# Ordonnancement sous contraintes de mémorisation : une optimisation efficace des ressources lors de la synthèse d'architecture

Gwenole Corre, Nathalie Julien, Eric Senn, Eric Martin

# Ordonnancement sous contrainte de mémorisation : une optimisation efficace des ressources lors de la synthèse d'architecture.

**Gwenolé Corre, Nathalie Julien, Eric Senn, Eric Martin**
LESTER, Université de Bretagne Sud
Centre de recherche - BP 92116 - 56321 LORIENT Cedex
mailto : prenom.nom@univ-ubs.fr
Tel : +33 (0)2 97 87 45 28 - Fax : +33 (0)2 97 87 45 00

**Résumé**
Les systèmes supportant des applications de traitement du signal et de l'image manipulent de plus en plus de données. Cela entraîne une utilisation intensive de la mémoire qui devient le point critique du système ; la mémoire limite les performances et représente une proportion importante de la consommation globale. Dans le contexte de la synthèse d'architecture, nous nous intéressons au développement d'une stratégie de conception et d'optimisation de la consommation des unités de mémorisation. Nous présentons, en particulier, la phase d'ordonnancement intégrant des contraintes de mémorisation définies en amont de la synthèse d'architecture. Ce nouvel ordonnancement, à partir d'un mapping mémoire prédéfini, permet de réduire la consommation de la partie traitement tout en garantissant une architecture mémoire cohérente.

Mots Clés: synthèse haut niveau, consommation, mémoire.

## 1 Introduction

GAUT est un outil de synthèse d'architecture dédié aux applications de Traitement du Signal et de l'Image (TDSI) possédant une contrainte de temps réel spécifiée par la cadence d'arrivée des données de l'application [1]. L'architecture synthétisée et optimisée repose sur un modèle de cœur de processeur de traitement du signal qui exploite les parallélismes spatiaux et temporels des applications pour l'accélération des traitements. La méthodologie de conception passe par différentes étapes : spécification comportementale, extraction du parallélisme, matérialisation et optimisation des différentes unités fonctionnelles puis description complète de l'architecture au niveau RTL. Comme les différentes unités fonctionnelles interagissent, elles sont hiérarchisées lors de la conception ; ainsi l'outil de synthèse GAUT s'intéresse en premier à la synthèse des unités de traitement. Une première contribution à l'amélioration de cet outil a été la prise en compte des aspects consommation lors de la synthèse de l'unité de traitement pour aboutir à l'outil GAUT_LP ; cependant la partie mémorisation n'était pas intégrée jusqu'alors [2]. Or, la surface et la consommation de l'unité de mémorisation deviennent prédominantes dans les applications de TDSI ; on sait notamment qu'il est prévu, en 2011, que 90% de la surface soit dédiée à la mémoire [3]. Dans le cadre d'une stratégie globale, nous nous proposons d'intégrer les contraintes et l'optimisation de la mémorisation dans l'outil GAUT.

Dans cet article, nous développons l'introduction de contraintes liées au mapping mémoire dans l'ordonnancement des opérations de l'unité de traitement. Nous présentons tout d'abord, en section 2, un état de l'art sur les techniques de conception des unités mémoire puis sur les techniques d'ordonnancement. Dans la section 3, nous explicitons la méthodologie que nous allons mettre en œuvre pour obtenir une unité de mémorisation optimisée. Les nouvelles contraintes et l'ordonnancement associé sont présentés et illustrés par un exemple en section 4. Quelques résultats sont détaillés en section 5 avant de conclure sur les travaux futurs.

## 2 Etat de l'art

### 2.1 Stratégie de conception

L'optimisation de la consommation des mémoires, au niveau architectural, vise d'une part à diminuer leur taille (ce qui limite la puissance statique) et d'autre part à limiter les transferts et les transitions sur les bus (ce qui réduit la puissance de commutation). Il existe différentes techniques d'optimisation dans la littérature, qui peuvent être classées suivant trois axes.

Le premier s'intéresse à la conception de hiérarchie et d'architecture mémoire pour des applications dont les séquences d'accès sont connues a priori. De nombreuses méthodologies proposent des techniques permettant de sélectionner une architecture faible consommation utilisant des mémoires caches ou des mémoires SRAM dédiées ou scratch-pads [4]. Le choix de l'architecture mémoire est effectué par rapport aux estimations en vitesse, surface et consommation obtenues à l'aide de modèles analytiques.

Le deuxième considère une hiérarchie et une architecture fixe ; la réduction de la consommation est réalisée par transformation du code. Ces transformations vont permettre de réduire les besoins en accès à la mémoire, les séquences d'accès mémoire ou le nombre de transitions sur les adresses. De nombreuses techniques d'ordonnancement des accès mémoire [5] et d'assignation des données ont été

développées en s'orientant vers l'optimisation de la consommation.
Le troisième optimise conjointement l'architecture mémoire et le code de l'application. Il est notamment développé à l'IMEC sous la dénomination de DTSE (Data Transfert and Storage Exploration) [6]. Cependant le temps nécessaire à l'obtention d'une solution optimale est relativement important car de nombreuses étapes ne peuvent être automatisées et requièrent l'expertise d'un concepteur avisé.

### 2.2 Techniques d'ordonnancement

Un des facteurs importants qui affecte le coût d'une architecture mémoire est l'ordre relatif des accès mémoire contenus dans la spécification. La plupart des méthodes d'ordonnancement prenant en compte la partie mémoire essaient de réduire le coût mémoire en estimant les besoins en terme de nombre de registres pour un ordonnancement donné (seulement pour les scalaires). Les quelques exceptions sont les "stream schedulers" [7], les "rotations schedulers" [8], les "percolation schedulers" [9]. Ils ordonnancent les accès et les opérations comme un contexte de compilation, en incluant des modèles temporels précis des accès mémoires afin d'améliorer les performances. Ces techniques tentent de réduire le nombre global d'accès mémoire et le nombre d'accès en parallèle. Cependant, elles ne tiennent pas compte des données accédées simultanément afin de faciliter la tâche postérieure d'assignation de registres et mémoires.
Dans le contexte de la synthèse d'architecture, quelques méthodes d'ordonnancement prennent en compte les aspects mémoire. Dans [10] les accès mémoires sont représentés comme étant des opérations multi-cycles dans un CDFG. Les nœuds mémoires sont ordonnancés comme des nœuds opérations en prenant en compte les conflits d'accès aux données. Les travaux développés par [11], un premier ordonnancement de type "force directed scheduling" est réalisé sur un DFG, les accès mémoire sont ensuite ré-ordonnancés en fonction d'une sélection et allocation mémoire visant à réduire le coût global de la mémorisation.
Notre étude se propose de développer une méthodologie visant à intégrer les contraintes liés à la mémorisation dans le flot de synthèse de l'outil GAUT Low Power. Nous proposons également d'intégrer les durées de vie des variables pour leur mémorisation ce qui n'est pas réalisé dans [10] et de réduire la complexité de l'ordonnancement par rapport à la technique développée en [11]. L'objectif est de trouver une solution architecturale répondant aux contraintes de l'application et à un compromis entre le temps de conception et la qualité de la solution.

## 3 Stratégie globale

La stratégie de conception (Figure 1) se décompose en trois phases : la détermination d'une hiérarchie mémoire et d'une distribution des données, la synthèse d'architecture sous contraintes de mémorisation et la définition des générateurs d'adresses.

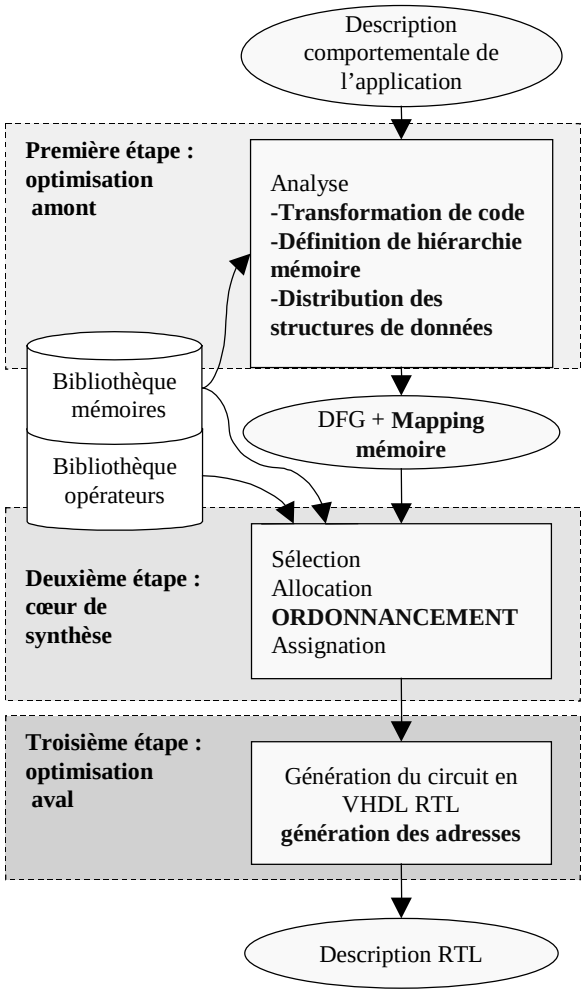


Figure 1 : stratégie d'intégration de la mémoire autour de GAUT

### 3.1 Détermination de la hiérarchie et de la distribution

La première étape se situe en amont de la synthèse d'architecture et de la compilation ; elle comporte des méthodes indépendantes de l'architecture ciblée. Elle commence par l'analyse du code source afin de modéliser l'application sous forme de graphes. Des transformations de code permettent éventuellement d'optimiser le nombre de transferts et le nombre de transitions sur les bus d'adresses. On peut ensuite définir la hiérarchie mémoire en nombre de niveaux de hiérarchie, en taille et largeur de chaque niveau et en nombre de ports : les choix s'effectuent par rapport à des fonctions de coûts s'appuyant sur des modèles caractérisant les mémoires en fonction de leurs paramètres architecturaux, algorithmiques et technologiques (librairie mémoire). Une fois la hiérarchie mémoire décidée, on peut procéder à la distribution des structures de données qui consiste à allouer une mémoire à chaque structure en se basant sur ses localités temporelles et spatiales. Le but est de favoriser le stockage des structures les plus souvent utilisées et de limiter les connexions vers l'unité de traitement et les niveaux hiérarchiques éloignés. L'information finale sur cette distribution parmi les niveaux de hiérarchie est appelée *mapping mémoire*.

### 3.2 Synthèse d'architecture avec contrainte de mémorisation

L'étape de synthèse consiste, à partir du *mapping mémoire* prédéfini et d'une bibliothèque de composants mémoire, à réaliser les phases d'allocation et d'ordonnancement du cœur de la synthèse architecturale de l'unité de traitement. Ces nouvelles contraintes de mapping mémoire seront principalement prises en compte lors de l'ordonnancement des opérations. En particulier, cet ordonnancement va permettre de réaliser un compromis entre les optimisations de l'unité de traitement et l'unité de mémorisation ; c'est ce que nous présentons ici (voir section 4).

### 3.3 Placement des données et définition des générateurs d'adresses

Après la synthèse d'architecture, les optimisations consistent à définir, pour chaque niveau de hiérarchie et en fonction de leur architecture, les générateurs d'adresses associés. Avant de réaliser les générateurs d'adresses, il faut s'intéresser au placement des données intermédiaires non préalablement affectées aux mémoires. Ce sont des données temporaires, engendrées par le traitement, qui ont une durée de vie suffisamment importante pour ne pas être stockées en registre dans l'unité de traitement. Il faut donc effectuer la distribution et le placement ces données dans les bancs mémoires. Une fois toutes les données placées dans l'unité de mémorisation, il faut définir les générateurs d'adresses sous forme de compteurs ou de FSM. Des techniques d'encodage peuvent être utilisées afin de diminuer le nombre de transitions sur les bus d'adresses.

Notre stratégie permet de réaliser des optimisations de l'unité de mémorisation tout au long de la synthèse d'architecture et non plus aval de la synthèse de l'unité de traitement. Cette approche augmente les possibilités d'optimisation de l'unité de mémorisation car elles ne sont plus contraintes par l'architecture de l'unité de traitement.

## 4 Méthode d'ordonnancement proposée

### 4.1 Définitions des contraintes de mapping mémoire

Avant de contraindre le cœur de synthèse de l'outil GAUT, il faut définir deux choses : une pré-assignation des données en mémoire et une caractérisation des composants mémoire dans lesquels ont été assignées les structures de données (sous la forme de bibliothèque de composants mémoire).

La pré-assignation des données en mémoire sera exprimée par le *mapping mémoire* qui est un fichier de contraintes contenant les structures à placer en mémoire, le numéro de bancs dans lequel elles sont distribuées et éventuellement leurs adresses (normes représentées en Figure 2 et 3).

```
X,    numbancs,    adresse, taille;
H,    numbancs,    adresse, taille;
```

Figure 2 : Exemple de fichier de contraintes pour LMS 4 points.

Les structures de données peuvent être éclatées de manière à contrôler la distribution et le placement des données comme représenté en Figure 3.

```
x(0),    numbancs,    adresse, ;
x(1),    numbancs,    adresse, ;
x(2),    numbancs,    adresse, ;
x(3),    numbancs,    adresse, ;
h(0),    numbancs,    adresse, ;
h(1),    numbancs,    adresse, ;
h(2),    numbancs,    adresse, ;
h(3),    numbancs,    adresse, ;
```

Figure 3 : Exemple de fichier de contrainte avec structures éclatées.

Pour caractériser les composants mémoire, il faut d'abord définir les paramètres qui seront utiles lors du cœur de la synthèse d'architecture. Les principaux paramètres sont

-les temps d'accès mémoire.
-les modes d'accès (burst, aléatoire).
-la taille mémoire.
-le nombre et type de ports.
-le coût (surface, puissance par mode d 'accès mémoire).

Une fois ces paramètres déterminés les composants mémoire peuvent être décrits dans les bibliothèques dont un exemple est fourni en Figure 4.

```
component ram_16x1024
generic
(
    time    : integer   := 5 ;
    area     integer    := 4265;
    consovar            : integer   := 348;
    consocst            : integer   := 36;
    size                : integer   := 16384
);
port
(
    a       :in td_logic_vector(9 downto 0);
    clk     :in  std_logic;
    we      :in  std_logic;
    oe      :in  std_logic;
    in      :in std_logic_vector(nb_bit-1 downto 0);
    out     :out std_logic_vector(nb_bit-1 downto 0)
)
end component;
```

Figure 4 : exemple de caractérisation de mémoire ram simple port en bibliothèque.

Dans la bibliothèque mémoire, figure 4, pour chaque composant mémoire, un temps d'accès en ns (time) est défini. La taille en nombre de bits (size) et la surface en nombre de CLBs (area) sont intégrées dans les paramètres ainsi que la puissance dynamique en mW/Mhz (consovar) et la puissance statique en mW (consocst). Les ports de la mémoire sont déclarés, ceci permet de définir le nombre et le type de ports de chaque mémoire. Les bibliothèques utilisées dans l'outil sont actuellement en cours de caractérisations ce

qui permettra, à terme, de réaliser une estimation du coût de la partie mémoire et d'intégrer ce coût lors de la synthèse.

### 4.2 Ordonnancement sous contrainte de mémorisation.

Dans l'ordonnancement proposé, Figure 5, les données en entrée des opérations peuvent provenir de mémoires internes contenant les données nécessaires à l'unité de traitement. Plusieurs opérations peuvent accéder à la même mémoire à un instant donné. La liste des opérations exécutables (*Ops_exe*) est d'abord triée suivant la mobilité des opérations ; le choix de l'opération à ordonnancer s'effectue maintenant suivant un critère d'accessibilité des opérations aux variables. Ce critère est défini de la manière suivante, toutes les opérations exécutables dont les nœuds variables associés sont stockés en registre ou dans une mémoire accessible vérifient le critère d'accessibilité des opérations aux variables. Une mémoire est dite accessible, si à l'instant de l'accès elle possède au moins un port d'accès libre. Donc, pour chaque opération sollicitant des données placées en mémoire, et suivant sa priorité, on vérifie que les données à accéder ne sont pas placées dans des bancs mémoires occupés. Lorsqu'une opération exécutable vérifie le critère d'accessibilité, il faut remettre à jour la liste des mémoires libres. Toutes les opérations ne répondant pas au critère d'accessibilité sont retirées de la liste des opérations exécutables. Par exemple, si une mémoire est occupée, alors une opération exécutable nécessitant un accès à cette mémoire, bien que prioritaire dans la liste, ne pourra être ordonnancée. Les opérations exécutables sont ordonnancées en fonction des opérateurs arithmétiques disponibles (*Opr_libre*). Une fois les opérations ordonnancées, il faut remettre à jour la liste des opérations exécutables, la listes des opérateurs libres et la listes des mémoires accessibles.

```
cycle =0;
Faire
    Trier la liste des Ops_exe                          (mobilité)
    Parcourir la  liste des Ops_exe
        si (Ops_courante = Ops_accessible) alors        (accessibilité)
            conserver Ops_courante dans la liste Ops-exe
            mettre à jour la liste des mémoire accessible
        sinon
            retirer Ops_courante de la liste Ops_exe
        fin si
        Ops_courante = Ops_suivante
    tant que Ops_courante ≠ ∅
    Ordonnancer Ops_exe  en fonction des Opr_libre
    Mettre à jour la liste des mémoires accessibles
    Mettre à jour les listes des Opr_libres et des Ops_exe
cycle = cycle+1;
Tant que toutes les opérations ne sont pas ordonnancées
```

Figure 5 : ordonnancement avec contrainte de mapping mémoire

Illustrons l'ordonnancement mis en œuvre par un exemple simple.

### 4.3 Exemple

Soit la spécification comportementale suivante (en VHDL) et son DFG associé, nous allons comparer l'influence de l'ordonnancement avec contrainte mémoire par rapport à l'ordonnancement classique de l'outil GAUT.

```
entity test is
port (a, b, c, d, e, f:in integer;
      x, y, z:out integer);
end add;
architecture test_arch of test is
begin
  constant latency := 40 ns : time;
Process
   variable tmp, tmp2 : integer;
begin
   tmp :=a+b;
   z := a + f;
   x := tmp + c;
   tmp2 := d+e;
   y <= tmp2 + f;
   wait for latency;
end process;
end test_arch;
```

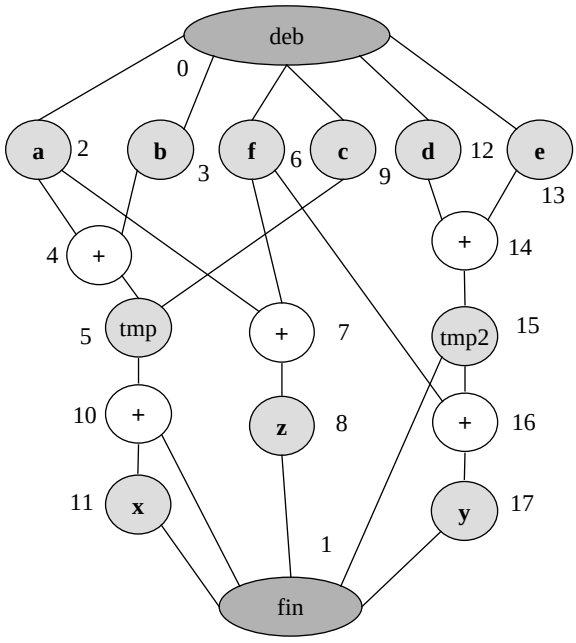


Figure 6 : description et DFG associé

Nous allons réaliser une distribution des données et définir un fichier de contrainte de mapping mémoire de façon à contraindre l'ordonnancement. Grâce à ce fichier de contraintes (Figure7), le DFG va pouvoir être annoté pour considérer les contraintes d'accès mémoire lors de la phase d'ordonnancement (Figure 8).

```
a,      mem1,   0,      ;
d,      mem1,   1,      ;
x,      mem1,   2,      ;
y,      mem1,   3,      ;
z,      mem1,   4,      ;
b,      mem2,   0,      ;
e,      mem2,   1,      ;
f,      mem3,   0,      ;
c,      mem3,   1,      ;
```

Figure 7 : Fichier de contraintes de mapping mémoire

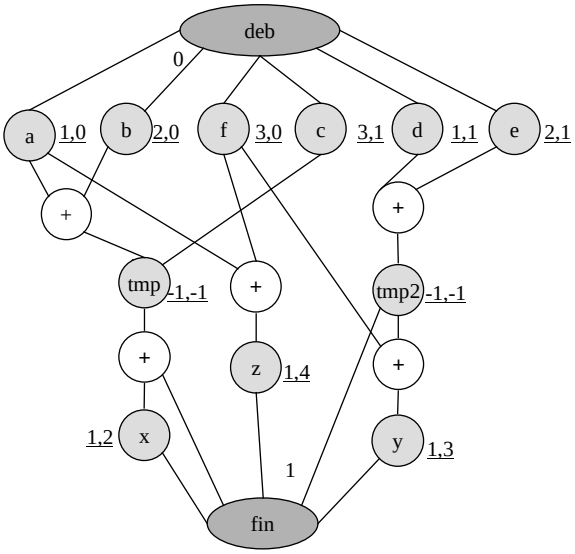


Figure 8 : DFG annoté

La phase d'ordonnancement prend en compte ce DFG annoté ; les valeurs soulignées dans le graphe correspondent respectivement au numéro de la mémoire et à l'adresse de la donnée. Ces valeurs sont mises à –1 lorsque les données ne sont pas placées en mémoire. Nous allons comparer les résultats de l'ordonnancement classique et de l'ordonnancement avec contraintes de mapping mémoire.

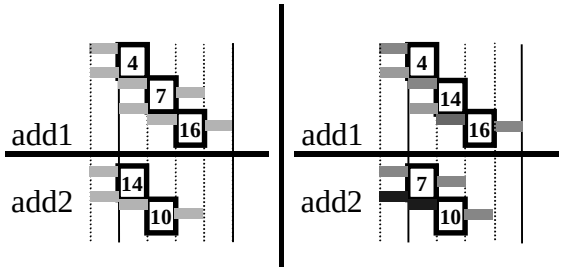


Figure 9 : ordonnancement classique vs ordonnancement mapping

Pour cet exemple, la période d'horloge est de 10 ns. De plus, les temps des opérateurs sélectionnés sont des multiples de ce temps d'horloge et dans l'exemple l'addition et les accès mémoire s'effectuent également en 10 ns. Pour respecter la contrainte de cadence fixée dans la spécification VHDL, 40 ns, l'outil sélectionne deux additionneurs. Dans l'ordonnancement classique, seule la mobilité des nœuds opérations est considérée : donc, parmi les opérations exécutables, 7, 14 et 4, les opérations les plus prioritaires, 4 et 14, sont ordonnancées au premier cycle et les opérations restantes sont ordonnancées de la même façon sur les cycles suivants. Avec les contraintes de mapping, les opérations 4 et 14 accèdent aux mêmes ressources mémoire ; l'opération 14 est donc retardée et 4 et 7 sont ordonnancées.

Cet ordonnancement, en limitant les ressources nécessaires à la mémorisation des données à l'intérieur de l'unité de traitement et leur utilisation, permet de réduire la puissance de la partie traitement. Nous illustrons cette baisse de consommation par quelques exemples dans la section suivante.

## 5 Expériences et résultats

Pour valider le nouvel ordonnancement et définir son influence sur l'architecture de l'unité de traitement, nous avons effectué plusieurs synthèses avec GAUT pour une application de FFT sur 32 points. Nous comparons les résultats de synthèse en fonction de différents mapping mémoire : sans contrainte de mapping (nomap) puis avec des contraintes de mapping introduisant différentes distributions de données en mémoire (map1 à map6). Dans le tableau de résultats suivant, nous avons réalisé différentes synthèses, nomap est le résultat d'une synthèse sans contrainte mémoire. Les échantillons sont placés dans un seul banc dans map1. Pour les autres synthèses, les échantillons sont placés dans deux bancs. Les 16 premiers dans le premier banc, les 16 derniers dans le banc2 pour map2. Un placement 8 par 8 est effectué pour map3, 4 par 4 pour map4, 2 par 2 pour map5 et enfin les échantillons pairs dans un banc, les échantillons impairs dans le second pour map6. Le nombre de ressources, la surface et le temps sont définis par l'outil GAUT. La consommation en puissance est déterminée par l'outil Xpower après synthèse sur un FPGA virtexE xcv400e_8bg432 (Tableau I). Seule la puissance dynamique est considérée car la puissance statique dépend uniquement du composant Virtex utilisé. Le nombre de ressources arithmétiques est le même pour toutes les solutions, à savoir, un multiplieur, un additionneur et un soustracteur.

| | Nombre de ressources | | | | Caractéristiques | | | |
|---|---|---|---|---|---|---|---|---|
| mapping | tri-states | registres | mux | demux | S (CLBs) | T (ns) | P (mW) | Δ(P) |
| nomap | 23 | 13 | 9 | 5 | 736 | 5810 | 37,17 | --- |
| map1 (1 banc) | 15 | 10 | 7 | 3 | 592 | 7660 | 22,73 | -38% |
| map2 (2 bancs) | 12 | 10 | 7 | 3 | 592 | 5810 | 34,35 | -7% |
| map3 (2 bancs) | 15 | 10 | 5 | 5 | 624 | 5810 | 21,85 | -41% |
| map4 (2 bancs) | 17 | 10 | 6 | 5 | 640 | 5810 | 17,77 | -52% |
| map5 (2 bancs) | 11 | 10 | 5 | 2 | 528 | 5810 | 21,63 | -41% |
| map6 (2 bancs) | 13 | 10 | 6 | 4 | 608 | 5810 | 19,26 | -48% |

Tableau 1 : synthèse d'un FFT 32 points pour différentes contraintes mémoire

La solution consistant à placer toutes les données dans un banc unique, map1, permet de réduire la puissance mais est pénalisante au niveau du temps. Une solution plus réaliste consiste en l'utilisation de deux bancs mémoire : une première possibilité, map2, consiste à placer les 16 premiers échantillons dans un banc, les 16 suivants dans l'autre et permet, sans pénalité de temps, d'optimiser la surface. La différence de ressources entre les différentes solutions s'explique par le fait de la mise en mémoire des échantillons qui permet à l'outil de mieux optimiser l'étape de fusion de registres. Cependant notre approche permet surtout de déduire le mapping mémoire optimum. En effet, par rapport à map 2, qui serait la pratique courante, compte tenu du fait de la sélection d'un opérateur multiplieur, d'un additionneur et d'un soustracteur, on constate que la solution map 4, qui consiste à placer les échantillons 4 par 4, diminue la puissance de 51% sans pénalité de temps. Cette différence peut s'expliquer par le fait d'optimiser l'ordonnancement sur un calcul de papillons en fonction du placement dans les bancs.

## 6 Conclusion

L'ordonnancement actuel de l'outil GAUT Low Power s'inscrit dans une politique d'optimisation de la partie opérative des algorithmes. Les différentes unités fonctionnelles sont fortement contraintes par l'unité de traitement ; de ce fait elles sont difficilement optimisables. La définition d'un mapping mémoire en amont de l'outil et la prise en compte de ce mapping au cœur de la synthèse va permettre de réaliser un compromis entre optimisation de l'unité de traitement et optimisation de l'unité de mémorisation. Par ailleurs, nous avons montré que l'ordonnancement mis en œuvre permettait d'obtenir un gain intéressant du point de vue de la consommation en puissance. Il permet de déterminer le meilleur mapping mémoire et, ainsi, de guider l'utilisateur pour placer les données correctement dans une architecture mémoire déjà définie.

Un axe de développement pourra être de mettre en concurrence ou d'associer le critère de stabilité de données, développé dans [2], qui permet de réduire l'activité à en entrée des opérateurs, et le critère d'accessibilité des opérateurs aux variables, en intégrant le coût en puissance des accès mémoire, pour obtenir une unité de traitement et une unité de mémorisation optimisées au niveau de la consommation.

## Bibliographie


[1] http://lester.univ-ubs.fr:8080/

[2] S. Gailhard, "conception d'architectures à faible consommation," *Thèse, LESTER, université de Rennes1*, janvier 1999.

[3] http://public.itrs.net/

[4] P. R. Panda, N. Dutt, A. Nicolau, "Memory issues in embedded System-On-Chip, optimisation and exploration," *Kluwer Academic Publisher*,1999. ISBN 0-7923-8362-1.

[5] R. Saied, C. Chakrabarti, "Scheduling for Minimizing the Number of Memory Accesses in Low Power Applications," *in Proceedings of VLSI Signal Processing*, pp. 169-178, October 1996.

[6] F. Catthoor et al, "Custom memory management methodology : Exploration of memory organisation for embedded multimedia system design," *Kluwer Academic Publisher*,1998. ISBN 0-7923.

[7] W.Verhaegh, P.Lippens, E.Aarts, J.Korst, J.van Meerbergen, A.van der Werf, "Improved Force-Directed Scheduling in High-Throughput Digital Signal Processing", *IEEE Trans. on Computer-aided design*, Vol.14, No.8, Aug. 1995.

[8] N.Passos, E.Sha, L-F.Chao, "Multi-dimensional interleaving for time-and-memory design optimization", In Proc. *IEEE Int. Conf. On Computer Design*, pp.440-445, Oct. 1995.

[9] A. Nicolau and S. Novack. Trailblazing," A hierarchical approach to percolation scheduling," In *Proc of ICPP*, St. Charles, IL, 1993.

[10] H. Ly, D. Knapp, R. Miller, D. McMillen,"Scheduling using Behavioral Templates," In *Proceeding of Design Automation Conference*, June 1995.

[11] J. Seo, T. Kim, P. Panda ,"An Integrated Algorithm for Memory Allocation and Assignment in High-Level Synthesis," In *Proceeding of Design Automation Conference*, pp 608-611, June 2001.