RENPAR'16 / CFSE'4 / SympAAA'2005 / JC
Le Croisic, France, 5 au 8 avril 2005

Modèle générique de hiérarchie mémoire pour l'exploration architecturale

Daniel Chillet, Lahcene Abdelouel, Olivier Sentieys

ENSSAT – Université de Rennes I,
IRISA,
6, rue de Kerampont - 22300 Lannion - France
prenom.nom@enssat.fr

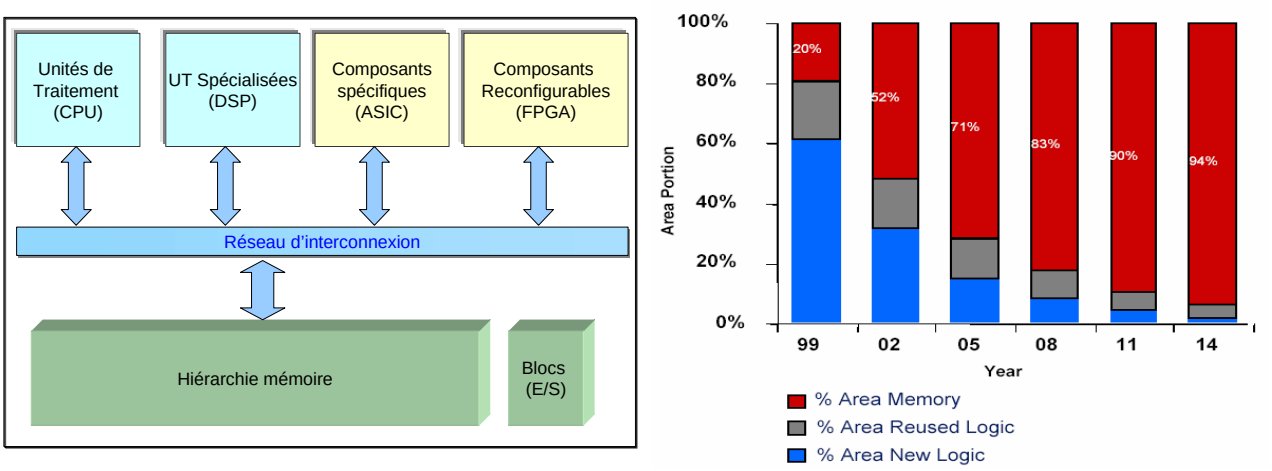
Résumé
Les performances en termes de calcul et consommation des applications à fort volume de données sont fortement dépendantes des accès mémoires. Aussi, la mise en œuvre d'une unité de mémorisation optimisée est plus qu'une nécessité. Cet article présente un modèle général de hiérarchie mémoire générique et souple pour s'adapter aux différentes architectures cibles possibles (SoC, FPGA, ARD). Le modèle architectural de la hiérarchie est présenté et une méthodologie de conception est développée. Cette méthodologie agit par raffinements successifs afin de trouver le placement optimal des données dans les différents niveaux de la hiérarchie mémoire. Celle-ci est optimisée pour l'application et tiens compte de l'architecture cible.

Mots-clés : System on Chip (SoC), hiérarchie mémoire, architecture, modèle

---

1. Introduction

Depuis quelques années, les concepteurs de circuits intégrés ont vu les capacités d'intégration des circuits augmenter de façon considérable leur offrant alors des possibilités d'implémentation de système complet sur une puce (*System On Chip, SoC*). Un SoC est généralement construit sur la base d'un médium de communication autour duquel vont graviter différents types de modules : des cœurs de processeurs ou cœurs de processeurs spécialisés, des accélérateurs matériels, des zones reconfigurables, des interfaces d'entrées sorties et enfin des mémoires. Un exemple de SoC est présenté sur la figure 1.a où chaque macrobloc réalise une fonction spécifique. Généralement, le CPU est chargé de l'agencement des différentes tâches de l'application, le DSP est habituellement responsable de décharger le CPU en assurant le traitement des motifs de calculs liés aux traitements du signal. Les composants spécifiques (ASICs) sont mis en œuvre pour traiter efficacement les fonctions critiques (chronophages) et les composants reconfigurables assurent au circuit des possibilités d'évolution. Enfin, la mémoire assure le stockage des données de l'application. Finalement, la communication entre les différents composants du système est assurée par un réseau d'interconnexions (bus, bus hiérarchiques, *crossbar, Network on Chip NoC*).
Conjointement à l'évolution des SoCs, les applications considérées pour ce type de système ont vu leurs besoins (en terme de puissance de calcul notamment) suivre une progression très importante.
La combinaison de ces deux évolutions place les concepteurs devant une multitude de solutions architecturales possibles créant un espace de recherche inexplorable exhaustivement. Alors que des techniques ont été développées pour prendre en charge tout ou partie de la partie opérative de l'application, l'aspect mémorisation a, jusqu'à présent, été peu exploré. Or, lorsque l'on observe les prévisions de répartitions des fonctionnalités au sein de l'architecture des SoCs pour les années à venir, on constate très clairement



a) b)

FIG. 1 – a) Architecture générale d'un SoC ; b) Part de la mémoire dans les SoC, source SIA Roadmap

que ces systèmes comporteront un pourcentage très important de mémoire (voir figure 1.b). En effet, les systèmes qui sont et seront développés couvrent le spectre des applications de communication au sens large pour lesquelles, même si les traitements sont complexes, le point critique relève du stockage des données et de leur acheminement. Sons et images y sont manipulés avec des résolutions importantes, engendrant des unités de stockages de plus en plus volumineuses, nécessitant souvent une organisation mémoire hiérarchique.
Aussi, l'étude d'implantation d'une application complète sur un SoC doit aujourd'hui prendre en considération les besoins en mémorisation et les capacités actuelles de ces circuits. Les travaux que nous menons actuellement concernent la définition d'une méthodologie de conception de hiérarchie mémoire pour les applications de traitement du signal (notamment les applications de télécommunication) pour des cibles de type SoC. Dans cet article nous focalisons notre attention sur la définition d'un modèle hiérarchique de la mémoire pour SoC ainsi que sur la définition d'une première trame de méthodologie adaptée à ce modèle hiérarchique.

La suite de l'article est organisée de la manière suivante. Dans la section 2, nous résumons l'état de l'art du domaine. La section 3 expose la problématique et présente le flot développé pour la conception de hiérarchie mémoire. La section 4 est consacrée à la modélisation de l'organisation mémoire. On y présente plus spécifiquement le modèle générique de l'architecture mémoire ainsi que le modèle des transferts des données. Dans la section 5, nous décrivons la méthodologie de conception développée pour construire -à partir du modèle générique- une hiérarchie mémoire dédiée à une application spécifique. Enfin, la section 6 conclut sur l'intérêt de notre approche et identifie quelques perspectives.

2. Etat de l'art

Les échanges entre les unités de traitement et les mémoires sont temporellement pénalisés par les performances de ces dernières. Il est admis que le *gap* entre les performances des unités de traitement et les temps d'accès des mémoires va croissant. Il s'ensuit que les performances globales des systèmes sont fortement limitées par les performances et l'architecture de l'unité mémoire. Ce problème trouve une réponse dans la mise en place d'une organisation hiérarchique des mémoires. Appliquée au monde des SoCs et des applications de traitement du signal, la mise en place d'une hiérarchie mémoire peut

bénéficier de connaissances particulières. En effet, les applications considérées sont généralement déterministes et il en découle que les besoins en mémorisation sont connus a priori (avant exécution).
De nombreuses études ont abordé la problématique de mise en place de hiérarchie mémoire au sein des systèmes, et l'on dénombre globalement trois types d'approches.

– La première approche consiste à déterminer, pour une application donnée, la hiérarchie optimale (nombre de niveaux, taille des mémoires...) en fonction de la localité des données [8] ;
– La seconde approche consiste à trouver, pour une architecture cible prédéfinie et une hiérarchie mémoire fixe, le placement optimal de chaque donnée dans le niveau de l'hiérarchie le plus adapté, afin de répondre aux contraintes de l'application [7, 10, 14] ;
– Enfin, la troisième approche combine la recherche d'une hiérarchie avec l'évaluation du placement des données dans le niveau hiérarchique le plus adapté et cela sous contrainte fixée par l'application [3, 2].

De nombreuses techniques de mémorisation développées selon ces différentes approches ont été abordées dans plusieurs contextes que nous énumérons dans les paragraphes suivants.

2.1. Synthèse de haut niveau et mémoire

Les premières techniques de synthèse de haut niveau ne se sont que très peu intéressées aux unités de mémorisation. Classiquement, les techniques d'ordonnancement abordent le problème de la synthèse mémoire en négligeant leur influence sur le coût mémoire. De même, la prise en compte des temps d'accès non homogène peut influencer le coût de la solution mémoire. La technique exposée dans [15] propose de revoir l'ordonnancement classique en y introduisant des contraintes de coût mémoire. Globalement, la technique travaille en trois étapes. La première étape est chargée de fournir une première solution qui est ensuite remise en cause par l'ordonnanceur (seconde étape) afin de placer les opérations de lecture et d'écriture en mémoire à des dates limitant le coût de stockage. Enfin la troisième étape effectue des regroupements de données afin de limiter le nombre de mémoire dans le système. Les résultats expérimentaux ont montré que le gain en surface était très peu significatif. Par contre le temps de recherche d'une solution est très largement diminué par rapport à une technique de recherche exhaustive bardée d'heuristiques. D'autre part, d'un point de vue consommation, la technique proposée permet un gain important. Sur les exemples présentés, le gain en consommation est en moyenne de 35%.

2.2. Hiérarchie mémoire

La mise en place de structures hiérarchiques pour répondre aux besoins en performances des systèmes classiques est une solution qui a été très largement explorée. Toutefois, une organisation à base de mémoires cache pose des problèmes dès lors que l'on aborde le domaine des applications temps réel. En effet, dans ce contexte, le concepteur n'accepte généralement pas que l'ensemble de l'ordonnancement qu'il a calculé soit mis à mal par un simple défaut de cache. Or, l'insertion d'un cache dans un système de ce type nuit gravement à sa prédictibilité et dès lors les caches sont très peu appréciés. Toutefois, une structure hiérarchique peut être conçue pour des besoins très spécifiques et une gestion particulière de cette hiérarchie peut aboutir à des gains importants. Dans [20], les auteurs proposent l'utilisation d'un cache des victimes afin de limiter les appels au cache de niveau deux et par la même limiter la chute de performances ainsi que le surcoût énergétique. Dans [9], Jouppi a démontré qu'un cache des victimes de seulement 4 entrées permettait une réduction entre 20 et 95 % des *miss* pour un cache à correspondance directe de 4 kOctets. Le cache des victimes est alors vue comme un paramètre de la conception au même titre que la taille d'un cache, ou son associativité. Le cache des victimes peut être placé ou non. Sur une configuration de cache à correspondance directe de 8 kOctets, Albera [1] a démontré que l'on avait un gain en consommation de l'ordre de 10% et un gain en performance de 3,5 % pour les Spec95.

2.3. Simultanéité des transferts

L'architecture d'un SoCs repose sur la mise en concurrence de nombreuses ressources sollicitant la mémoire pour y produire et/ou y consommer des données. Dans [16], les auteurs s'intéressent à ce problème dans le contexte où la mémoire est à l'extérieur du SoC. La problématique est liée à la gestion des conflits créés par les multiples requêtes qui vont parvenir au système mémoire. Ces requêtes, issues des différents blocs du SoC, n'ont pas forcément les mêmes cadences de fonctionnement et n'ont pas toutes la même "criticité". L'idée proposée dans cet article consiste à placer un bloc d'interface qui va récolter l'ensemble des requêtes pour la mémoire et d'ordonnancer ces demandes afin de soumettre, à la mémoire *Off-Chip*, une seule séquence d'accès. Le bloc proposé est composé d'un ordonnanceur de requêtes vers la mémoire et c'est à lui qu'incombe la gestion du séquencement des requêtes par rapport aux priorités des tâches (un critère de qualité de service est notamment pris en compte par l'ordonnanceur).

2.4. Mémoire et FPGA

Les circuits de type FPGA, dont le concept de reconfiguration a fait son apparition dans les SoC, a fait l'objet d'études spécifiques concernant les capacités de mémorisation. En effet, les constructeurs proposent actuellement des FPGA disposant de blocs mémoires de tailles de plus en plus grandes (jusqu'à 10 MB dans les FPGA les plus récents, comme le Virtex 4 de Xilinx ou le Stratix II d'Altera). Les solutions organisationnelles proposées reprennent une part des axes de recherche énumérés par Wilton dans [19]. Déclinées sous différentes caractéristiques, de temps d'accès, de taille, de consommation..., ces mémoires offrent un réel potentiel d'optimisation pour la synthèse mémoire. Dans [13], les auteurs présentent une formulation du problème de synthèse mémoire sur FPGA disposant d'une hiérarchie mémoire complexe. La formulation s'appuie sur le formalisme ILP (Integer Linear Programming) et propose la prise en compte de l'ensemble des paramètres du problème. Ainsi, le nombre de mémoires disponibles, leurs configurations (largeur $\times$ profondeur), les nombres de ports d'entrées/sorties, les temps d'accès en lecture et en écriture sont des paramètres de la formulation et sont pris en compte comme contraintes globales. La fonction globale minimisée est une fonction de coût qui est une composition des paramètres de latence, de délai de traversée des broches d'entrées/sorties ainsi que du coût des broches d'entrées/sorties. Les auteurs précisent que leur formulation s'appuie sur des structures de données identifiées auparavant et donc déjà formées. Ce point est important car il précise les limites de la méthode.

Compte tenu de l'existence de la mémoire au sein des circuits de type FPGA et de sa non-utilisation pour réaliser du stockage, plusieurs auteurs proposent d'utiliser une partie de celle-ci comme source de fonctions logiques. Ainsi, dans [17, 18, 5], les auteurs proposent des méthodes permettant de *mapper* des fonctions logiques classiques dans des zones de mémoire inutilisées par ailleurs.

2.5. Mémoire et faible consommation

La mémoire ayant un coût surfacique important dans les circuits de type SoC et étant très largement sollicitée, on conçoit aisément que la consommation engendrée par cette unité puisse être très importante. Négliger le critère consommation lors de la conception de cette unité au profit de la consommation de l'unité de traitement est une erreur fondamentale, tant du point de vue de la consommation dynamique que du point de vue de la consommation statique. En effet, il semble de plus en plus évident que dans les années à venir, la consommation statique ne pourra plus être ignorée lors de la conception. Ceci est la conséquence d'une évolution technologique ramenant petit à petit la consommation statique au même ordre de grandeur que la consommation dynamique. Dans l'article [11], les auteurs proposent une méthode permettant de placer les différents accès aux différentes mémoires de telle sorte qu'il soit possible de compacter le maximum d'activité sur quelques mémoires et donc faire apparaître des mémoires avec peu d'accès. Dans ce cas, les mémoires ayant peu d'accès peuvent alors être mises en veille pendant des périodes non négligeables, ce qui permet un gain en consommation relativement important. La méthode proposée part du principe que les accès ont déjà été ordonnancés pour satisfaire la partie calcul. La mé-

thode considère l'ensemble des données comme étant scalaires et travaille sur un système composé de deux bancs mémoires. Le gain obtenu est de l'ordre de 16%.

L'ensemble des travaux énumérés dans les paragraphes précédents présente des points intéressants mais ne propose pas de méthode globale pour tenter de répondre efficacement au problème du stockage des données dans un circuit de type SoC. Nous proposons d'attaquer cette problématique en s'attachant tout d'abord à définir un modèle hiérarchique générique suffisamment souple, puis en proposant une méthodologie de conception adaptée.

3. Problématique et positionnement de nos travaux

Dans cette section, nous présentons tout d'abord le contexte de nos travaux avant de poser la problématique et les questions qui en découlent.

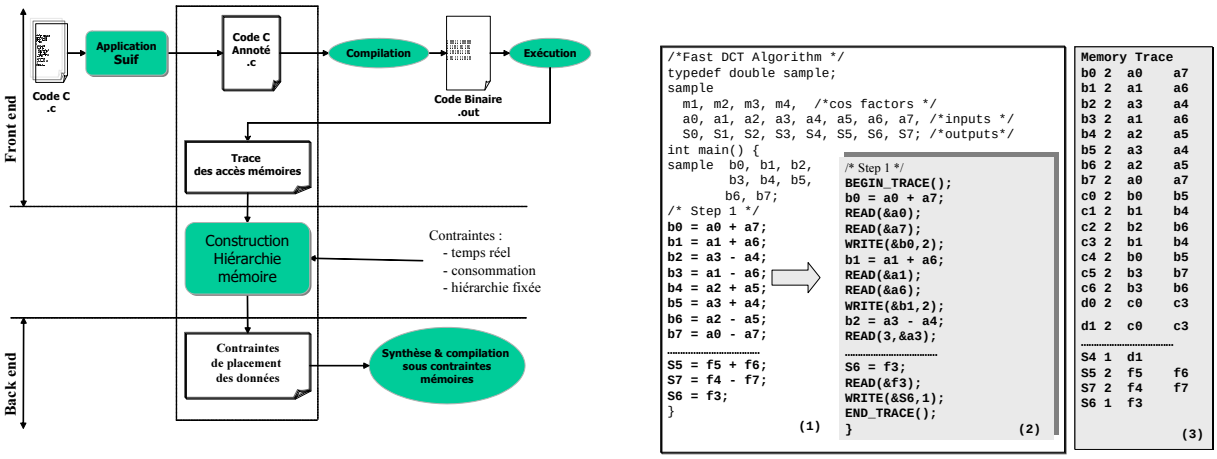
Contexte : Soit une application manipulant un ensemble de données important et devant être implémentée sur un SoC disposant d'une organisation mémoire hiérarchique complexe. Les données sont manipulées au travers d'opérations de lecture et d'écriture (de transferts vers ou depuis la mémoire). La spécification des besoins de l'application est contenue dans une liste d'accès à la mémoire.

Problématique : Quel est, du point de vue des critères de surface, consommation et performances, le placement "optimal" des données dans les différentes mémoires du SoC ?

La recherche d'une réponse à cette question soulève des sous problèmes liés aux points suivants :
– quel est le nombre de niveaux mémoires à mettre en place dans la hiérarchie ?
– quelles sont les tailles des mémoires dans les différents niveaux de la hiérarchie ?
– quels liens faut-il créer entre ces niveaux de hiérarchie mémoire ?
– comment les générateurs d'adresses associés à chacune des mémoires doivent-ils être conçus ?

L'exploration exhaustive de l'espace des solutions est évidemment impossible compte tenu notamment de la quantité de données à considérer. Pour contourner cette complexité, nous proposons une approche incrémentale (développée à la section 5) en trois étapes sans remise en cause des choix effectués aux étapes précédentes.

Le flot de conception global que nous proposons (voir figure 2.a) débute par une première étape réalisée via le *front end* de compilation *Suif*. Sur la base de *Suif*, nous avons développé un module qui nous permet d'effectuer une annotation des accès mémoires de toute application codée en C (figure 2.b.1). Les annotations sont positionnées automatiquement à chaque fois qu'un accès en lecture ou en écriture est détecté par l'outil *Suif*. Cette détection engendre l'ajout d'un appel à une procédure (figure 2.b.2), qui dans notre cas effectue une sortie textuelle dans un fichier indiquant l'opération mémoire (lecture ou écriture), et des informations concernant cet accès mémoire (dépendance de l'opération mémoire vis à vis des autres données de l'algorithme).
Le point d'entrée de notre méthodologie est donc une liste d'accès mémoire définissant l'ordre des opérations ainsi que les dépendances entre celles-ci (figure 2.b.3). Le point de sortie de la méthodologie spécifie le stockage des données dans les différentes mémoires du SoC. Ce point de sortie peut ensuite constituer l'entrée d'un outil *Back end* réalisant la synthèse des autres unités sous contrainte mémoire. Les techniques développées dans [6] peuvent alors être appliquées afin de définir la partie traitement.



a) b)

FIG. 2 – a) Position de nos travaux dans un flot global de construction d'une hiérarchie mémoire ; b) Exemple de code source annoté et de trace des accès mémoires

4. Modélisation de l'organisation mémoire

Dans cette section, nous décrivons l'organisation mémoire au travers de sa modélisation architecturale et de la modélisation des transferts de données. Le premier modèle est matériel et présente la structure sur laquelle nous développons notre méthodologie. Le second modèle, permet d'identifier les transferts de données dans l'organisation mémoire proposée.

4.1. Modèle architectural

Dans le contexte qui nous intéresse, le déterminisme complet de la séquence des accès aux données, permet d'envisager la construction d'une hiérarchie mémoire en adéquation avec l'application. La recherche de cette adéquation conduit généralement à un modèle architectural utilisant beaucoup de mémoires dédiées et indépendantes. Ce modèle est d'autant plus attrayant que ces mémoires peuvent être embarquées et qu'il est dès lors possible d'envisager des organisations internes complexes. Ces nouvelles organisations offrent un réel potentiel d'optimisation en performances et en consommation.

À partir des points mentionnés ci-dessus, nous avons défini un modèle générique d'architecture mémoire. Ce modèle, dont une présentation est donnée par la figure 3, est basé sur une hiérarchie à trois niveaux. La justification des trois niveaux se base sur les raisons suivantes.

– Dans les systèmes à base de microprocesseurs, une hiérarchie à trois niveaux semble une solution performante. La plupart des microprocesseurs récents intègre d'ores et déjà deux caches séparés L1 (instructions & données) de niveau 1 et un cache unifié L2 de second niveau. La tendance actuelle montre que le troisième niveau de cache commence à apparaître dans quelques processeurs.
– Certains constructeurs de circuits reconfigurables de types FPGA, proposent des structures hiérarchiques constituées de un ou deux, parfois de trois niveaux de mémoires. On citera par exemple, les circuits de la famille Virtex de Xilinx qui proposent deux niveaux de mémoires, le premier niveau est offert via la configuration des cellules standard (CLB) en blocs mémoires. Le deuxième niveau de mémoire est constitué de plusieurs blocs mémoires double port pouvant aller jusqu'à 4 kbits. On citera encore la famille des circuits Stratix du constructeur Altera dans laquelle une organisation nommée Trimatrix en trois niveaux de mémoires est disponible.

Les trois niveaux de la hiérarchie mémoire que nous proposons se décomposent en deux pseudo caches

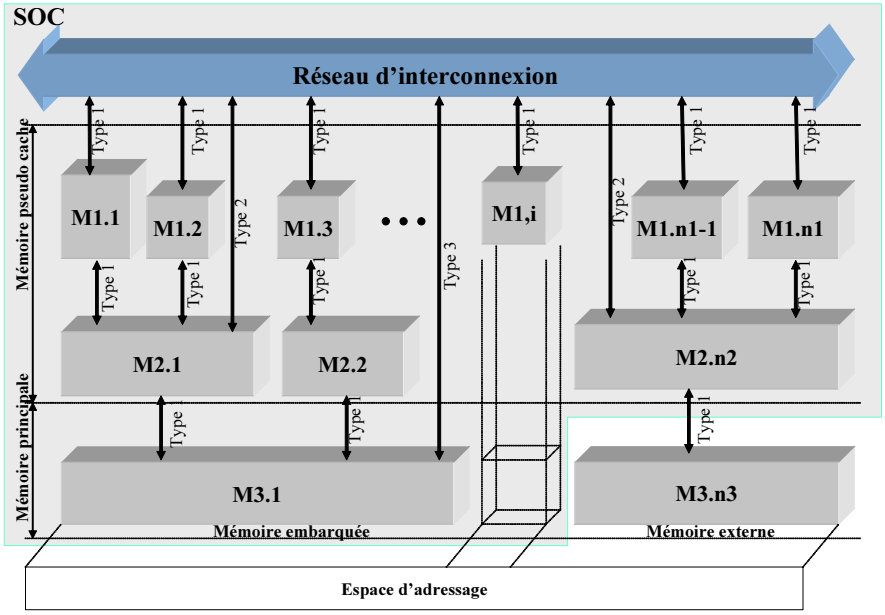


FIG. 3 – Modélisation de hiérarchie mémoire basée sur trois niveaux de mémoires, et illustration des types de transferts de données envisagés entre les niveaux de mémoires de la hiérarchie

de niveau 1 et 2 et une mémoire principale de niveau 3. Le terme pseudo cache est ici utilisé puisque nous faisons référence à un fonctionnement prévisible (en relation avec le déterminisme de la séquence d'accès), ce qui n'est évidemment pas le cas dans les systèmes classiques.

Du point de vue du développeur, il existe un seul espace de nommage global : le programmeur voit donc une seule mémoire. L'espace d'adressage est divisé entre la Mémoire ScratchPad (MSP, directement *mappée* dans l'espace adressable) et la mémoire principale de niveau trois, constituée par les blocs RAM embarqués (M3.1) et éventuellement par les bancs mémoire RAM externes (M3,n3). Les pseudo caches de niveau 1 et 2 sont construits à partir de blocs mémoires de différentes tailles. Les blocs mémoires de petites tailles seront utilisés pour la sauvegarde des variables et des petits tableaux les plus fréquemment accédés. Ces blocs mémoires permettront l'exploitation de la localité temporelle des données de l'application. La localité spatiale, concernant l'accès à des données contiguës, sera exploitée au travers des blocs mémoires de grandes tailles.

4.2. Modèle de transferts de données

En s'appuyant sur le modèle architectural, nous avons défini un modèle de transfert entre les différents niveaux de mémoire. Ce modèle offre plusieurs types de connexions notées Type 1, 2 et 3. Chacun de ces types peut supporter des temps d'accès différents.

– Les connexions de type 1 nous donnent la possibilité de transférer des données entre les différents niveaux hiérarchiques, par exemple entre les mémoires caches de niveau 2 et les mémoires caches de niveau 1 ou entre la mémoire principale et la mémoire cache de niveau 2.
– Les connexions de type 2 (resp. 3) offrent une plus grande flexibilité en permettant les transferts de données entre les niveaux de mémoire cache 2 (resp. 3) et les registres de l'unité de traitement.

Les transferts de type 1 reproduisent le fonctionnement normal des caches classiques où les données passent d'un niveau à l'autre sur base de la fréquence à laquelle elles sont référencées. Les transferts de

type 2 et 3 se justifient dans la mesure où il sera opportun pour certaines données moins fréquemment accédées et placées dans des niveaux supérieurs (2, 3) de s'affranchir de la traversée de toute la structure hiérarchique de la mémoire cache pour atteindre les registres de l'unité de traitement, ce qui semble être un problème à la fois pour le traitement temps réel et la faible consommation dans le contexte des SoCs. Dans notre modèle de transferts, nous avons limité les transferts possibles entre les entrées/sorties (E/S) et les niveaux de mémoire. Les E/S sont en relation directe avec la mémoire principale (de plus haut niveau) qui agit comme tampon. Cette solution a été retenue pour des raisons de simplicité de réalisation du SoC.

5. Méthodologie globale de conception

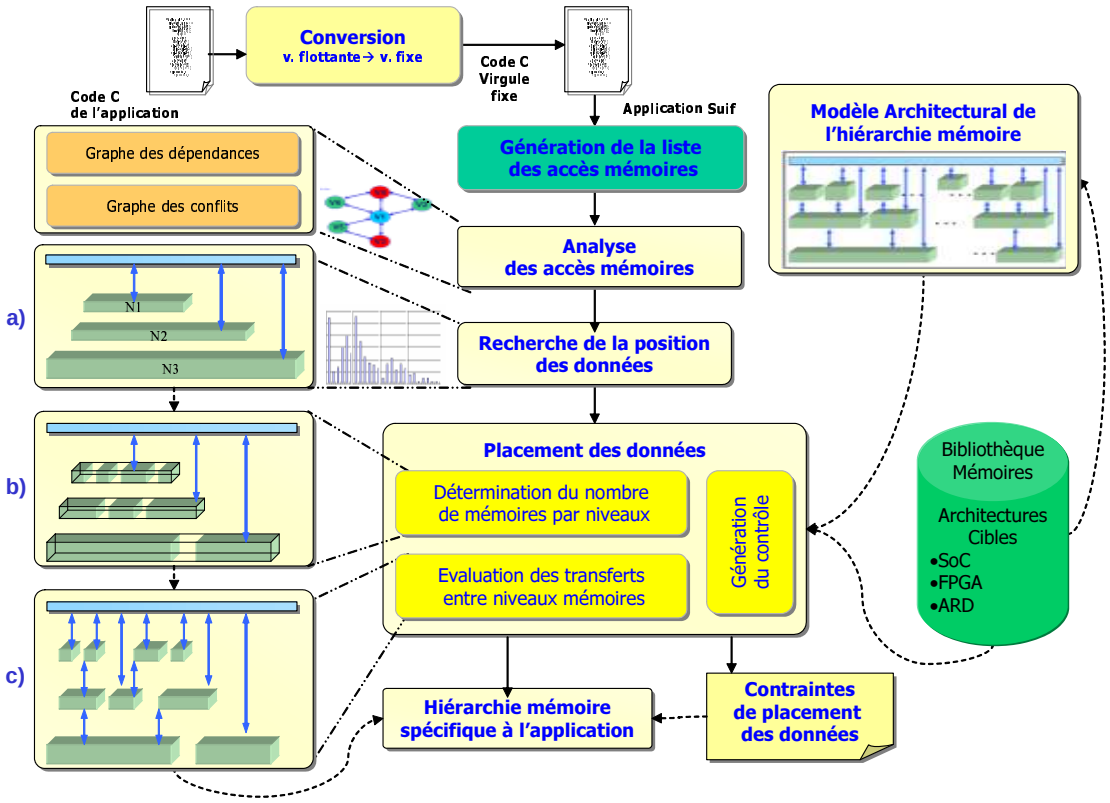


FIG. 4 – Méthodologie proposée selon une découpe en trois étapes, permettant d'aboutir au modèle architectural de la hiérarchie mémoire

Comme nous l'avons indiqué précédemment, le point de départ de notre méthodologie est l'algorithme écrit dans le classique format flottant des langages évolués type C. Or, pour des raisons de fonctionnement temps réel, de faible consommation et de coût, l'implémentation exige généralement l'emploi du format à virgule fixe. Notre équipe a développé une méthodologie de transformation de spécification en virgule flottante vers une spécification en virgule fixe. Cette méthodologie cible des architectures programmables et prend en compte des contraintes de précision et de temps d'exécution [12]. L'outil de conversion implémentant cette méthodologie permet de déterminer la largeur de chaque donnée de l'algorithme en

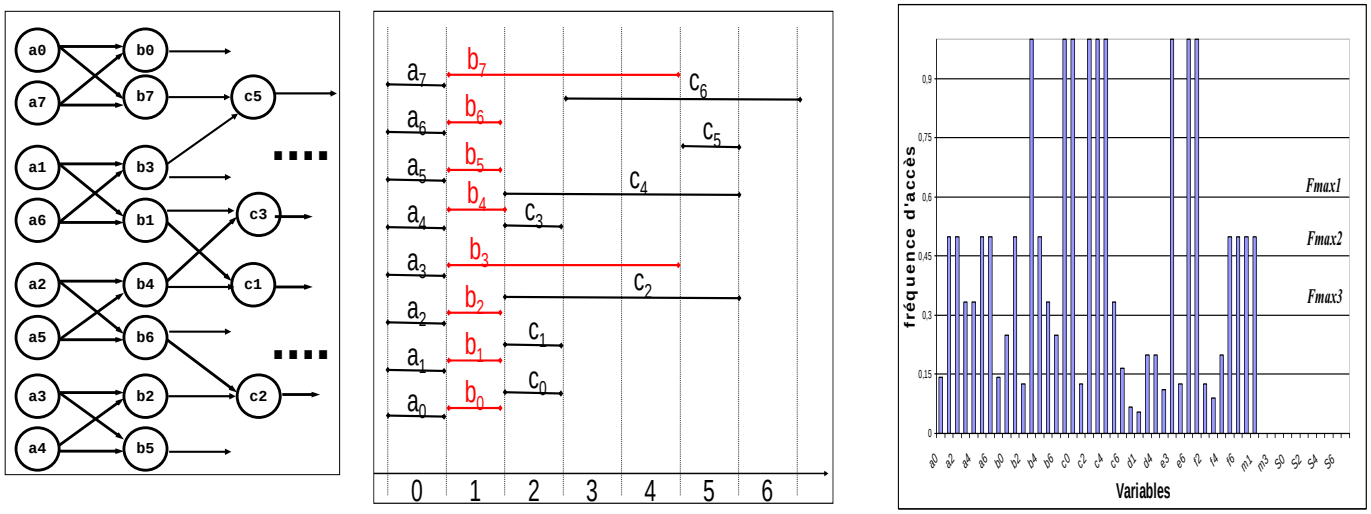
prenant en compte l'ensemble des types des données manipulées au sein de l'architecture programmable. Cette première étape nous permet de définir le format de codage en virgule fixe approprié et pourra entraîner une réduction significative de l'espace mémoire nécessaire aux données de l'application. Le module *front end Suif* nous permet ensuite d'extraire la liste des accès mémoires et les dépendances entre les données. Notre méthodologie doit nous permettre de construire une hiérarchie mémoire et de trouver le placement optimal des données dans les différents niveaux de la hiérarchie mémoire. La hiérarchie mémoire est basée sur le modèle générique défini dans la section 4.
En combinant des mémoires de tailles et de capacités différentes et en permettant des transferts en parallèle entre plusieurs blocs mémoires et plusieurs niveaux de caches, le modèle hiérarchique proposé est de notre point de vue, aussi souple et globale que possible. Il peut couvrir une gamme importante de cibles allant des mémoires embarquées dans les SoCs aux modules mémoires externes (SRAM et DRAM) tout en exploitant les possibilités de mémorisation offertes par les FPGAs et architectures reconfigurables. Mais cette diversité et cette flexibilité dans la hiérarchie entraînent des problématiques nouvelles : déterminer la meilleure architecture mémoire et le placement optimal des données dans la hiérarchie n'est pas un problème facile.
La méthodologie que nous proposons recherche le placement optimal des données dans les différents niveaux de la hiérarchie mémoire optimisée pour l'application. Cette approche se déroule en trois étapes qui sont les suivantes (voir figure 4) :
– la première étape a pour objectif de déterminer la position des données dans les différents niveaux de mémoire (figure 4.a) ;
– la seconde étape détermine, pour chaque niveau mémoire, le nombre de mémoires nécessaires (figure 4.b) ;
– enfin, la troisième étape évalue les transferts entre niveaux mémoires (figure 4.c).
Nous détaillons, dans les paragraphes suivants, ces différentes étapes.

5.1. Recherche de la position des données

Cette première étape consiste à rechercher la meilleure position de stockage des données en fonction de leur fréquence d'accès. Dans cette étape, il n'est fait aucune hypothèse sur les caractéristiques finales des mémoires. De plus, on fait l'hypothèse que chaque niveau mémoire est composé d'une seule mémoire ayant autant de ports d'entrée/sortie que désiré. Cette hypothèse sera levée dans l'étape 2.

La détermination de la position des données dans la hiérarchie s'appuie sur une formulation ILP. Cette formulation s'exprime à partir de variables dont nous donnons les définitions ci-dessous.
– Soit $Data$ l'ensemble des données de l'algorithme : $Data = \{d_i\}_{\forall\, i=1...D}$ avec d_i la i^{ieme} donnée de l'algorithme et D le nombre de données manipulées par l'algorithme.
– Soit Ad_i l'ensemble des accès à la i^{ieme} donnée : $Ad_i = \{ad_{ij}\}_{\forall\, j=1...Na_i}$, avec Na_i le nombre d'accès à la i^{ieme} donnée. Les accès en lecture ou en écriture ne sont pas différenciés.
– Soit Td_i l'ensemble des temps de cycle d'accès à la i^{ieme} donnée : $Td_i = \{td_{ij}\}_{\forall\, j=1...Na_i}$
Les temps de cycle d'accès pour chaque donnée sont obtenus par une opération de pré-ordonnancement du graphe des accès mémoire. L'extraction de la liste des accès par le module *Suif*, nous fournit une information d'ordre à respecter pour l'enchaînement des opérations de lecture et écriture. Cette liste d'accès est modélisée par un graphe d'accès mémoire (figure 5.a) et ce graphe est ensuite simplement pré-ordonnancé par un algorithme ASAP (*As Soon As Possible*). Ce pré-ordonnancement nous fournit alors une information, a priori, sur les temps de cycle auxquels les données seront lues ou écrites en mémoire (figure 5.b).

A partir des variables précédemment définies, nous calculons, pour chaque donnée, la fréquence d'accès



a) b) c)

FIG. 5 – a) Graphe de dépendances des variables ; b) Graphe des durées de vies des variables ; c) Fréquences des accès mémoires

maximale à laquelle la donnée est susceptible d'être accédée. Cette fréquence est définie par la relation suivante :

$$Fd_i = \frac{1}{MIN_{\forall\ j=1...Na_i-1}((td_{i(j+1)} - td_{ij})}$$

Cette fréquence donne une indication sur la pertinence de stockage de chaque donnée dans les différents niveaux mémoires. Une donnée ayant une fréquence d'accès élevée devra plutôt être stockée dans un niveau mémoire de faible taille et donc rapide. Cette information de fréquence d'accès des données est mise en adéquation avec les fréquences d'accès maximales des niveaux mémoires (figure 5.c). Ces dernières sont définies comme suit :

– Soit $Fmax_n$ la fréquence d'accès maximale du n^{ieme} niveau mémoire.

Alors la position P_i de la i^{ieme} donnée dans un niveau de la hiérarchie est fixée de la façon suivante :

$$\begin{aligned} P_i &= 0 \quad si\ Fmax_1 < Fd_i \\ P_i &= 1 \quad si\ Fmax_2 < Fd_i \leq Fmax_1 \\ P_i &= 2 \quad si\ Fmax_3 < Fd_i \leq Fmax_2 \\ P_i &= 3 \quad si\ 0 < Fd_i \leq Fmax_3 \end{aligned}$$

Le cas $P_i = 0$ indique qu'il est impossible de stocker la donnée dans le niveau mémoire le plus rapide et que dans ce cas elle devra être stockée dans les éléments de traitement.

5.2. Détermination du nombre de bancs mémoires par niveaux et placement des données dans ces bancs

Au stade précédent de la méthodologie, les niveaux mémoires sont supposés de taille infinie et on ne se préoccupe pas du nombre de ports nécessaires pour satisfaire le parallélisme des accès. La seconde

étape a pour objectif de travailler sur chaque niveau afin de déterminer le nombre de bancs mémoires nécessaires. Ce nombre de bancs mémoires dépend principalement du nombre d'accès aux mémoires à assurer en parallèle, mais il dépend aussi du nombre de ports de chaque mémoire. Nous proposons donc de rechercher, dans un premier temps, le nombre de ports d'accès mémoire nécessaire pour chaque niveau de la hiérarchie. Puis, en fonction des mémoires disponibles dans la bibliothèque, nous définissons le nombre de mémoires de chaque niveau.
La recherche du nombre de ports s'appuie sur un graphe de conflit d'accès mémoire. Celui-ci est construit par niveau de mémoire et la recherche de la clique de cardinalité maximale fournit le nombre de bancs mémoires nécessaire pour assurer la simultanéité des transferts. Le graphe de conflit GC_n de niveau n est constitué d'un ensemble de nœuds N_n représentant les données d_i à mémoriser au niveau n, et d'un ensemble d'arêtes A_n représentant les conflits d'accès entre les données.

$$\begin{aligned} GC_n &= (N_n, A_n) \\ N_n &= \{d_i \mid P_i = n \qquad \forall\, i = 1 \ldots D\} \\ A_n &= \{(a_i, a_j) \mid P_i = P_j = n \ \wedge \ \exists\, k, l \mid ad_{jk} \in]ad_{il} - Tmin_n \ ; \ ad_{il} + Tmin_n[\} \end{aligned}$$

Avec $Tmin_n = \frac{1}{Fmax_n}$ le temps d'accès minimum que peut supporter le n^{ieme} niveau mémoire. Une arête (a_i, a_j) est créée entre les nœuds d_i et d_j si et seulement si, il existe un conflit de transfert entre les deux données correspondantes. La notion de conflit entre deux données apparaît dès lors qu'il existe un transfert, pour chacune des données, et que ces transferts sont dans le même temps de cycle mémoire.

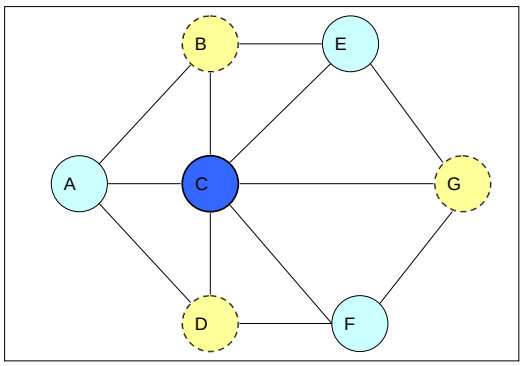


FIG. 6 – Coloriage du graphe de conflit d'accès

La recherche du nombre de ports d'accès à la mémoire nécessaires pour mémoriser l'ensemble des données est équivalente à la recherche de "l'index chromatique" du graphe. Il s'agit de rechercher le nombre minimum de couleurs nécessaires pour colorier tous les nœuds du graphe sous la contrainte que deux nœuds, représentant deux données du même niveau mémoire, reliés par une arête ne portent pas la même couleur (figure 6). Une formulation est donnée ci-dessous.

$$NbPorts_n = IndexChromatique\,(GC_n) = NbMinCouleurs\,(GC_n)$$

La contrainte à respecter est la suivante :

$$Cd_i \neq Cd_j \ \ si\ P_i = P_j = n \ \ \wedge \ \ \exists\ (a_i, a_j)$$

Et avec Cd_i la couleur de la i^{ieme} donnée.

C'est à ce niveau qu'intervient la bibliothèque de mémoires. En effet, il faut mettre en adéquation le nombre de ports d'accès des mémoires disponibles avec le nombre de ports globals nécessaires. Il faut assurer la pré-sélection de NbM_n mémoires au niveau n de la hiérarchie, tel que ces NbM_n mémoires présentent au moins $NbPorts_n$ ports d'accès. Si chacune des mémoires de la bibliothèque présente un

et un seul port d'accès, alors $NbM_n = NbPorts_n$. Dans cet article, nous supposons que la bibliothèque de mémoires est constituée uniquement de mémoires simple port.
Pour un niveau mémoire donné, on réalise ensuite une distribution des données dans les NbM_n bancs mémoires du niveau considéré. Cette phase consiste à attribuer un numéro de ports d'accès mémoire à chaque donnée de la séquence de transferts du niveau mémoire en question. On peut assimiler cette tâche au coloriage du graphe de conflits avec un nombre de couleurs égal au nombre de ports minimum de ce niveau. Chaque couleur représentant alors un numéro de port d'accès mémoire. Compte tenu de l'hypothèse faite précédemment sur la constitution de la bibliothèque (uniquement des mémoires simple port), l'attribution d'un numéro de ports à chaque donnée est équivalente à la distribution d'une donnée dans une mémoire spécifique.
De nombreux travaux ont abordé cette problématique, et une formulation proche de celle que nous utilisons peut être trouvée dans [4].

5.3. Evaluation des transferts entre niveaux mémoires

Durant cette dernière étape, les caractéristiques de la cible matérielle vont être prises en compte de façon plus complète. L'étape 2 nous indique, par niveau mémoire, quels sont les nombres de bancs mémoires ainsi que les tailles de ces mémoires à mettre en place. Dans le cas où le circuit physique ne propose pas suffisamment de mémoire au niveau n, alors cette étape évalue le transfert de blocs de données entre le niveau n et le niveau $n+1$. La technique utilisée consiste à analyser si les données ont des durées d'utilisations disjointes ou pas. Lorsque deux données ont des durées d'utilisations disjointes alors une seule case mémoire peut être utilisée pour stocker ces deux données. Toutefois, cela peut engendrer le renvoi du stockage de l'une et/ou l'autre des données vers le niveau supérieur. C'est le cas, par exemple, lors du stockage d'une constante qui n'est plus utile à l'algorithme (pendant un temps suffisamment long) et qui peut alors laisser sa place en mémoire au profit d'une autre donnée.
Cette étape doit prendre en considération le taux d'utilisation des différentes mémoires ainsi que les aspects consommation. En effet, il est important de s'attacher à fournir une solution assurant un taux d'utilisation des mémoires le plus élevé possible, de même il est important de considérer, à chaque fois qu'un transfert entre niveaux mémoires est mis en place, que celui-ci n'engendre pas un surcoût en consommation trop important.
L'ensemble des paramètres nécessaires pour cette étape est défini dans la bibliothèque de mémoires. Cette bibliothèque ne doit pas se limiter aux principales caractéristiques des mémoires telles que le type, la capacité, l'organisation et latence, mais doit rassembler le maximum de paramètres susceptibles d'être exploités par les algorithmes de sélection/allocation. Le tableau 1 donne une liste non exhaustive des paramètres pertinents qui peuvent intervenir lors du choix des mémoires :
D'un point de vue temporelle, l'aspect déterministe de la séquence d'accès aux données permet d'envisager des transferts entre niveaux mémoires sans interférer avec les besoins de l'application. D'un point de vue consommation, le transfert de données entre le niveau n et $n-1$ peut trouver un intérêt. La relation ci-dessous fournit un exemple qui permet de juger de l'intérêt du transfert de la i^{ieme} donnée du niveau n au niveau $n-1$.

$$1 * (Pr_n + Pw_{n-1}) + (Nr_i * Pr_{n-1} + Nw_i * Pw_{n-1}) \leq (Nr_n * Pr_n + Nw_n * Pw_n)$$

Avec Nr_i le nombre de lectures de la i^{ieme} donnée et Nw_i le nombre d'écritures cette même donnée.
Cette expression fait le bilan énergétique de la mise en place du transfert d'une donnée entre le niveau n et $n-1$. La partie gauche de l'expression exprime le fait que la donnée est transférée du niveau n vers le niveau $n-1$ et que l'ensemble des accès à cette donnée est donc réalisé dans le niveau $n-1$. La partie droite de l'expression exprime le stockage de la donnée au niveau n et par conséquent le coût énergétique des accès à ce niveau.

Mémoire	Param.	Caractéristique désignée
Organisation et accès	W	Taille de la plus petite cellule adressable en nombre de bits
	L	Nombre de cellules (mots) mémoires ou profondeur mémoire
	S	Capacité mémoire (Nombre de Kbits) : S=L*W
	Np	Nombre -et éventuellement type- de ports des mémoires Multiports.
	Mode	Différents modes d'accès : Accès en mode aléatoire, en mode rafale (brust)
Temps d'accès	tac	Cycle-mémoire : temps nécessaire pour effectuer une requête de transfert
	tw	Temps d'accès en écriture (Mémorisation de l'information)
	tr	Temps d'accès en lecture (Restitution de l'information)
	tb	Temps d'accès pour effectuer une requête de transfert en mode rafale
	ts	Temps de veille : temps nécessaire pour la mise en veille de la mémoire
	ta	Temps de réveil : temps nécessaire pour la réactivation de la mémoire
Consommation	Pw	Consommation en énergie des accès en écriture
	Pr	Consommation en énergie des accès en lecture
	Pm	Consommation en énergie pour le maintien de l'information
	Ps	Consommation en énergie pour la mise en veille de la mémoire
	Pa	Consommation en énergie pour la réactivation de la mémoire

TAB. 1 – Caractéristiques des mémoires

Un ensemble de règles similaires peut être établi sur la base des caractéristiques des mémoires données dans le tableau 1.

6. Conclusion

Dans cet article, nous avons posé les bases d'une méthodologie de conception de hiérarchie mémoire ciblant des circuits de type FPGA et/ou SoC. La méthode proposée s'appuie sur une décomposition du problème en plusieurs étapes, permettant dans un premier temps de travailler sur une solution hors de toutes considérations architecturales, puis de prendre en compte la cible d'implémentation. Nous ciblons un modèle architectural hiérarchique souple et générique lors des premières étapes de conception, puis nous dérivons la solution obtenue pour qu'elle puisse être *mappée* sur la cible finale. L'intérêt de cette stratégie repose sur le fait de découpler la partie définition de l'organisation mémoire de la partie placement des données. En effet, dans une première étape, on vise l'adaptation du modèle aux besoins spécifiques de l'application en fixant les paramètres (optimaux) de l'organisation mémoire. Ce n'est que dans une seconde étape, basée essentiellement sur une bibliothèque des caractéristiques des mémoires, que la méthodologie agit par des raffinements successifs sur le modèle pour aboutir à une hiérarchie mémoire tenant compte de l'architecture cible.

Le travail présenté dans cet article est actuellement en phase d'outillage informatique et cette phase devrait nous permettre d'obtenir des résultats très prochainement. La méthodologie que nous avons présentée s'appuie sur les accès de chaque donnée de l'algorithme. Il est clair que le volume de données manipulées peut très vite s'avérer un point critique et conduire à l'impossibilité d'obtenir une solution dans un temps raisonnable. Les travaux futurs s'orienteront vers la prise en charge des variables multi-dimensionnelles et les groupes de données. Dans ce cas, chaque tableau ou groupe de données sera considéré comme une donnée unique et notre méthodologie peut assez simplement être adaptée en proposant une étape supplémentaire de regroupement des données.

Bibliographie

1. Albera (G.) et Bahar (R.). – Power and performance tradeoffs using various cache configurations. *In : International Symposium on Computer Architecture*. – 1998.

2. Catthoor (Francky). – Energy-delay efficient data storage and transfer architectures and methodologies : Current solutions and remaining problems. *Journal of VLSI Signal Processing Systems*, vol. 21, n3, jul 1999, pp. 219–231.
3. Catthoor (Francky), Wuytack (S.), Greef (E. De), Balasa (F.), Nachtergaele (L.) et Vandercappelle (A.). – *Custom Memory Management Methodology*. – Norwell, MA, USA, Kluwer Academic Publishers, 1998.
4. Chillet (Daniel), Sentieys (Olivier) et Corazza (Michel). – Memory unit design for real time dsp applications. *In : Ninth Great Lakes Symposium on VLSI*. – Ann Arbor, Michigan, USA, Mar, 04 - 06 1999.
5. Cong (Jason) et Xu (Songjie). – Technology mapping for (fpgas.) with embedded memory blocks. *In : FPGA '98 : Proceedings of the 1998 ACM sixth international symposium on Field programmable gate arrays*. pp. 179–188. – Monterey, CA, feb 1998.
6. Corre (Gwénolé), Senn (Eric), Julien (Nathalie) et Martin (Eric). – A memory aware behavioral synthesis tool for real-time vlsi circuits. *In : GLSVLSI '04 : Proceedings of the 14th ACM Great Lakes symposium on VLSI*. pp. 82–85. – Boston, MA, USA, avril 2004.
7. Diguet (J. Ph.), Wuytack (S.), Catthoor (F.) et Man (H. De). – Formalized methodology for data reuse exploration in hierarchical memory mappings. *In : ISLPED '97 : Proceedings of the 1997 international symposium on Low power electronics and design*. pp. 30–35. – Monterey, California, United States, aug 1997.
8. Jacob (Bruce L.), Chen (Peter M.), Silverman (Seth R.) et Mudge (Trevor N.). – An analytical model for designing memory hierarchies. *IEEE Transactions on Computers*, vol. 45, n10, oct 1996, pp. 1180–1194.
9. Jouppi (Norman P.). – Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *In : ISCA '90 : Proceedings of the 17th annual international symposium on Computer Architecture*. pp. 364–373. – Seattle, Washington, United States, jun 1990.
10. Kolson (David), Nicolau (Alex) et Dutt (Nikil D.). – Elimination of redundant memory traffic in high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 15, nov 1996, pp. 1354–1364.
11. Lyuh (Chun-Gi) et Kim (Taewhan). – Memory access scheduling and binding considering energy minimization in multi-bank memory systems. *In : DAC '04 : Proceedings of the 41st annual conference on Design automation*. pp. 81–86. – San Diego, CA, USA, jun 7-11 2004.
12. Menard (Daniel), Chillet (Daniel), Charot (François) et Sentieys (Olivier). – Automatic floating-point to fixed-point conversion for dsp code generation. *In : CASES '02 : Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*. pp. 270–276. – Grenoble, France, oct 2002.
13. Ouaiss (Iyad) et Vemuri (Ranga). – Hierarchical memory mapping during synthesis in fpga-based reconfigurable computers. *In : DATE '01 : Proceedings of the conference on Design, automation and test in Europe*. pp. 650–657. – Munich, Germany, mar 2001.
14. Saied (R.) et Chakrabarti (C.). – Scheduling for minimizing the number of memory accesses in low-power applications. *In : Proc. of the VLSI Signal Processing Workshop*, pp. 169–178. – San Francisco, CA , USA, oct 1996.
15. Seo (Jaewon), Kim (Taewhan) et Panda (Preeti R.). – An integrated algorithm for memory allocation and assignment in high-level synthesis. *In : DAC '02 : Proceedings of the 39th conference on Design automation*. pp. 608–611. – New Orleans, Louisiana, USA, jun 10 - 14 2002.
16. W. D. Schwaderer (Sonics Inc. (USA)). – Solving (soc) shared memory resource challenges. – IP Based SoC Design'2002, International Workshop & Exhibition, oct 30-31, Grenoble, France 2002.
17. Wilton (Steven J. E.). – Implementing logic in fpga embedded memory arrays : Architecture implications. *In : CICC 1998 : IEEE Custom Integrated Circuits Conference*. – Santa Clara, CA, USA, may 11-14 1998.
18. Wilton (Steven J. E.). – Heterogeneous technology mapping for fpgas with dual-port embedded memory arrays. *FPGA '00 : Proceedings of the 2000 ACM eighth international symposium on Field programmable gate arrays*, feb 2000, pp. 67–74.
19. Wilton (Steven J.E.). – Embedded memory in fpgas : Recent research results. *In : IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*. – 1999.
20. Zhang (Chuanjun) et Vahid (Frank). – Using a victim buffer in an application-specific memory hierarchy. *In : DATE '04 : Proceedings of the conference on Design, automation and test in Europe*. pp. 220–225. – CNIT La Defese, Paris, France, feb 16-20 2004.