Conception et systèmes embarqués complexes

Master 2004

Antoine Fraboulet, Tanguy Risset
antoine.fraboulet@insa-lyon.fr, tanguy.risset@ens-lyon.fr
Lab CITI, INSA de Lyon, Lab LIP, ENS de Lyon

Architecture mémoire et optimisation

Plan

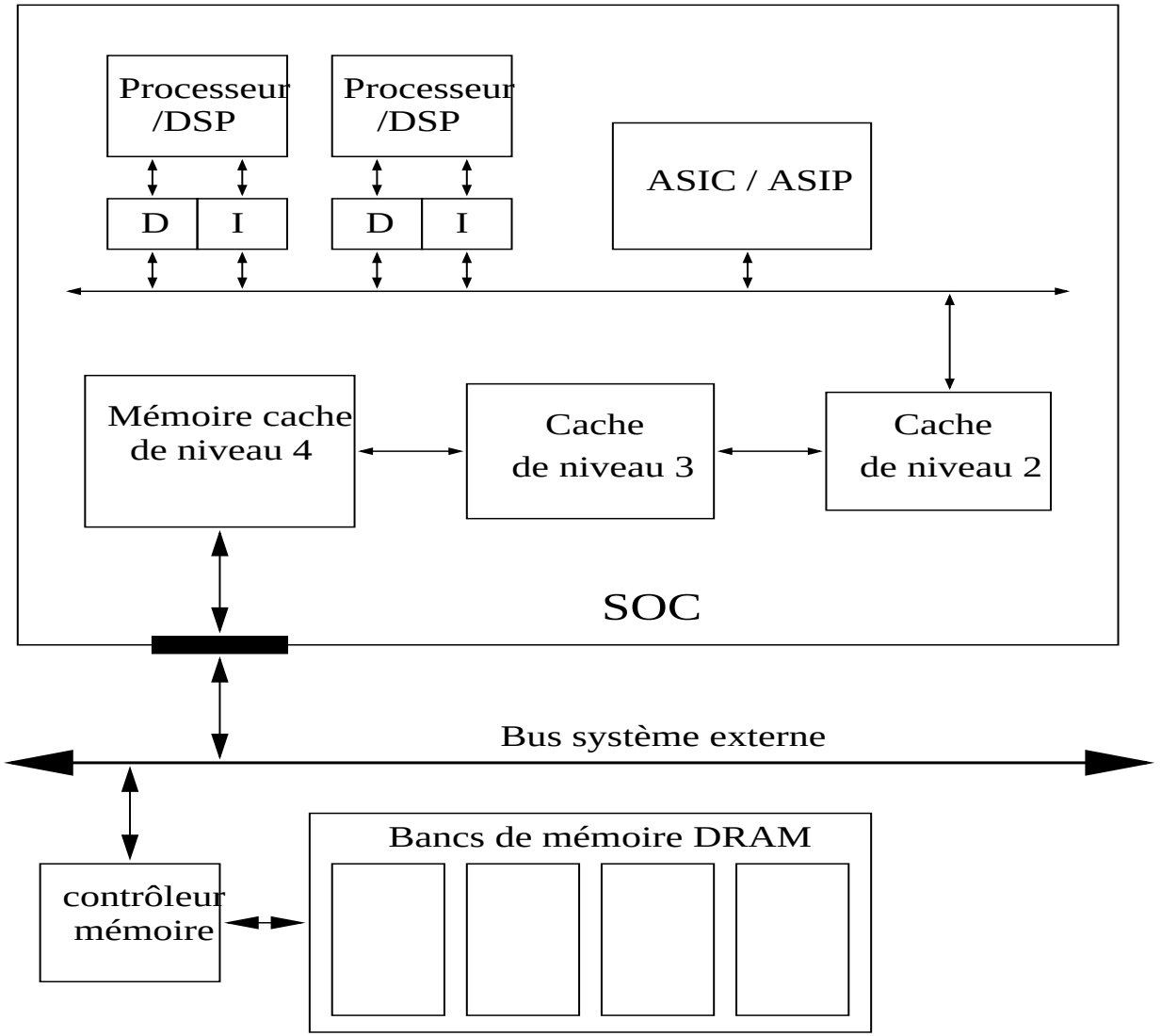
1. Codesign et architecture
2. Transformation de code et optimisation mémoire
3. Mise en pratique et intégration des transformations

Motivations et besoins

- Les principales sources de consommation électrique dans un système sont réparties entre
  - Calculs
  - Communications
  - Stockage mémoire

- Les systèmes embarqués traitent des volumes de données de plus en plus importants
  - La proportion de mémoire dans les SoC devrait passer à 70% en 2005 . . . et à 94% en 2014
  - Les performances des mémoires ne suivent pas celles des processeurs et des réseaux

Architectures des mémoires



Architecture des mémoires

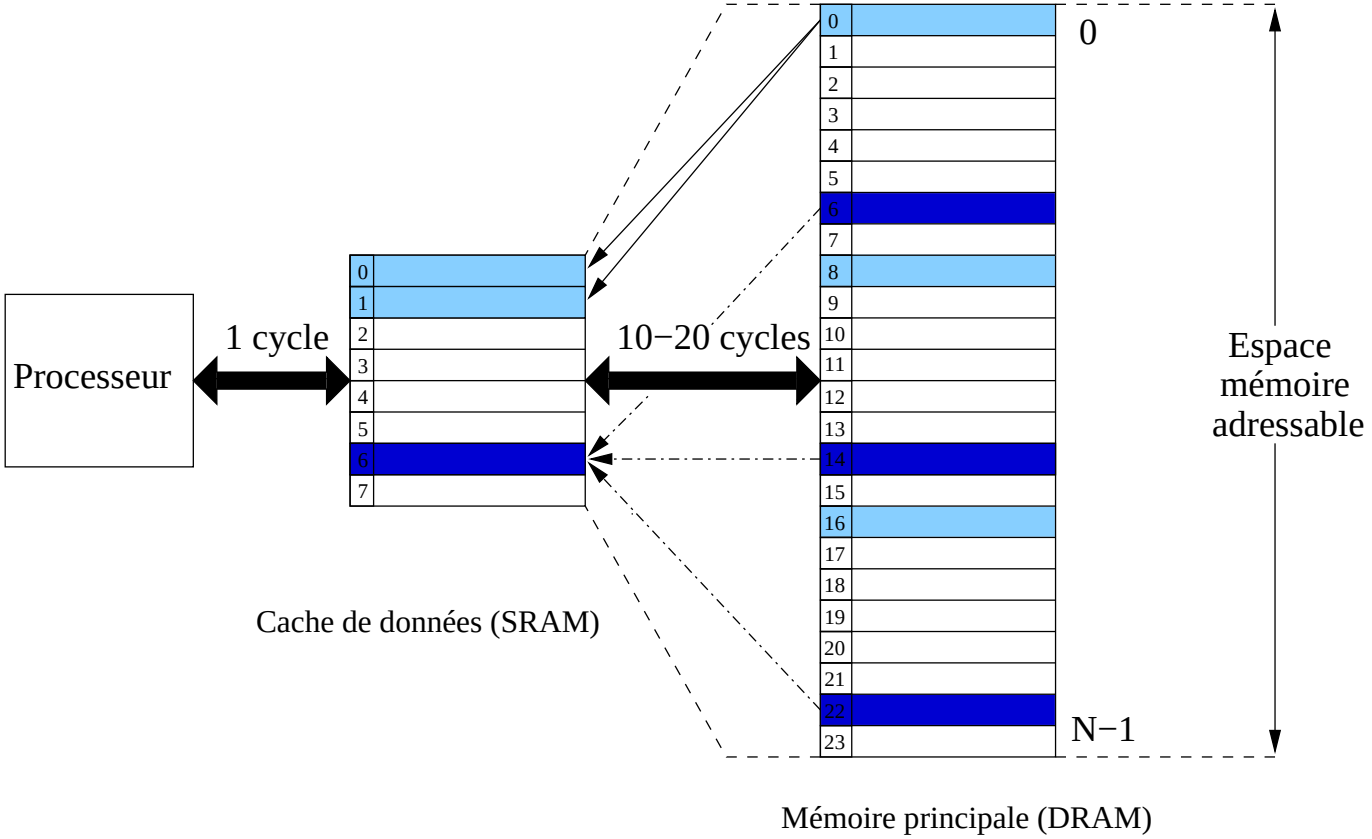
- Types de mémoires
  - Mémoires statiques (SRAM) : petites, rapides, consommatrices, peu denses
  - Mémoires dynamiques (DRAM) : grandes, lentes, très denses, transactions chères
- Critères
  - Taille des différents modules
  - Nombre de transferts entre les niveaux

Coût des accès

Le coût dépend du niveau auquel on accède (Ko et Balsata) : exemple sur un hiérarchie à 4 niveaux

- puissance moyenne dissipée sur puce
  - niveau 1 : 150 mW
  - niveau 2 : 300 mW
  - niveau 3 : 700 mW
- quatrième niveau, mémoire externe DRAM de grande capacité
  - 12,71 W par transaction

Mémoires Caches



- Premier accès à une donnée obligatoire
- Éviter les transferts redondants
  - Limiter les défauts de capacité
  - Limiter les défauts de conflits

Caches matériels : fonctionnement

1. Adressage
2. Défauts de cache
3. Remplacements
4. Recopies en mémoire

Caches matériels : adressage

L'architecture matérielle des caches permet d'avoir une grande variété de fonctionnement :

- **Cache à adressage direct** : un bloc est recopié dans le cache à une adresse unique selon sa provenance (modulo sur l'adresse).
- **Cache associatif par ensemble** : un bloc peut être copié dans un ensemble de lignes (2, 4 ou 8 voies en général).
- **Cache totalement associatif** : chaque bloc de mémoire peut être copié dans n'importe quelle ligne de cache.

La complexité de gestion du cache croit en fonction de son degré d'associativité (surface et consommation électrique).

Caches matériels : défauts de cache

Un défaut de cache correspond au chargement d'une donnée en mémoire vers la mémoire cache pour y être utilisé.

- **défaut obligatoire** : la première fois qu'une donnée est accédée. Ces défait de cache ne peuvent être évités
- **défaut de capacité** : cache de taille non suffisante. Des blocs doivent être remplacés lors de l'exécution du programme.
- **default de conflit** : blocs se projetant dans les mêmes lignes de cache. Ne se produit pas dans un cache totalement associatif.

Caches matériels : remplacement

Choix de l'élément à remplacer lors d'un conflit:

- **l'élément le plus ancient** : least recently used (LRU)
- **l'élément le moins utilisé** : least frequently used (LFU)
- **ordre d'ancienneté** : first in first out (FIFO)
- **aléatoire** : favorise une allocation uniforme

Caches matériels : ré-écritures

Lorsqu'une ligne doit être remplacée ou lors d'une écriture en mémoire le cache doit assurer la cohérence avec la mémoire principale.

- Cas où la ligne est présente dans le cache
  - Écriture simultanée : une écriture est effectuée vers la mémoire à chaque écriture dans le cache (*Hit Write through*)
  - Les écritures sont différées et mise en attente d'un remplacement (*Hit Write Back*)
- Cas où la ligne n'est pas présente
  - Lecture de la ligne complète et modification (*Miss Fetch on Write*)
  - Écriture sans chargement de la donnée (*Miss Write Around*)

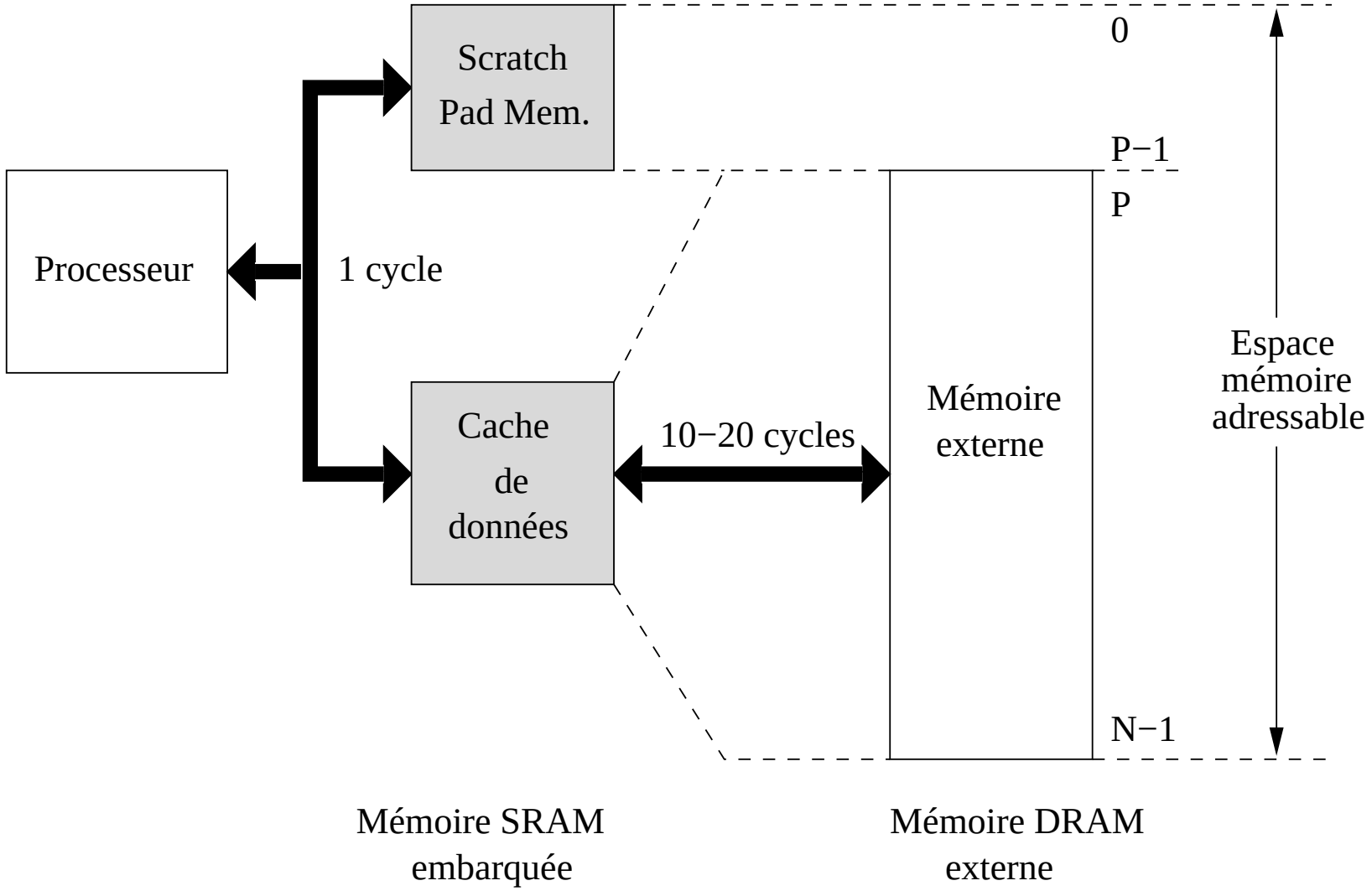
Caches logiciels

Les caches matériels restent utilisés dans la majorités des cas. Cependant dans certains cas les concepteurs préfèrent maîtriser la gestion des données. Ils utilisent alors un cache géré de façon logicielle : **scratch pad memory**.

- Permet d'adapter le comportement du cache de façon dynamique à l'application
- Gestion plus précise de la cohérence des niveaux de mémoire (contexte multiprocesseur)

- Nécessite une plus grande attention de la part du programmeur
- Chaîne de compilation adaptée

Cache logiciel



Création d'une hiérarchie mémoire dédiée

La mise en place des mémoires caches dépend beaucoup de l'application cible.

- La performance et la consommation ne sont pas évaluées de la même façon
  - performance : chemin critique de l'application
  - consommation : activité moyenne des niveaux
- Expérimentations (Schiue et Chakrabarti) sur une application MPEG
  - cache de 64 octets avec des lignes de 4 octets associatives par 8 : décompression en 142000 cycles pour une consommation d'énergie de 283μJ.
  - architecture la plus performante trouvée par les auteurs utilise un cache de 512 octets avec des lignes de 16 octets associatives par 8. Effectue les mêmes calculs en 121 000 cycles mais dépense une énergie de 1110μJ

Création d'une hiérarchie mémoire dédiée

- Compromis à trouver entre vitesse d'exécution et consommation
- L'espace de recherche est discrétisé pour permettre une recherche dans les solutions possibles.
  - taille totale de la mémoire embarquée nécessaire
  - partitionnement de cette mémoire entre
    - mémoire *scratch pad*, caractérisé par sa taille
    - cache de données : taille et taille des lignes de cache
  - En complément il faut une **allocation mémoire** permettant de placer les objets aux adresses correspondant à leur utilisation optimale.

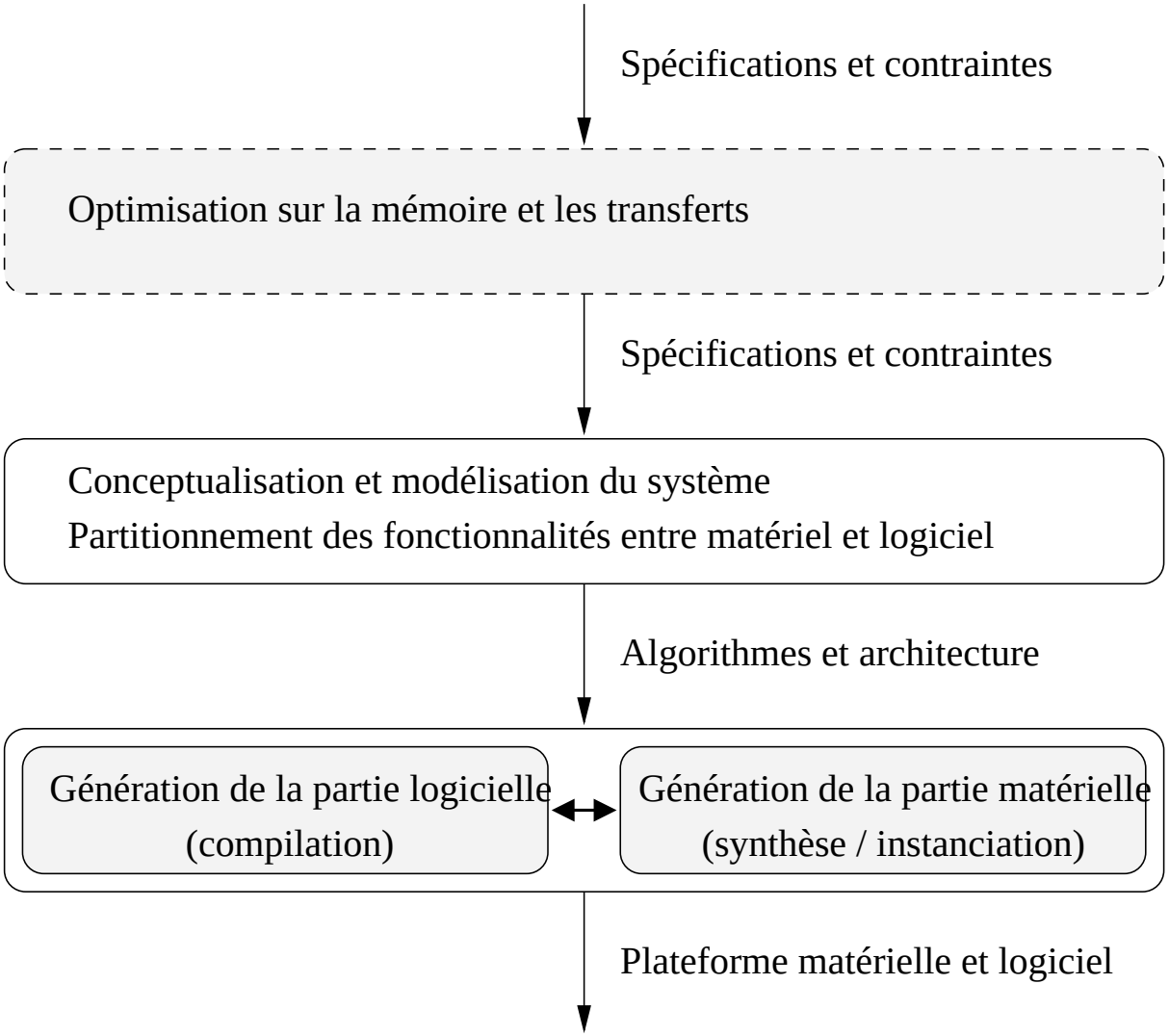
Cache d’instructions

Les caches d’instruction permettent de sauvegarder le code de l’application lorsque celui-ci est exécuté de manière répétitive

- cache d’instructions prédécodées : évite l’étape de décodage
- cache de boucles : permet de garder le corp de boucle
- cache de fonctions : conserve le corps d’une fonction

Les caches d’instruction permettent de limiter la bande passante entre le processeur et la mémoire. Leur efficacité est très liées à la qualité du binaire produit par l’éditeur de lien (placement des fonctions en mémoire).

Codesign et architecture



Transformation pour une architecture définie

- Analyse de programme et transformations
- Accès aux données et critères d'optimisation
- Transformations pour une hiérarchie matérielle
- Optimisation conjointe de l'architecture et du code : DTSE

Analyse de programme et transformations

- Transformations interactives pour la conception, intégrées dans une méthodologie ou source à source
- Mesure de la localité à gros grains
  - prise en compte globale des nids de boucles
  - dépendances prises sur les tableaux entiers
- Mesure de la localité à grains fins
  - prise en compte des dépendances à l'intérieur des boucles
  - transformations sur les boucles (alignement, décalage, torsion, transformations unimodulaires)

Analyse de programme

On s'intéresse ici aux dépendances entre les données utilisées dans les opérations 01 et 02 d'un programme:

- *dépendance de flot*: 02 est dépendante de 01 par une dépendance de flot si 02 utilise en lecture un emplacement mémoire écrit par 01 et si aucune autre opération, dans l'ordre séquentiel, n'écrit à cet emplacement mémoire entre 01 et 02.
- *anti-dépendance*: 02 est dépendante de 01 par une anti-dépendance si 01 utilise en lecture un emplacement mémoire écrit par 02 et si aucune autre opération, dans l'ordre séquentiel, n'écrit à cet emplacement mémoire entre 01 et 02.
- *dépendance de sortie*: 02 est dépendante de 01 par une dépendance de sortie si 02 et 01 sont des écritures consécutives dans le même emplacement mémoire.
- *dépendance d'entrée*: 01 et 02 ont une dépendance d'entrée si 02 et 01 sont des lectures consécutives dans le même emplacement mémoire.

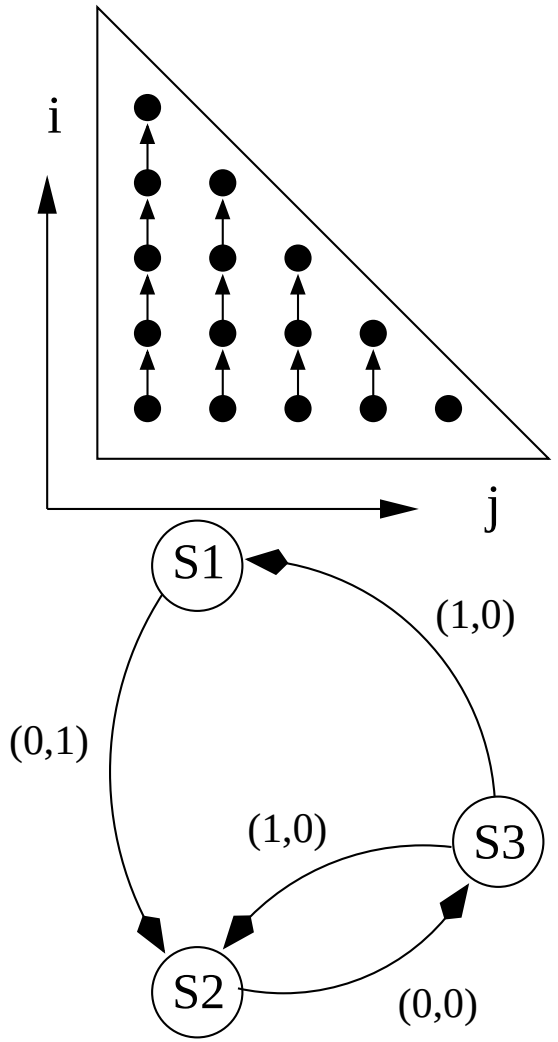
Représentation des dépendances

- Graphe de dépendance développé

```
for(i=1;i<N;i++)
  for(j=1;j<N-i+1;j++) {
    a[i][j] = a[i-1][j];
  }
```

- Graphe de dépendance réduit

```
for(i=1;i<n;i++)
  for(j=1;j<n;j++) {
    S1:  a[i,j]=c[i-1,j];
    S2:  b[i,j]=a[i-1,j]+c[i-1,j];
    S3:  c[i,j]=b[i,j];
  }
```



Représentation des dépendances

On peut aussi définir des approximations des vecteurs de distance par des polyèdres.

- Domaine d'itération d'une instruction S

$$D(S) = \{x \in \mathbb{Z}^{n_S} \,|C_S.x \geq c_S\}$$

où n_S est la dimension du domaine d'itération de S.

- Les relations de dépendance entre deux instructions S et T sont données par des relations affines de $\mathbb{Z}^{n_S} \rightarrow \mathbb{Z}^{n_T}$.
- Cette représentation est exacte lorsque les domaines d'itérations sont représentés par des équations affines.
- L'analyse de dépendance, dans un contexte général, est indécidable
- On doit prendre en compte certaines restrictions sur les programmes.

Transformations de programmes

- Transformations de programme
  1. Analyse des dépendances
  2. Modification de la structure du calcul
  3. Réécriture d'un programme équivalent
- Critères d'optimisation
  - Réduction de la taille mémoire nécéssaire au calcul
  - Localité des calculs pour une réutilisation maximale

Place mémoire d'une application

- Dépend de la *durée de vie* des variables et des tableaux en mémoire
  - Date de première écriture
  - Date de dernière lecture
- Réutilisation de la mémoire en dehors de la durée de vie des variables
- Estimation de la taille mémoire

$$\text{Coût Mémoire} = \max_{\forall t} \left\{ \sum_{v \in \text{Variables vivantes}(t)} taille(v) \right\}$$

- La précision de l'estimation dépend du niveau de conception auquel on se place.

Localité spatiale des accès

Distance physique entre deux accès à la mémoire, influence les conflits de cache dans les mémoire non (totalement) associatives

- On peut maximiser le nombre de distances k inférieure à une borne donnée $max \mid k_{b,t} \leq K \mid$ (boucle b, tableau t).
- On peut aussi minimiser la valeur moyenne des distances d'un programme : $min(\sum_b k_{b,t})$.

Les modifications apportées ici ont une influence sur les méthodes de placement des données en mémoire.

Localité temporelle des accès

La localité temporelle est donnée par le temps écoulé entre deux accès successifs à la même adresse mémoire.

- borner les délais d'accès à un emplacement par un seuil, sachant que lorsque la contrainte ne peut pas être respectée le délai peut être dépassé.
- minimiser le temps entre deux accès successifs.

$$\sum_{(i',i)\in DoubleAccès} |i' - i|$$

où $DoubleAccès$ est l'ensemble des dépendances correspondant à deux accès successifs à un même emplacement mémoire.

base principale des transformations pour l'optimisation de la gestion mémoire

Transformations de programmes

Transformations portant principalement sur les boucles et les tableaux.

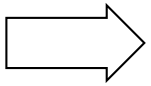
- Permet de manipuler le code utilisant les tableaux en mémoire
- Facteur important de gain en place mémoire
- Utilisation des propriétés de localité des accès dans les répétitions

Fusion de boucle

Permet de regrouper plusieurs boucles en une seule.

- Utile pour transformer des tableaux en variables scalaires.
- Permet de rapprocher la production de la consommation des valeurs

```
for(i=1; i<n; i++)
  a[i] = ... ;
for(i=1; i<n; i++)
  b[i] = a[i-2];
```



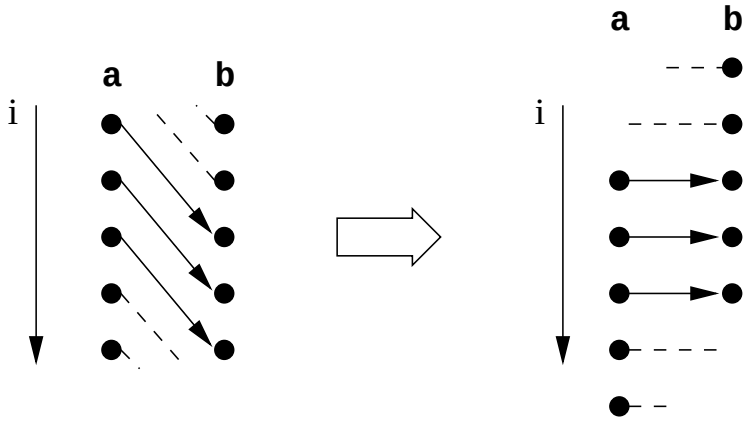
```
for(i=1; i<n; i++) {
  a[i] = ... ;
  b[i] = a[i-2];
}
```

Décalage d'instructions

On peut redéfinir une date d'ordonnancement pour les calculs dans les boucles.

- technique appelée aussi *software pipelining* ou *retiming*

```
for(i=1; i<n; i++) {
  a[i] = ... ;
  b[i] = a[i-2];
}
```



prologue

```
for(i=1; i<n-2; i++) {
  a[i] = ... ;
  b[i+2] = a[i];
}
```

épilogue

Pavage de boucles

Permet de fractionner une boucle pour diminuer la quantité de donnée manipulées dans la boucle la plus interne (conflits de capacités).

```
for(i=1; i<n-2; i++) {
  a[i] = ... ;
  b[i+2] = a[i];
}

for(i=1; i<(n-2)/P; i=i+P)
  for(j=; j<P; j++) {
    a[i*P+j] = ... ;
    b[i*P+j+2] = a[i*P+j];
}
```

Echange de boucles

Permet de changer le sens de parcour d'un tableau multidimensionnel pour avoir les dépendances les plus importantes dans la boucle la plus interne.

```
for(i=1; i<(n-2)/P; i=i+P)
  for(j=; j<P; j++) {
    a[i*P+j] = ... ;
    b[i*P+j+2] = a[i*P+j];
}

for(j=; j<P; j++)
  for(i=1; i<(n-2)/P; i=i+P) {
    a[i*P+j] = ... ;
    b[i*P+j+2] = a[i*P+j];
}
```

Composition des transformations

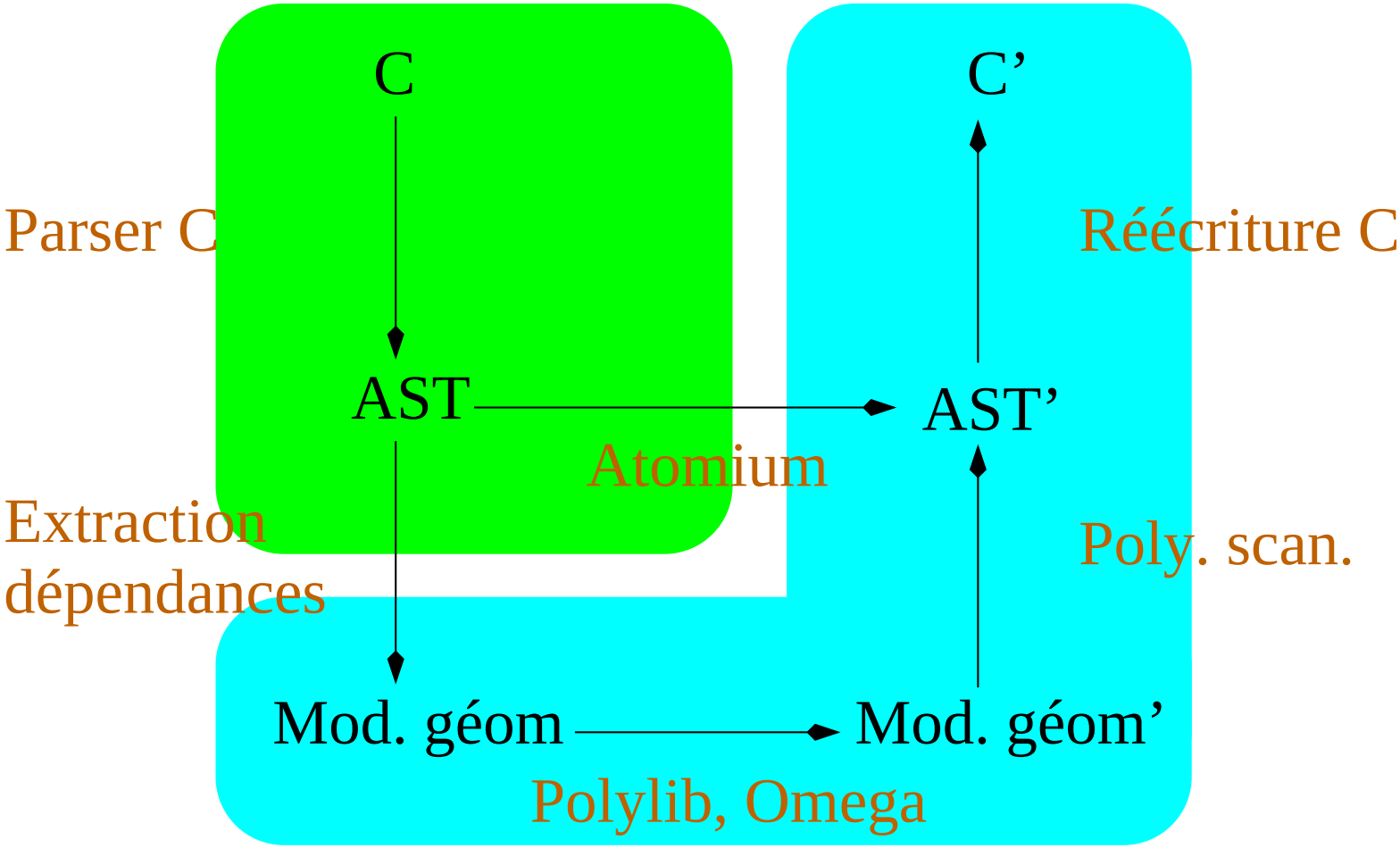
- Les tranformations de haut niveau ne sont pas efficaces en elles mêmes.
  - Il faut les seconder par des outils de placement et des éditeurs de liens appropriés.
- Les transformations ne sont efficaces que si elles sont pleinements intégrées.
  - De nombreuses transformations se complètes, d'autres annulent leurs effets.
- Il faut mettre en place une chaine complète d'optimisation
- Les transformations de haut niveaux permettent d'avoir des gains importants si elles sont effectuées dès le début de la conception.

Méthodologie de conception/compilation

DTSE Data Transfer and Storage Exploration

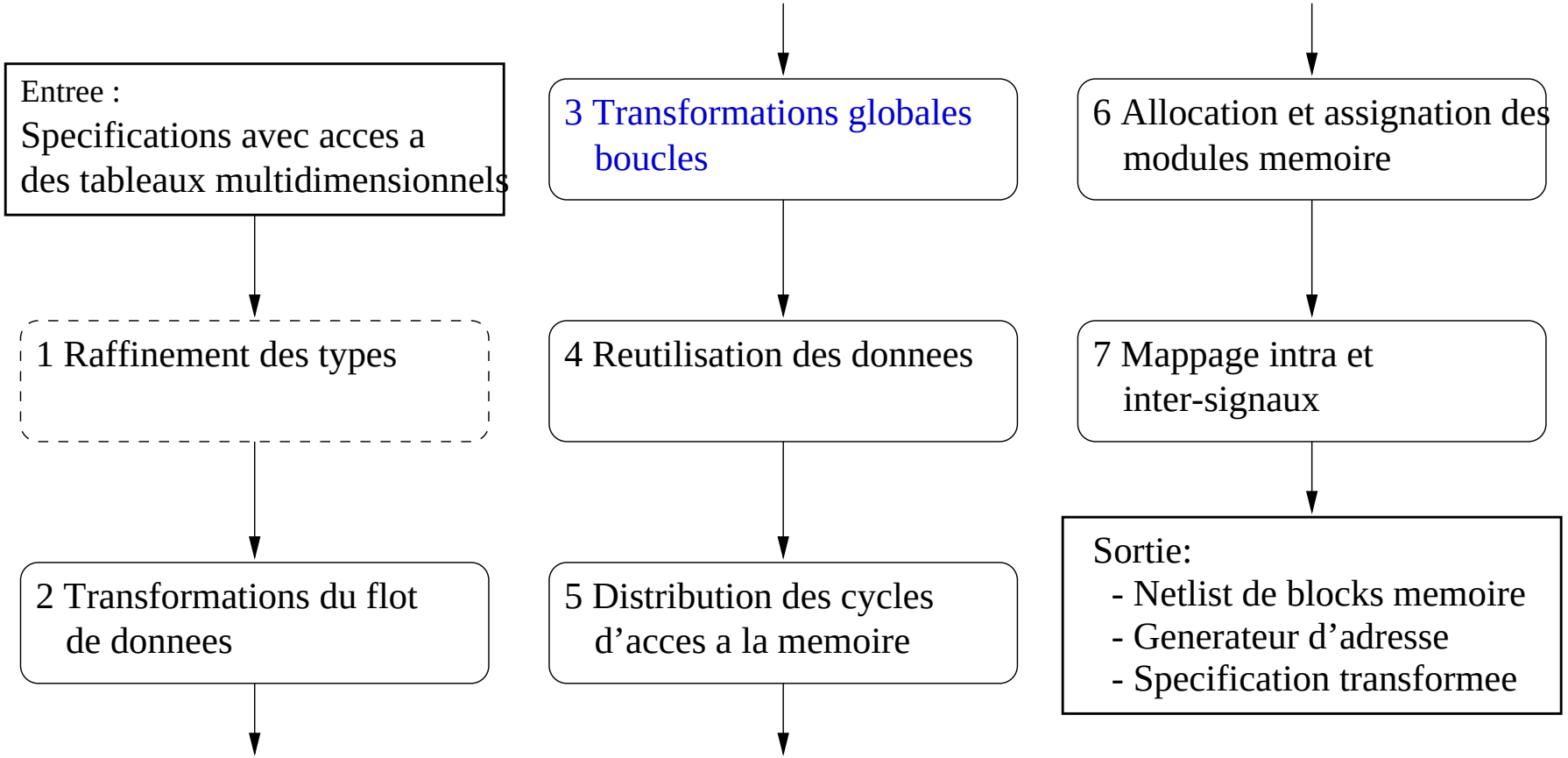
- Méthodologie développée à IMEC (Belgique)
- Permet de travailler sur une application tout au long de la conception

Chaîne de compilation



Méthodologie proposée

DTSE: Data Transfer and Storage Exploration



Etape 1

Transformations de raffinement de type

- Réduit la largeur de bit de donnée : optimisation algébriques
- Changement d'algorithme (DFT contre FFT)
- Passage d'arithmétique flottante en arithmétique virgule fixe

- Cette étape ne fait pas partie intégrante de la DTSE mais est très importante lors de la conception
- Cette étape ne peut être automatisée

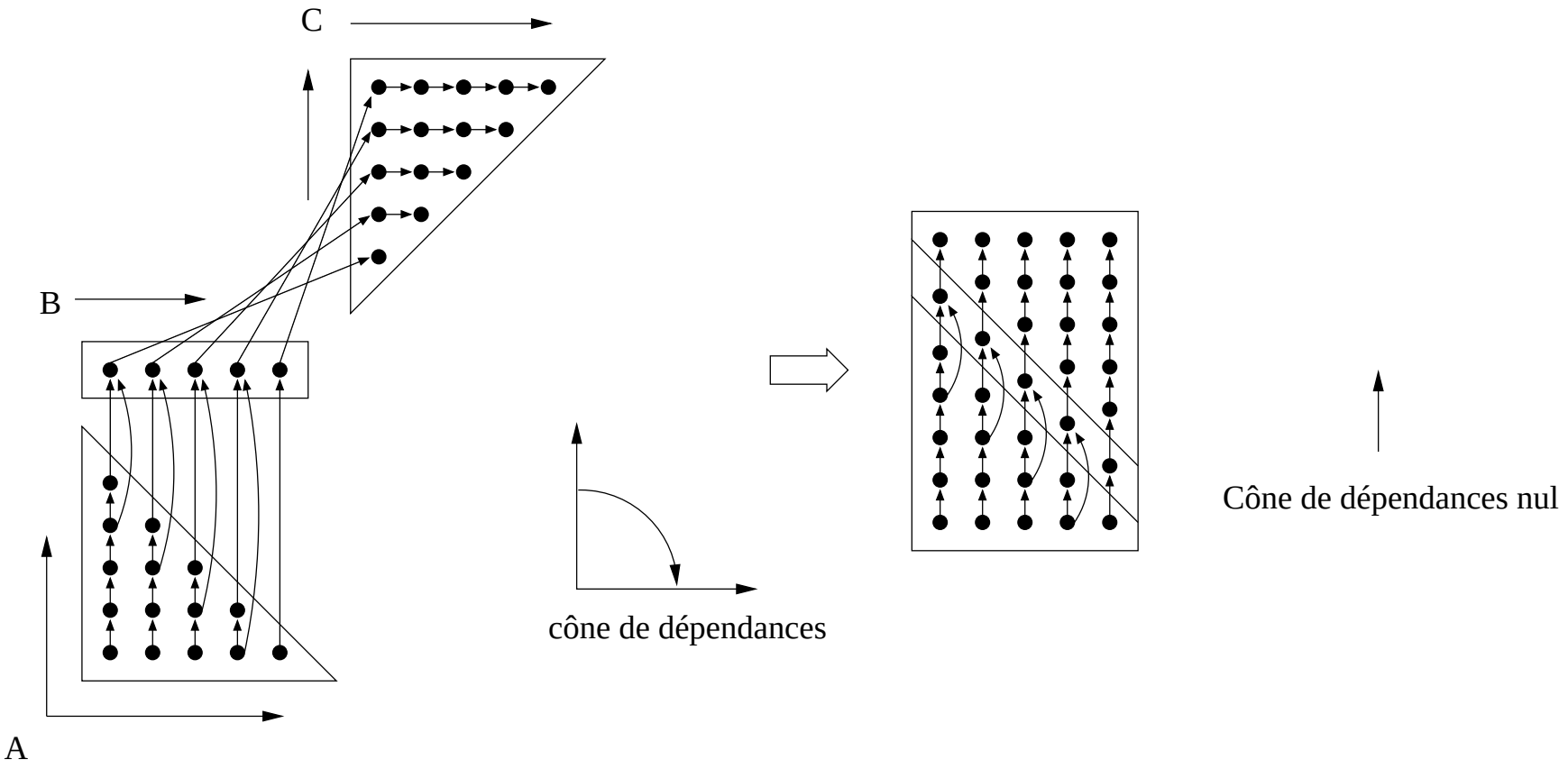
Etape 2

Transformation de flot de données

- Réduction des transferts redondants
  - Substitution de tableaux
  - Prise en compte de l'associativité des opérations (fonctions)
  - Prise en compte de recalcul au lieu de stockage dans des tableaux
- Cette étape ne peut également pas être automatisée

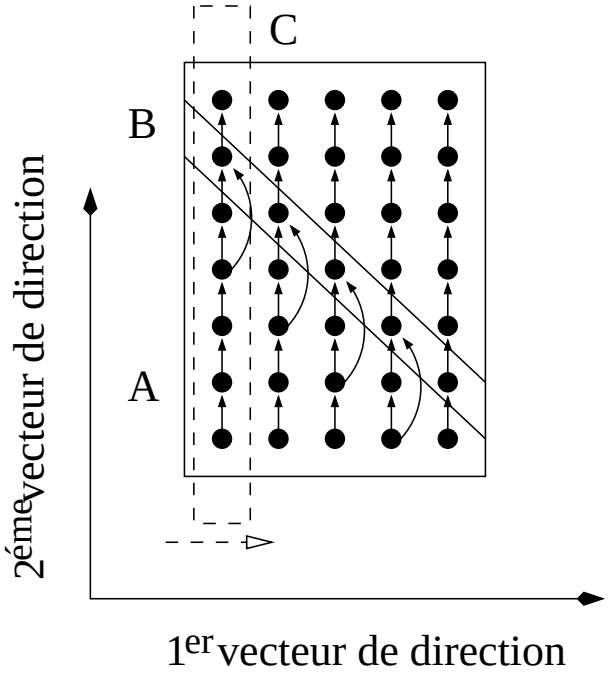
Etape 3 : Ordonnancement DTSE

- Une première phase permet l'optimisation de la régularité et de la longueur des dépendances



Ordonnancement DTSE

- Une deuxième phase permet de trouver un ordonnancement pour reconstruire une exécution séquentielle



- Résolution efficace pour les cas simples
- Extension pour les problèmes complexes
- Fonctions objectives variées
- Intégré dans une méthodologie d'optimisation
- Le code produit peut être difficile à relire

Etape 4

Data Reuse : réutilisation des données

- Utilisation de la hiérarchie mémoire

```
for(;;x++)
  something = image[...][x];
for(;;x++)
  somethingelse = image[...][x];

for(;;x++)
  line_cache[x] = image[...][x];
for(;;x++)
  thesamething = line_cache[x];
for(;;x++)
  thesamethingelse = line_cache[x];
```

- Utilisation d'un arbre de réutilisation pour faire la mise en place
- Peut servir d'entrée à un compilateur

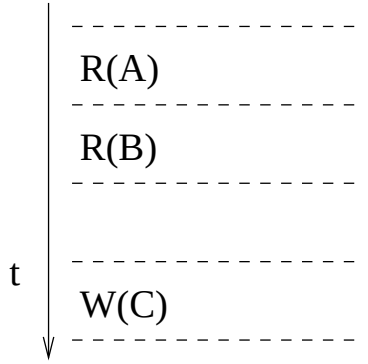
Etapes 5

Attribution des cycles de lecture/écriture

- Permet de prendre en compte les accès simultanés aux mémoires

```
for(i=0 to 100)
{
  C[i]=A[i]+B[i];
}
```

```
for(i=0 to 100)
{
  a=A[i];
  b=B[i];
  c=a+b;
  C[i]=c;
}
```

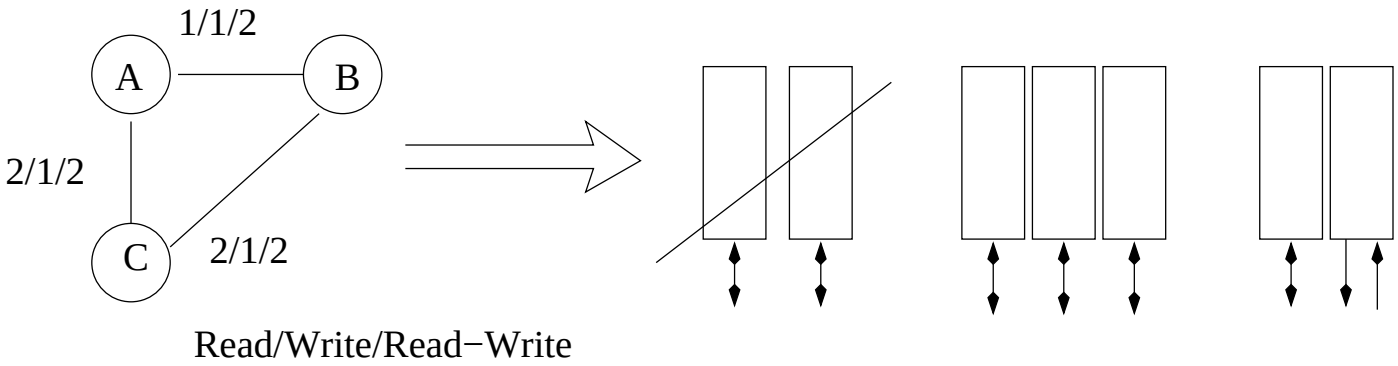


- Ordonnancement des entrées/sorties
- Construction d'un graphe de conflit (allocation de ressource)

(Etape 6)

Étape utilisée lors de la mise en place d'une hiérarchie mémoire dédiée (construction de l'architecture)

- Allocation d'une distributioin de la mémoire en bancs
- Assignation des tableaux aux bancs mémoire
- Etape dirigée par l'utilisation de bibliothèques de composeants caractérisé



- Recherche exhaustive parmis toutes les possiblités
- Nécessite une intéraction avec l'étape 5.

Etape 7 : *In Place Mapping*

- Permet d'utiliser la durée de vie des objet pour les placer au mieux en mémoire.
- Réutilisation d'espace physique (différent de l'étape 4)
- Utilise un modèle polyédrique
- Rend délicat le calcul des adresses en mémoire

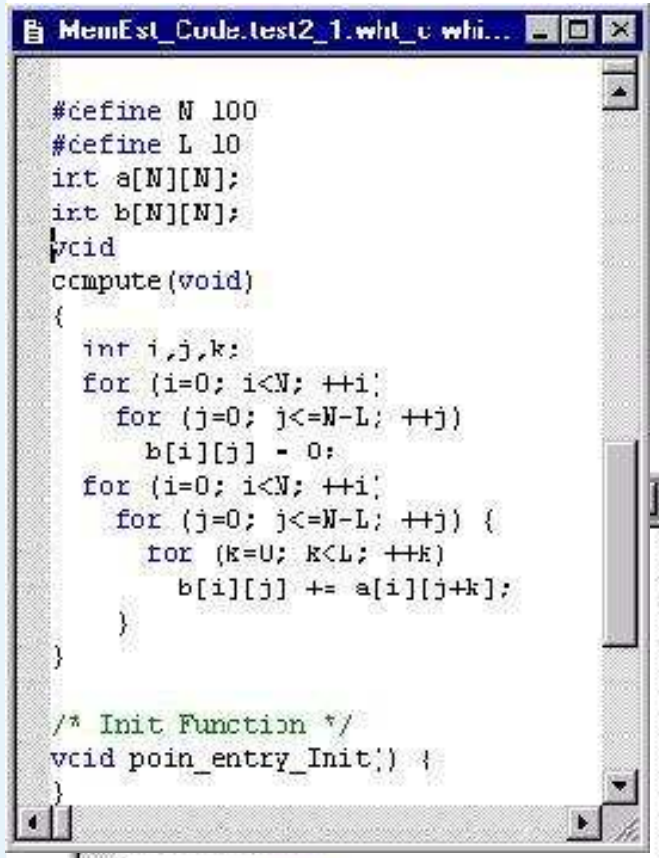
```
int A[100][100];
int B[20][20];
int C[1000];
for(i,j,k,l)
{
  B[i][j] = fun(B[j][i],
     A[i+k][j+l]);
}
```

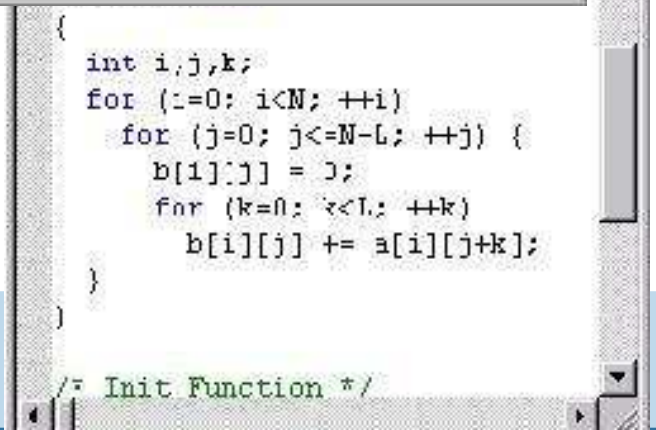
```
int mem[10400];
for(i,j,k,l)
{
  mem[10000+i+20*j] =
     fun(mem[10000+j+20*i],
         mem[i+k+100*(j+l)]);
}
```

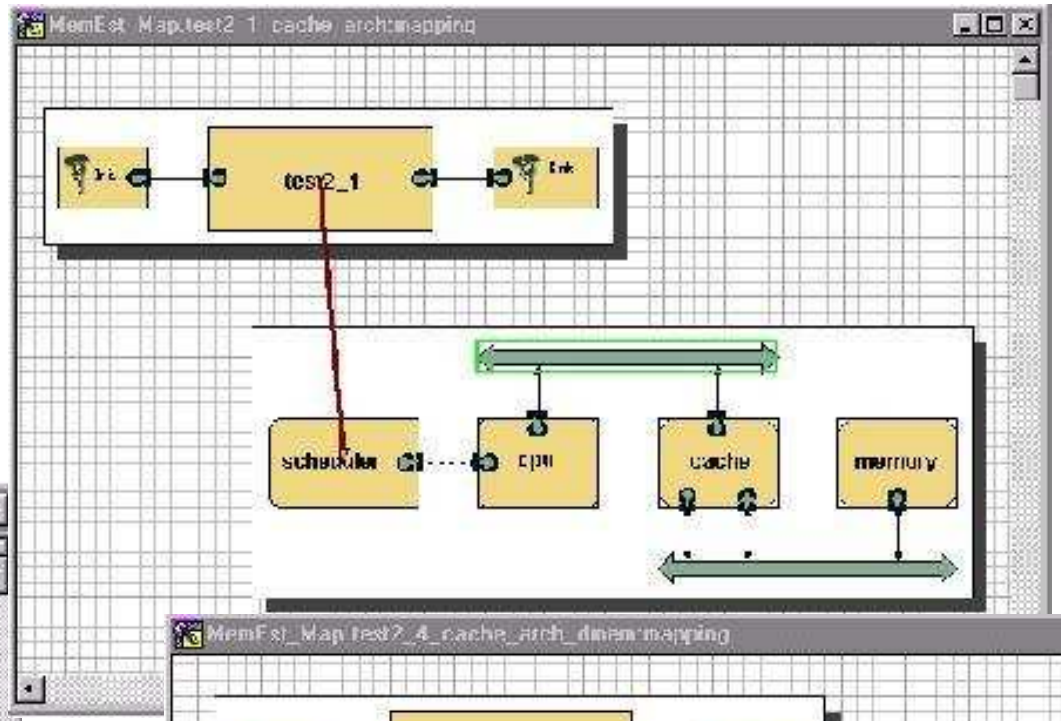
Mise en pratique et intégration des transformatio

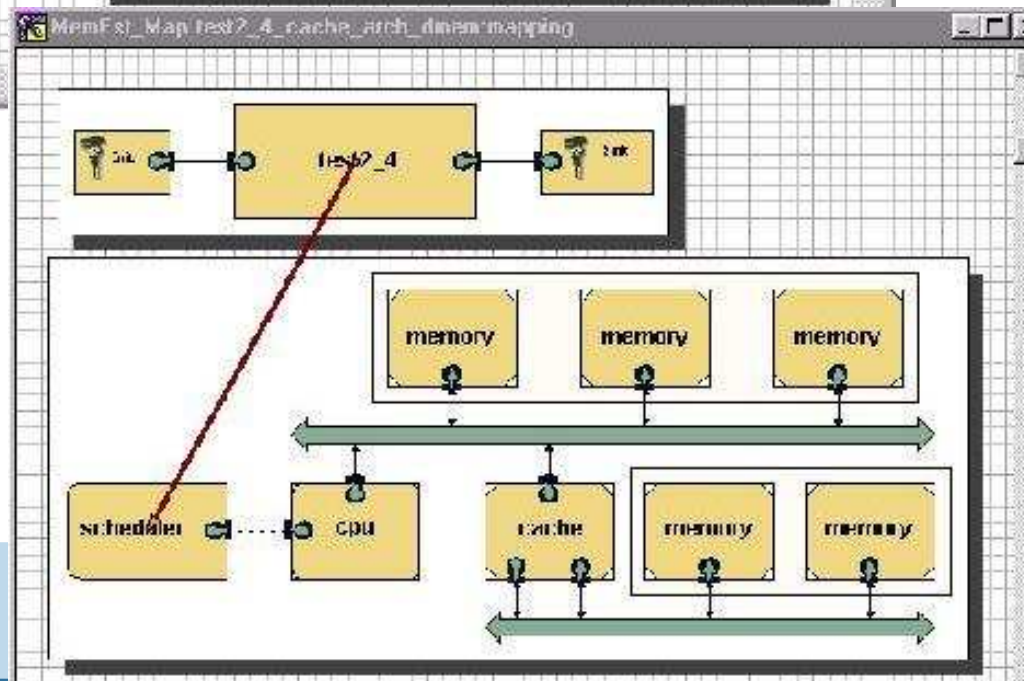
- Utilisation de plateforme de Codesign
- Intégration des transformations dans le flot de synthèse
- Validation des fonctions de coût

Expérimentations : utilisation de VCC









Résumé

- Utilisation de ces techniques dans les systèmes communicants : consommation et temps
- Applications cibles : multimédia, optimisation de protocoles, système d'exploitation
- Une automatisation globale de toutes les étapes semble peu réaliste
  - L'expertise du concepteur est indispensable pour guider les outils et doit être prise en compte

Présentations

- Simulation de système complet sur puce
  - Reim DOUMAT
  - Thomas Watteyne
- Réseau sur puce
  - Jacques Saraydaryan
  - Niu Meng
- Langages de spécification
  - Li Zao
- OS embarqué
  - Pierre Parrend
  - Soumaya Ghaddab
- Compilation pour l'embarqué
  - Florent Bouchez
- Synthèse haut niveau de circuits
  - Betânia STEFFEN ABDALLAH