+=filt_1[i]*wind_o[i]; filt_o[0]=accu; }
  { int i=9[0]; float accu=0.0;
  while(i--) accu+=adap_coeff[i]*wind_o[i]; filta_o[0]=accu; }
  sub_er[0]=filta_o[0]-filt_o[0];
  if(_IOok_[0]) fprintf(visu_file_, "%f\n", sub_er[0]);
  (filt_o+0)[0]=sub_er[0]*gain_1[0];
  { int i=9[0]; while(i--) adap_coeff[i]-=wind_o[i]*(filt_o+0)[0]; }
  { int i; for(i=0; i<8; i++) wind_o[i]=wind_o[1+i]; }
} /* end while */  /* _____main_loop_end_____ */
  fclose(gensig_file_);
  fclose(visu_file_);
  return 0;
} /* end of main */
```

5.9 Test pour l'application EGA2C biprocesseur

Fichier source m4 pour le processeur root

```
comment{SynDEx v4.3 INRIA 1998/7/23-1:42:30}comment
comment{application ega2c: real equalizer on 2 TMS320C40}comment
comment{ega2c1.m4 source file for processor root}comment

processor_(root) comment{Arbre: (root 0(3 P))}comment
  tree_up_(-1) tree_dn_(0) tree_dn_end

  integer_(_IOok_,1)
  real_(filt_1,9)
  real_(wind_o,9)
  real_alias_(gensig_sig,wind_o,8)
  real_(filt_o,1)
  real_(sub_er,1)
  semaphores_(3)

thread_(0)
  PPL_ini_(0)
while_(_IOok_)
  Suc1_(0)
  PPL_o_real(0, 1, gensig_sig)
  PPL_o_integer(0, 1, _IOok)
  Suc1_(1)
  PPL_o_real(0, 1, filt_o)
  PPL_i_real(0, 1, sub_er)
  Pre0_(2)
endwhile_(_IOok_)
  PPL_end_(0)

main_ini_
  integer_ini_(1,_IOok_)
  real_ini_(0.0,wind_o,0,8)
  real_ini_(0.0,filt_1,0,9)
  real_ini_(1.0,filt_1,4)
  data_ini_end
  gensig_ini_()
  visu_ini_()
  spawn_(0)
while_(_IOok_)
  gensig_get_(gensig_sig)
  Pre1_(0)
  realDotProduct(9, filt_1, wind_o, filt_o)
  Pre1_(1)
  real_window_(1,9, wind_o)
  Suc0_(2)
  visu_put_(sub_er)
endwhile_(_IOok_)
  gensig_end_()
  visu_end_()
main_end_
end_
```

Fichier source m4 pour le processeur P

```
comment{SynDEx v4.3 INRIA 1998/7/23-1:42:30}comment
comment{application ega2c: real equalizer on 2 TMS320C40}comment
comment{ega2c2.m4 source file for processor P}comment

processor_(P) comment{Arbre: (root 0(3 P))}comment
  tree_up_(-1) tree_dn_(0) tree_dn_end

  integer_(_IOok_,1)
  real_(gain_1,1)
  real_(adap_coeff,9)
  real_(winda_o,9)
  real_alias_(gensig_sig,wind_o,8)
  real_(filt_o,1)
  real_alias_(gain_o,filt_o,0)
  real_(filta_o,1)
  real_(sub_er,1)

  semaphores_(3)

thread_(3)
  PPL_ini_(3)
while_(_IOok_)
  PPL_i_real(3, 1, gensig_sig)
  Pre0_(0)
  PPL_i_integer(3, 1, _IOok)
  PPL_i_real(3, 1, filt_o)
  Pre0_(1)
  Suc1_(2)
  PPL_o_real(3, 1, sub_er)
endwhile_(_IOok_)
  PPL_end_(3)

main_ini_
  integer_ini_(_IOok_,1)
  real_ini_(0.1,gain_1)
  real_ini_(0.0,winda_o,0,8)
  real_ini_(0.0,adap_coeff,0,9)
  data_ini_end
  spawn_(3)
while_(_IOok_)
  Suc0_(0)
  realDotProduct(9, adap_coeff, winda_o, filta_o)
  Suc0_(1)
  realSub(filta_o, filt_o, sub_er)
  Pre1_(2)
  realMul(sub_er, gain_1, gain_o)
  realEqualizer(9, gain_o, winda_o, adap_coeff)
  real_window_(1,9, winda_o)
endwhile_(_IOok_)
  main_end_
end_
```

5.10 Test pour application EGA2C avec chronométrage

Fichier source m4 pour le processeur root

```
comment{SynDEx v4.3 INRIA 1998/7/23-1:42:45}comment
comment{application ega2c: real equalizer on 2 TMS320C40}comment
comment{ega2c1.m4 source file for processor root}comment

processor_(root) comment{Arbre: (root 0(3 P))}comment
  tree_up_(-1) tree_dn_(0) tree_dn_end

  integer_(_IOok_,1)
  real_(filt_1,9)
  real_(wind_o,9)
  real_alias_(gensig_sig,wind_o,8)
  real_(filt_o,1)
  real_(sub_er,1)

  Chronos_(40)
  semaphores_(3)

thread_(0)
  PPL_ini_(0)
while_(_IOok_)
  Suc1_(0)
  PPL_o_real(0, 1, gensig_sig)
  PPL_o_integer(0, 1, _IOok_)
  Suc1_(1)
  PPL_o_real(0, 1, filt_o)
  PPL_i_real(0, 1, sub_er)
  Pre0_(2)
endwhile_(_IOok_)
  PPL_end_(0)

main_ini_
  integer_ini_(1,_IOok_)
  real_ini_(0.0,wind_o,0,8)
  real_ini_(0.0,filt_1,0,9)
  real_ini_(1.0,filt_1,4)
  data_ini_end
  Chrono_ini_
  gensig_ini_()
  visu_ini_()
  spawn_(0)
while_(_IOok_)
  Chrono_lap(100)
  gensig_get_(gensig_sig)
  Chrono_lap(101)
  Pre1_(0)
  Chrono_lap(102)
  realDotProduct(9, filt_1, wind_o, filt_o)
  Chrono_lap(103)
  Pre1_(1)
  Chrono_lap(104)
  real_window_(1,9, wind_o)
  Chrono_lap(105)
  Suc0_(2)
```

```
  Chrono_lap(106)
  visu_put_(sub_er)
  Chrono_lap(107)
endwhile_(_IOok_)
  gensig_end_()
  visu_end_()
  Chrono_end_
main_end_
end_
```

Fichier source m4 pour le processeur P

```
comment{SynDEx v4.3 INRIA 1998/7/23-1:42:45}comment
comment{application ega2c: real equalizer on 2 TMS320C40}comment
comment{ega2c2.m4 source file for processor P}comment

processor_(P) comment{Arbre: (root 0(3 P))}comment
  tree_up_(-1) tree_dn_(0) tree_dn_end

  integer_(_IOok_,1)
  real_(gain_1,1)
  real_(adap_coeff,9)
  real_(winda_o,9)
  real_alias_(gensig_sig,wind_o,8)
  real_(filt_o,1)
  real_alias_(gain_o,filt_o,0)
  real_(filta_o,1)
  real_(sub_er,1)

  Chronos_(45)
  semaphores_(3)

thread_(3)
  PPL_ini_(3)
while_(_IOok_)
  PPL_i_real(3, 1, gensig_sig)
  Pre0_(0)
  PPL_i_integer(3, 1, _IOok)
  PPL_i_real(3, 1, filt_o)
  Pre0_(1)
  Suc1_(2)
  PPL_o_real(3, 1, sub_er)
endwhile_(_IOok_)
  PPL_end_(3)

main_ini_
  integer_ini_(1,_IOok_)
  real_ini_(0.1,gain_1)
  real_ini_(0.0,winda_o,0,8)
  real_ini_(0.0,adap_coeff,0,9)
  data_ini_end
  Chrono_ini_
  spawn_(3)
while_(_IOok_)
  Chrono_lap(200)
  Suc0_(0)
  Chrono_lap(201)
  realDotProduct(9, adap_coeff, winda_o, filta_o)
  Chrono_lap(202)
  Suc0_(1)
  Chrono_lap(203)
  realSub(filta_o, filt_o, sub_er)
  Chrono_lap(204)
  Pre1_(2)
  Chrono_lap(205)
  realMul(sub_er, gain_1, gain_o)
  Chrono_lap(206)
```

```
  realEqualizer(9, gain_o, winda_o, adap_coeff)
  Chrono_lap(207)
  real_window_(1,9, winda_o)
  Chrono_lap(208)
endwhile_(_IOok_)
  Chrono_end_
main_end_
end_
```

Fichier de commentaire des chronométrages

Ce fichier permet de faire l'association entre les étiquettes numériques des chronométrages et le début de l'étape correspondante du `main` (qui est également la fin de l'étape précédente). Les communications interprocesseurs sont désignées par le nom du port de sortie produisant les données, préfixé par un point d'exclamation pour les communications sortantes, ou par un point d'interrogation pour les communications entrantes.

```
root
 100  gensig
 101 !gensig_sig
 102  filt
 103 !filt_o
 104  wind
 105 ?sub_err
 106  visu
 107  main_loop_
P
 200 ?gensig_sig
 201  filta
 202 ?filt_o
 203  sub
 204 !sub_err
 205  mul
 206  ega
 207  winda
 208  main_loop_
```

Bibliographie

[1] D. Harel, A. Pnueli. *On the development of reactive systems.* In K. R. Apt, editor, Logics and Models of Concurrent Systems, Springer Verlag, New York, 1985.

[2] Paul Feautrier (http://www.prism.uvsq.fr/ paf). *Dataflow Analysis of Scalar and Array References.* Int. J. of Parallel Programming, 20(1):23–53, February 1991.

[3] 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.

[4] F. Ennesser, C. Lavarenne, Y. Sorel. *Méthode chronométrique pour l'optimisation du temps de réponse des exécutifs SynDEx.* Rapports de Recherche INRIA 1769, 1992.

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

[6] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

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

[8] A. Benveniste, G. Berry. *the synchronous approach to reactive and real-time systems.* Proceedings of the IEEE, 79(9):1270-1282, Sep., 1991.

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

[10] R.E. Bryant. *Graph-Based algorithms for boolean function manipulation.* IEEE Transaction on Computers, C-35(8):677-691, Aug. 1986.

[11] M. Cosnard, A. Ferreira. *On the real power of loosely coupled parallel architectures.* Parallel Processing Letters, Vol. 1, 2:103-112, 1991.

[12] C.A. Mead, L.A. Conway. *Introduction to VLSI systems.* Ed. Addison-Wesley, 1980.

[13] C. Coroyer, Z. Liu. *Effectiveness of heuristics and simulated annealing for the scheduling of concurrent tasks: an empirical comparison.* INRIA Research Report 1379, 1991.

[14] C. Lavarenne, Y. Sorel. *Performance Optimization of Multiprocessor Real-Time Applications by Graphs Transformations.* Proc. of the PARCO93 conference, France, 1993.

[15] Y. Sorel. *Massively Parallel Computing 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.

[16] Yves Sorel. *Real-time embedded image processing applications using the AAA methodology.* Proc. of the IEEE International Conference on Image Processing, Lausanne, September 1996.

[17] Page web de la méthodologie Adéquation Algorithme Architecture et du logiciel SynDEx qui la supporte : `http://www-rocq.inria.fr/syndex`

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

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