Langages synchrones avec horloges périodiques

Soutenance de stage

Alexandre Bertails

Laboratoire de Recherche en Informatique, Université Paris-Sud

11 septembre 2006

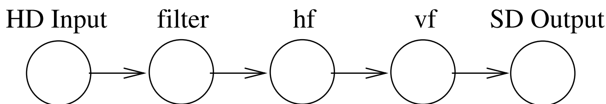
Problématique

- Contexte :
  - calculs réguliers et contraintes temps réel dans l'embarqué
  - pas du *best-effort*
  - garanties statiques
- Buts :
  - étude des formalismes
  - production de code impératif
- Plan :
  - une application typique : le *downscaler*
  - formalismes classiques du domaine
  - compilation de langages synchrones avec horloges périodiques

Une application typique : le *downscaler* (rétrécisseur)

- De la haute-définition vers la basse-définition (de 1920x1080 vers 720x480) : $1920 * 1080 * (9+8+1) * 50 \approx 10^9 op/s$
- Ce n'est qu'un exemple (mais réel) qui illustre bien le temps-réel et le calcul intensif
- Implémentation en LUCID SYNCHRONE pour expérimenter plusieurs approches
  - Filtre de convolution + élection de pixels
  - Afficheur
  - Protocole de communication vidéo CCIR656

Downscaler : architecture

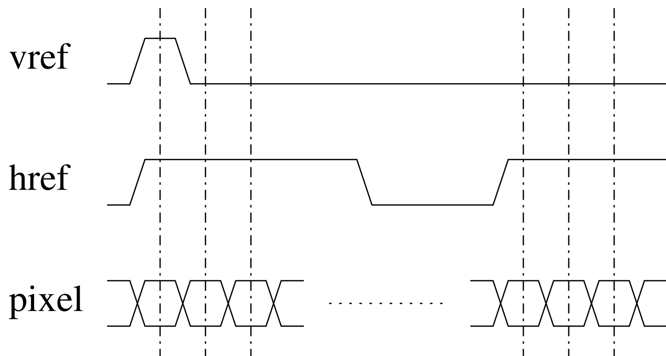


Architecture :

- Filtre de convolution + élection de pixels
- Afficheur
- Protocole de communication vidéo CCIR656

```
let node main () =
  let (vref, href, pixel) =
    video_in () in
  let (vref', href', pixel') =
    downscaler (vref, href, pixel) in
  video_out (vref', href', pixel')
```

CCIR656 : chronogrammes



Styles de programmation synchrone

En réalité, plusieurs versions du *downscaler* :

- Horloges (rythmes) : information de présence des pixels placée dans les horloges
- Signaux valués : information de présence des pixels encodée dans un signal ; tests de présence
- Automates : machine à état

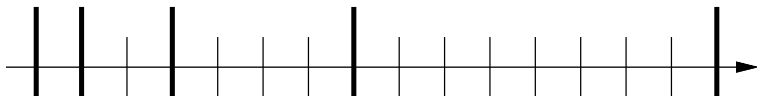
Les horloges semblent être le style de programmation adapté.

Le dataflow synchrone

- Modèle du temps synchrone
- Basé sur les équations de suites (flots de données)
- Pour des circuits et du logiciel
- Exécution 0-délai : il existe une exécution sans mécanisme de synchronisation par tampons
- Adapté à la vérification (model-checking), outils de simulation

Le dataflow synchrone : horloges

- L'activation des nœuds est gardée par des *horloges* (rythmes d'échantillonnage)
- Qu'est-ce qu'une horloge ? $\longrightarrow$ flot booléen
- Les horloges sont quelconques
- Exemple : horloges actives sur les puissances de 2



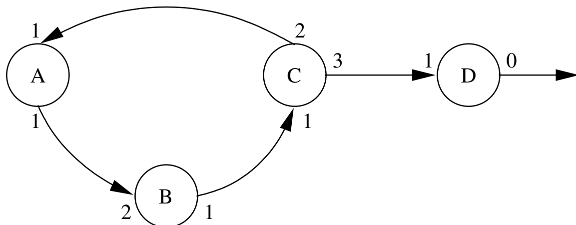
Le synchrone : compilation

- Code de simulation (ou boucle simple) :

```
1: while true do
2:     read(input)
3:     modify(state)
4:     emit(output)
5: end while
```

- Horloge de base = rythme le plus rapide
- Horloge $\Rightarrow$ `if`
- Chaque nœud gère la mémoire et les horloges de ses nœuds fils

Synchronous Dataflow [Lee & Messerschmidt, 87]

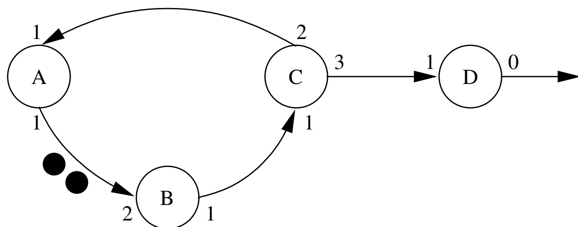


- Processus communicant par FIFO [Réseaux de Kahn, 87]
- Arc = FIFO
- Nœud = processus de calcul (n'importe quel langage hôte)
- Renseignements fournis sur le nombre de données
  - produites : les arcs de sortie
  - consommées : les arcs entrants
- Ordonnancement cyclique statique des nœuds.

SYNCHRONOUS DATAFLOW : ordonnancement

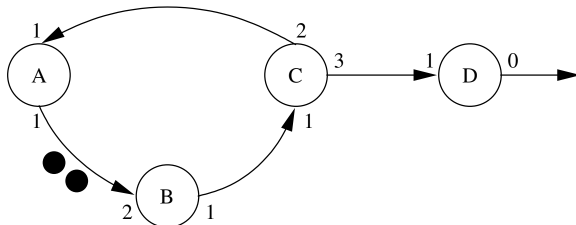
- L'ordonnancement doit vérifier les contraintes suivantes :
  - les données sont disponibles à l'activation d'un noeud
  - les tampons alloués ne débordent jamais
- L'algorithme
  - en entrée : la matrice topologique (représentation du SDF) et les quantités de données présentes à l'initialisation
  - en sortie : un ordonnancement et la taille des tampons (= mémoire)

SYNCHRONOUS DATAFLOW : exemple



Équations d'équilibre	*Plus petite solution*
$1.n_A = 2.n_C$	$n_A = 2$
$1.n_A = 2.n_B$	$n_B = 1$
$1.n_B = 1.n_C$	$n_C = 1$
$3.n_C = 1.n_D$	$n_D = 3$

SYNCHRONOUS DATAFLOW : exemple



- Ordonnancement : BCD^3A^2
- Taille des tampons :

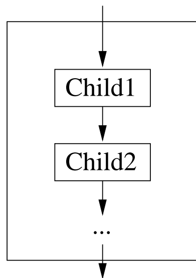
Arc	Taille du tampon
$(A \rightarrow B)$	2
$(B \rightarrow C)$	1
$(C \rightarrow A)$	2
$(C \rightarrow D)$	3

STREAMIT : présentation

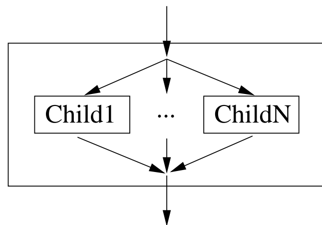
- Restriction structurelle des SDF
- Chaque nœud de calcul a un seul fil d'entrée et un seul fil de sortie.
- Composition des flots au travers d'un nombre limité d'opérateurs.
- On peut consulter des données qui n'ont pas encore été consommées.

`filter` désigne un processus contenant du code à exécuter.

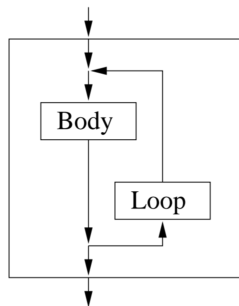
STREAMIT : opérateurs



pipeline

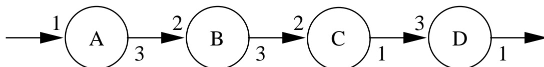


splitjoin



feedback

STREAMIT : ordonnancement



- *Single Appearance Schedule* : $A^4(B^2C^3D)^3$. Taille des tampon et latence maximales ; code sous forme de boucles imbriquées.
- *Push Schedule* : $E^2CDBF^2BC^2D$ où $E = ABC$ et $F = CE$. Taille des tampons et latence minimales ; code répliqué.

L'algorithme d'ordonnancement utilisé est un compromis : ordonnancement hiérarchique.

STREAMIT / Synchrone : comparaison

- STREAMIT
  - + : adapté au streaming ; ordonnancement statique efficace : code avec boucles
  - - : modèle contraignant ; pas de propriétés temps-réel ; essentiellement pour des systèmes périodiques (streaming)
- SYNCHRONE
  - + : temps-réel dur ; pas de restrictions structurelles (horloges quelconques)
  - - : composition des horloges pas assez souple ; pas de compilation vers du code en boucles imbriquées ; non satisfaisant pour le calcul intensif (code de simulation)

Vers une nouvelle approche . . .

Motivations

- Cadre : langages synchrones
- Restriction : systèmes périodiques
- Pourquoi :
  - un cadre spécifique pour les systèmes périodiques
  - exploiter le caractère périodique pour la compilation
  - quels rapports entre SDF et STREAMIT ?
- Introduction d'horloges spécifiques : les horloges périodiques

Horloges périodiques

- *Rappel* : horloge périodique $c =$ flot booléen
- *Propriété* : $c(x+p) = c(x)$
- *Langage* : $\mathbf{(}0^*1(0|1)^*\mathbf{)}$
- *Exemple* : l'horloge (101) représente la suite infinie $101101\ldots$

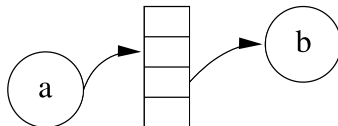
(101)

Un petit langage d'équations

$$
\begin{array}{rcl}
e & ::= & x \\
 & & \mid i \\
 & & \mid \texttt{op}\ (e, e) \\
 & & \mid \texttt{op}\ e \\
 & & \mid e\ \texttt{fby}\ e \qquad (*\ \textit{followed by}\ *) \\
 & & \mid e\ \texttt{when}\ clock \\
 & & \mid \texttt{merge}\ clock\ e\ e \\
eq_list & ::= & e\ .\ eq_list \\
 & & \mid e\ . \\
clock & ::= & \left(0^*1(0|1)^*\right)
\end{array}
$$

Ordonnancement

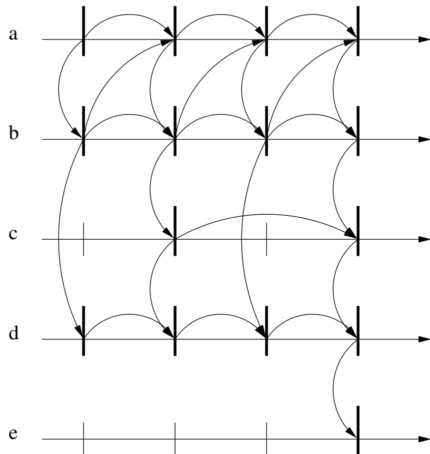
- Deux phases : initialisation et cyclique
- Mémoire = tampon circulaire + index



- Algorithme :
  1. Calculer la période de réaction : *ppcm* des périodes
  2. Construire les graphes de dépendances de données Γ^{init} et Γ^{cycl}
  3. Extraire les ordonnancements en tenant compte des accès à la mémoire

Ordonnancement : exemple

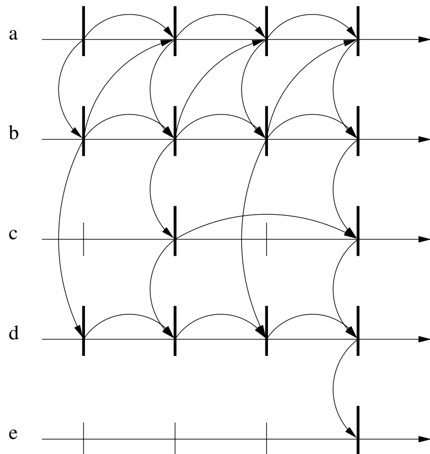
```
a = 0 fby b
b = a + 1
c = b when (01)
d = merge (01) c
             (b when (10))
e = d when (0001)
```



Ordonnancement : exemple

1. a , $i[b] = 1$
2. b , $i[a] = 0$
3. a , $i[b] = 1$
4. b , $i[a] = 0$
5. a , $i[b] = 1$
6. b , $i[a] = 0$
7. a , $i[b] = 1$
8. b , $i[a] = 0$
9. c , $i[b] = 2$
10. c , $i[b] = 0$
11. d , $i[b] = 3$, $i[c] = 0$
12. d , $i[b] = 0$, $i[c] = 1$
13. d , $i[b] = 1$, $i[c] = 0$
14. d , $i[b] = 0$, $i[c] = 0$
15. e , $i[d] = 0$

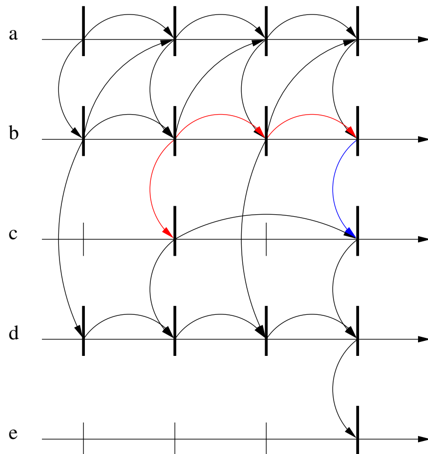
$$\mathcal{M} = \begin{pmatrix} a & \rightarrow & 1 \\ b & \rightarrow & 4 \\ c & \rightarrow & 2 \\ d & \rightarrow & 1 \\ e & \rightarrow & 1 \end{pmatrix}$$



Ordonnancement : exemple

1. a , $i[b] = 1$
2. b , $i[a] = 0$
3. a , $i[b] = 1$
4. b , $i[a] = 0$
5. a , $i[b] = 1$
6. b , $i[a] = 0$
7. a , $i[b] = 1$
8. b , $i[a] = 0$
9. c , $i[b] = 2$
10. c , $i[b] = 0$
11. d , $i[b] = 3$, $i[c] = 0$
12. d , $i[b] = 0$, $i[c] = 1$
13. d , $i[b] = 1$, $i[c] = 0$
14. d , $i[b] = 0$, $i[c] = 0$
15. e , $i[d] = 0$

$$\mathcal{M} = \begin{pmatrix} a & \rightarrow & 1 \\ b & \rightarrow & 4 \\ c & \rightarrow & 2 \\ d & \rightarrow & 1 \\ e & \rightarrow & 1 \end{pmatrix}$$



Ordonnancement : discussion

- Si pas d'offset, deux ordonnancements sous les formes suivantes :
  - $(ab)^4c^2d^4e$: code bien factorisé sous boucle. Taille de la mémoire à allouer maximale. *cf.* *Single Appearance Schedule*
  - $d(ab)^2cd^2(ab)^2cde$: code plus expansé. Taille de la mémoire à allouer minimale. Les données calculées sont utilisées le plus rapidement possible. *cf.* *Push Schedule*
- **Limitations** : on a des offsets et on expanse trop.

Conclusion

- Implémentations :
  - le *downscaler* en LUCID SYNCHRONE
  - divers programmes en STREAMIT
- Résultats :
  - premier algorithme d'ordonnancement de systèmes synchrones périodiques
  - similitudes entre nos ordonnancements et ceux de STREAMIT
- Futur :
  - mise en œuvre
  - modularité
  - boucles imbriquées

Conclusion

- Implémentations :
  - le *downscaler* en LUCID SYNCHRONE
  - divers programmes en STREAMIT
- Résultats :
  - premier algorithme d'ordonnancement de systèmes synchrones périodiques
  - similitudes entre nos ordonnancements et ceux de STREAMIT
- Futur :
  - mise en œuvre
  - modularité
  - boucles imbriquées

Merci de votre attention. Avez-vous des questions ?