Polycopié de CS442-partie LG

— Version 2022/2023 —

Laure Gonnord - (et l'équipe synchrone de Verimag)

Premature optimization is the root of all evil (or at least most of it) in programming.

Donald Knuth, *Décembre 1974, Conférence du Prix Turing 1974, Communications of the ACM.*

Afin d'améliorer ce poly n'hésitez pas à me soumettre toute critique, suggestion, remarque ou correction, dans mon casier ou, électroniquement, à l'adresse

`Laure.Gonnord@esisar.grenoble-inp.fr`

Table des matières

Applications Synchrones (CS442)

Introduction, problématiques

Laure Gonnord

Grenoble INP/Esisar

2022-2023

Plan

1 Introduction
Généralités

2 Systèmes pour le temps-réel

3 Langages pour le temps-réel

4 Quelles applications ?

Contexte - Système embarqué

Définition

An embedded system is a special-purpose computer system designed to perform one or a few dedicated functions. It is usually embedded as part of a complete device including hardware and mechanical parts. (Wikipedia)

- Industrie légère : téléphones portables, appareils ménagers
- Industrie lourde : aéronautique, aérospatiale, ...

Contraintes

- Gestion d'un système physique dans son environnement.
- Contraintes d'échéances : molles/dures.
- Importance de l'optimisation (mémoire, énergie).

Contenu de CS442

- *En lien avec le cours de temps-réel*
- Méthode synchrone pour la conception des systèmes temps réels.
- Développement d'applications synchrones, validation logicielle.

▶10 séances de 1.5h "académiques" (CM+TP, Laure Gonnord), en parallèle de 15h de programmation "langage industriel" (Guillaume Marie, Crouzet).

Plan

1. Introduction
   Généralités
2. Systèmes pour le temps-réel
3. Langages pour le temps-réel
4. Quelles applications ?

Credits

Intro honteusement pompée sur les transparents
`http://beru.univ-brest.fr/~singhoff/supports.html`

Définition

"En informatique temps réel, le comportement correct d'un système dépend, non seulement des résultats logiques des traitements, mais aussi du temps auquel les résultats sont produits. "[2]

Différentes notions de déterminisme

Le **déterminisme** est une notion-clef :

- Déterminisme logique : les mêmes entrées appliquées au système produisent les mêmes résultats.
- Déterminisme temporel : respect des contraintes temporelles (échéances, rythme, ...).

Les contraintes de temps

Important Un système temps réel n'est pas un système “qui va vite” mais un système qui satisfait à des contraintes temporelles.

Quelques ordres de grandeur :

- La milliseconde pour les systèmes radar.
- La seconde pour les systèmes de visualisation humain.
- qq heures : production chimique
- ...

Des contraintes plus ou moins dures

Le besoin en **garantie de service** (niveau de respect des contraintes) peut être différent :

- Systèmes temps réel dur ou critique.
- Systèmes temps réel souple.

Exemple 1 : domaine de l'avionique

Système temps réel critique :

- Contraintes temporelles : temps de réponse, échéance, date d'exécution au plus tôt, cadence, etc.
- Dimensionnement au pire cas et réservation des ressources.
- Utilisation de redondance matérielle et logicielle.
- Matériel et logiciel dédiés. Système fermé, validé a priori.
- Système réparti synchrone : commandes de vol, radars, moteurs, etc.

Exemple 2 : multimédia sur le Web

Système temps réel souple :

- Contraintes temporelles : gigue, délais de bout en bout, temps de réponse. Synchronisations intra et inter-flux.
- Plate-forme généraliste. Non déterminisme temporel à cause du matériel et du logiciel (ex : PC + windows).
- Application interactive.
- Nombre de flots inconnu.
- Débits variables et difficiles à estimer hors ligne.

Autres exemples / domaines d'activité

- Transports (métro, aérospatiale, SIG : systèmes d'info géographique et systèmes de régulation automobile).
- Médias (décodeurs numériques openTV).
- Services téléphoniques (téléphone mobile, auto-commutateur).
- Supervision médicale, écologique.
- Système de production industriel : centrale nucléaire, chaîne de montage, usine chimique.
- Robotique (ex : PathFinder)

Plan

1 Introduction
   Généralités

2 Systèmes pour le temps-réel

3 Langages pour le temps-réel

4 Quelles applications ?

Problématique

Objet : montrer que réaliser un système temps-réel (matériel+logiciel) est un **cauchemard**.

Les systèmes généralistes à temps partagé 1/2

Objectifs de tels systèmes :

- Faciliter l'accès aux ressources.
- Masquer les ressources (process, mémoire, disques).
- Recherche de l'équité, maximisation du débit global.

Les systèmes généralistes à temps partagé 2/2

Deux styles de programmation :

- Mono-programmation : interaction synchrone entre le couple processeur/mémoire et les périphériques.
- Multi-programmation : interaction asynchrone. Le processeur profite du temps ainsi libéré pour effectuer d'autres traitements.

Ordonnancement sous Linux

Les soucis des ordonnanceurs généralistes :

- Pas de prise en compte de l'urgence ou de contrainte temporelle.
- Politique souvent opaque.
- Temps de réponse inconnue.

Les entrées/sorties 1/2

Périphériques

Processeur
Caches

Mémoire

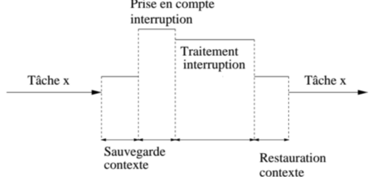
E/S
Contrôleur DMA

Bus

Modes d'échanges :

- Scrutation active de l'unité : polling. **Monopolise le processeur**. Comportement temporel déterministe.
- Interruptions.
- Accès direct à la mémoire (DMA ; Direct Memory Access). Contention sur le bus. Comportement temporel **indéterministe**.

Les entrées/sorties 2/2



Interruptions et préemptivité :

- Priorité des interruptions non modifiable par le concepteur.
- Nombre d'occurrences inconnu. Acquisition et traitement successifs ▶long, non déterministe.
- UNIX = verrouillage important, peu préemptif.

▶Il faut découpler acquisition et traitement des interruptions, **source d'erreur**. Attention au choix du matériel.

Problématiques d'accès aux ressources

Synchronisation, communication et accès aux ressources effectués grâce à des sémaphores :

- Sémaphore = compteur entier/booléen + file d'attente.
- Utilisation : exclusion mutuelle, paradigme classique de coopération (producteur/consommateur, lecteur/rédacteur).

Mais :

- Certaines tâches sont bloquées ▶Impact sur l'ordo.
- **inversion de priorité** : tâche de + faible prio bloquant une tâche de plus forte prio pendant une durée inconnue.

Conclusion de cette partie

Les systèmes généralistes ne sont pas **temps-réel** :

- Ils ne sont pas déterministes : matériel / logiciel.
- L'ordonnanceur temps partagé n'offre aucune garantie.
- (linux) le noyau est non préemptif (donc si un autre processus + prio a besoin du processeur FAIL)

▶POSIX 1003.1b sous Linux pour systèmes TR souples
▶**Systèmes dédiés pour systèmes TR durs**
▶Entre les deux : utiliser un micronoyau temps réel cohabitant avec le noyau linux (ex RTLINUX).

Plan

1. Introduction
   Généralités
2. Systèmes pour le temps-réel
3. Langages pour le temps-réel
4. Quelles applications ?

Langages de bas niveau

C/C++, Assembleur :

- Largement diffusés et utilisés à ce jour.
- Accès direct aux ressources de bas niveau. Idéal pour les E/S.
- Doit être couplé avec les services du système (pour synchronisation, ordonnancement) ▶bibliothèques.
- Langage généralement restreint (sous-partie ayant un comportement temporel déterministe facile à évaluer).
- Peu adapté aux logiciels complexes et/ou volumineux.
- Pas de normalisation, donc peu de portabilité : logiciel "ad hoc"

▶On pratiquera un peu.

Un exemple de langage haut niveau

Ex : Ada 2012 [3] Langage conçu, entre autres, pour le support des applications temps réel :

- Abstractions temps réel : tâche, interruption, ordonnancement (priorité fixe et dynamique), synchronisation (sémaphore), timer et gestion du temps, outils de communication (rendez-vous).
- Interface/syntaxe et COMPORTEMENT du langage normalisés par l'ISO ▶forte portabilité.
- Compilation séparée. Typage fort. ▶Fiable. Adapté à la production de logiciels volumineux.

▶Langage complexe, que nous n'utiliserons pas (dommage).

Une autre solution : l'approche synchrone

Ex : Lustre [1]

- Séparation des problématiques : fonctionnel/temps.
- Paradigme de programmation plus simple : ordonnancement calculé à la compilation, communication mémoire partagée.
- La gestion du temps est faite à postériori (horloges physiques).
- Langage haut niveau avec peu de primitives de construction. Compilation séparée, typage.
- Générateur de code C/C++ ▶portabilité intéressante
- Un écosystème avec des outils de vérification formelle.

▶**l'essentiel de ce cours !**

Plan

1 Introduction
Généralités

2 Systèmes pour le temps-réel

3 Langages pour le temps-réel

4 Quelles applications ?

Les applications réactives

Systèmes à commander

Capteurs / ordres opérateur

Actionneurs

Ordinateur/Automates temps réel

Dessin par E. Foudrinier pour CS442.

Quelques définitions

Systèmes transformationnels

- Système qui réalise une fonction sans interaction avec l'extérieur

O=F(I)

Systèmes réactifs (interactifs)

- Système qui adapte sa réponse aux sollicitations externes (inputs) ou internes (states) : le système répond au plus vite

O=F(I, S)

Systèmes réactifs temps réel

- Système interactif dont la réponse est bornée dans le temps

O=F(I, S, t)

Dessin par E. Foudrinier pour CS442.

Conclusion

- Contexte : application critiques (et temps-réel).
- Modèle de programmation : boucle réactive.
- Besoin de garanties (fonctionnelles, non fonctionnelles).

▶Les langages synchrones.

Bibliography I

Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud.
The synchronous data flow programming language lustre.
Proceedings of the IEEE, 79, 1991.

John Stankovitch.
Misconceptions about real-time computing.
Proceedings of the IEEE, 21, 1988.

Luigi Zaffalon and Pierre Breguet.
Programmation concurrente et temps réel avec ADA 95.
Presses polytechniques et universitaires romandes.

Applications Synchrones (CS442)

Le langage Lustre

Laure Gonnord

Grenoble INP/Esisar

2022-2023

Plan

1. The Synchronous approach
2. The Lustre language
   - Clock sampling
   - Common errors in Lustre
3. Simulation of Lustre programs
4. Compilation of Lustre programs
5. Conclusion

Reactive system

(CC-BY-SA Captainm/Wikipedia)

- React to inputs :
  - Acquire inputs on sensors ;
  - Compute ;
  - Produce values on actuators.
- Actions impact the environment, thus subsequent inputs ;
- Response time must be *bounded*.

Programming

Functional correction

Compute the correct output values.

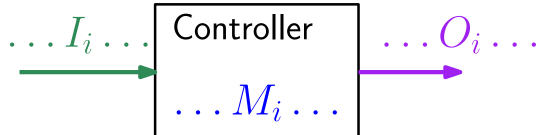
Temporal correction

Compute faster than the reactivity constraint.

Programming : the Functional Part

Remarks

- O_i depends *only* on the $I_1, I_2, ..., I_i$;
- Computations performed with *bounded memory* M_i.



Programming is ...

- ... identifying inputs and outputs ;
- ... defining :
  - The **output function** $O_i = f(I_i, M_i)$;
  - The **transition function** $M_{i+1} = g(I_i, M_i)$

The Asynchronous approach

Parallel processes ⇒ Concurrent multi-task implementation.

Difficulties :

- Scheduling : handle hard to predict execution times, jitter, etc ;
- Inter-task communications : handle communication order (priorities, rendez-vous, semaphores, etc).

Globally non-deterministic.

The synchronous approach

Real-time is replaced by a simplified, abstract, *logical time*.

- *Instant* : one reaction of the system ;
- Logical time : sequence of *instants* ;
- The program describes what happens during each instant ;
- *Synchronous hypothesis* : computations complete before the next instant. If so :
  - ⇒ We can ignore time inside an instant, only the order matters ;
  - ⇒ We are only interested in how instants are chained together.

Synchronous languages vs others

Advantages :

- Semantics defined formally ⇒ enables formal proofs and provable compilation ;
- High abstraction level ⇒ less work for the programmer, more for the compiler ;
- Bounded memory and execution time ;
- Barely needs an OS.

Disadvantages :

- Produced code less efficient than hand-written code ;
- Synchronous hypothesis hard to ensure (WCET, distributed systems) ;
- Not well-suited for multi-rate systems.

Plan

1. The Synchronous approach
2. The LUSTRE language
   - Clock sampling
   - Common errors in Lustre
3. Simulation of LUSTRE programs
4. Compilation of LUSTRE programs
5. Conclusion

Flows and clocks

- LUSTRE is a data-flow language : computations are triggered by incoming data ;
- In LUSTRE, every expression and variable is a flow ;
- *Flow* : infinite sequence of values + clock ;
- *Clock* : defines when a flow is present (has a value).

Example

x	3	4	5	2	6	...
y	True			False	True	...

Point-wise extension of classic operators

Example

c	True	False	True	False	...
x	x_1	x_2	x_3	x_4	...
y	y_1	y_2	y_3	y_4	...
x+y	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	...
`if c then x else y`	x_1	y_2	x_3	y_4	...

Delay operator

- The *pre* operator denotes the previous value of a flow ;
- Undefined for the first instant ;
- Usually combined with the initialisation operator `->` : `x->y` is *x* on the first instant, *y* otherwise.

Example

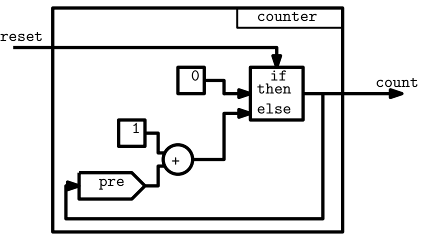
x	x_1	x_2	x_3	x_4	...
y	y_1	y_2	y_3	y_4	...
`pre x`		x_1	x_2	x_3	...
`y->pre x`	y_1	x_1	x_2	x_3	...

Structuration : nodes

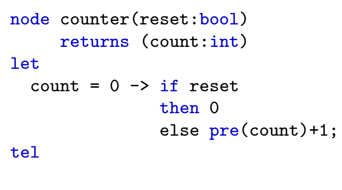
- A Lustre program consists of a set of *nodes*;
- The *main node* is specified by the compilation line;
- Each node contains a set of *equations*, which defines output flows from input flows;
- Equations are unordered;
- Nodes can be instantiated in flow expressions (like functions).

Example : A Resettable Counter

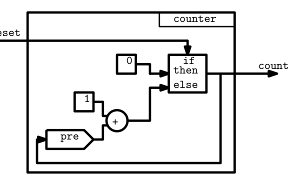
```
node counter(reset:bool) returns (count:int)
let
  count = 0 -> if reset then 0 else pre(count)+1;
tel
```



Example : A Resettable Counter

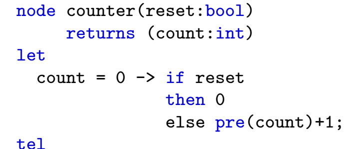


```
node counter(reset:bool)
     returns (count:int)
let
  count = 0 -> if reset
               then 0
               else pre(count)+1;
tel
```

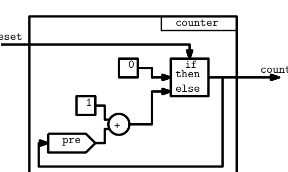


time	0	1	2	3	4	
reset						...
count						...

Example : A Resettable Counter

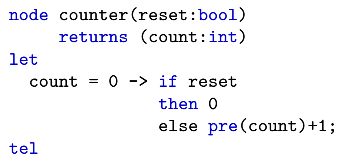


```
node counter(reset:bool)
     returns (count:int)
let
  count = 0 -> if reset
               then 0
               else pre(count)+1;
tel
```

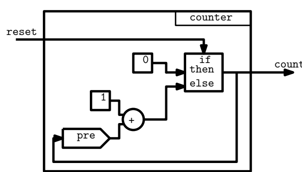


time	0	1	2	3	4	
reset	False					...
count	0					...

Example : A Resettable Counter

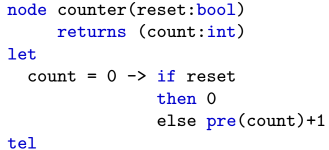


```
node counter(reset:bool)
     returns (count:int)
let
  count = 0 -> if reset
               then 0
               else pre(count)+1;
tel
```

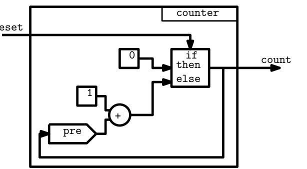


time	0	1	2	3	4	
reset	False	False				...
count	0	1				...

Example : A Resettable Counter

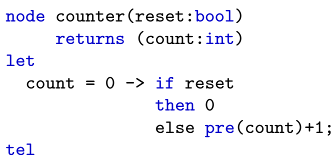



```
node counter(reset:bool)
     returns (count:int)
let
  count = 0 -> if reset
               then 0
               else pre(count)+1;
tel
```

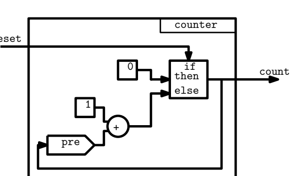



time	0	1	2	3	4	
reset	False	False	False			...
count	0	1	2			...

Example : A Resettable Counter

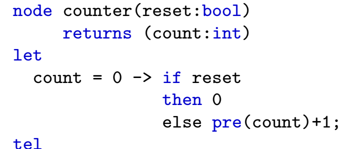



```
node counter(reset:bool)
     returns (count:int)
let
  count = 0 -> if reset
               then 0
               else pre(count)+1;
tel
```

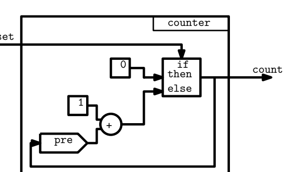



time	0	1	2	3	4	
reset	False	False	False	True		...
count	0	1	2	0		...

Example : A Resettable Counter




```
node counter(reset:bool)
     returns (count:int)
let
  count = 0 -> if reset
               then 0
               else pre(count)+1;
tel
```

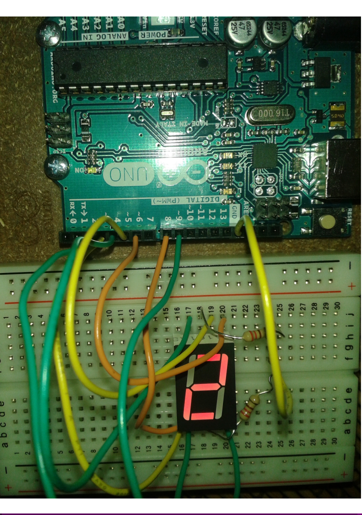
time	0	1	2	3	4	
reset	False	False	False	True	False	...
count	0	1	2	0	1	...

Example 2

Let's count until 42, and reset if a button is pressed, and blink a led one over two cycles.

```
node cpt(reset:bool) returns
     (sevseg:  int; led_on: bool) ;
let
   sevseg = 0  -> if (reset or pre(sevseg = 42))
                  then 0 else pre(sevseg)+1;
   led_on = true -> not pre(led_on);
tel
```


Running example : desired result (teasing Lab3)



Exercise time !

up to you !

Sampling

- The *when* operator under-samples a flow. `x when c` is present only when c is true, in which case its value is x
- The *current* operator replaces absences due to the `when` by the last present value of the flow.

c	True	False	False	True	...
x	x_1	x_2	x_3	x_4	...
`y=x when c`	x_1			x_4	...
`current y`	x_1	x_1	x_1	x_4	...

`current(x when c)`$\neq$`x`

Sampling : example

```
node count3(reset, c:bool) returns (count:int)
  var c1, c2:int;
let
  c1=current(counter(reset when c));
  c2=current(counter2(reset when not c));
  count=if c then c1 else c2;
tel
```

c	True	True	True	False	True	...
count						...

Sampling : an example with pre

```
node p_when(i:int,c:bool) returns (o1,o2:int)
let
  o1=pre(i when c);
  o2=pre(i) when c;
tel
```

c	T	T	F	T	F	F	T	...
i	1	2	3	4	5	6	7	...
o1								...
o2								...

Single assignment

Do not write :

```
node counter(reset:bool) returns (count:int)
let
  if reset then count=0 else count=0->pre(count)+1;
tel
```

▶Each flow is only defined **once**.

Causality

Do not write :

```
node error(reset:bool) returns (count:int)
let
  x=y+1; y=x+2;
tel
```

▶No immediate loop !

- Verified by a *causality analysis* ;
- A `pre` is missing somewhere.

Clocks and undefined values

- Under-sampling (`when`) introduces undefined values, which must not be accessed. The flow is said to be *absent*.
- Only flows with the same clock can be combined : `x+x when c` is forbidden !

x	1	3	5	0	-2	...
c	False	False	True	False	True	...
current (x when c)						...

Trick :

- Use clocks initially true : `x when (true->c)`
- Force a default value

Plan

1 The Synchronous approach

2 The LUSTRE language
Clock sampling
Common errors in Lustre

3 Simulation of LUSTRE programs

4 Compilation of LUSTRE programs

5 Conclusion

Node simulation : chronogram

Demo time (luciole)

```
luciole toto.lus mainnode
```

this is graphical, there is another way to simulate without using old-school graphical packets.

Node simulation : C

Other demo.

```
lus2c toto.lus mainnode -loop
```

generates a .c file

the -compilation- process to obtain this simulation will be explained later

Lab session - Lab 2 - Sept 15th, 2022

On the git.

Plan

1. The Synchronous approach
2. The Lustre language
   - Clock sampling
   - Common errors in Lustre
3. Simulation of Lustre programs
4. Compilation of Lustre programs
5. Conclusion

Lustre in the development cycle

- ✔ Write the synchronous program (formal, high abstraction level);
- **Compile** ▶generation of C code (medium abstraction level)

Valid Programs

Static analyses for checking correctness before the actual code generation :

- Causality (no cycle)
- Initialisation analysis (pre)
- **Clock calculus** ▶No access to absent values.

Clock calculus : type checking

Classical (ML-like) type inference/check : Main ideas :

- each expression has a type for values and another type for its clock ;
- there is a type for the basic clock ;
- a clock type is derived by applying operators on clocks : + does not modify a clock type, but needs its two operands to be compatible ; `pre` defines a subclock of its operands, `when` also.

▶All expressions are typed. If not typable, the compiler rejects. [Equality of conditions is only syntactically checked]

Sequential Code Generation - Goal

Generate a (C) program of the form :

```
init(memory)
each period do
  read(inputs);
  outputs=f(M,inputs);     } step()
  memory=g(M,inputs)
  write(outputs)
done
```

▶Goal here : generate `init`, `f`, `g`, infinite loop

Simple syntax-based code generation

Each node is compiled into a separate procedure :

- flow definition ▶variable assignment ;
- pointwise operator ▶classical operator ;
- `pre`, `->` ▶memories ;
- `when` ▶tests (if).
- sequentialization of the equation system.

Compilation of pre, example

```
node foo(i:int) returns (dec:int)
let
  dec = 0 -> i;
tel
```

compiles into (**new init variable**) :

```
if init then dec = 0 else dec = y;
```

Compilation of our example as a simulation loop

```
node cpt(reset:bool) returns
     (sevseg:  int; led_on: bool) ;
let
   sevseg = 0  -> if (reset or pre(sevseg = 42))
                  then 0 else pre(sevseg)+1;
   led_on = true -> not pre(led_on);
tel
```

`lus2c demo7seg.lus cpt -loop`

▶generates a .c and a main for simulation. (cf Lab)

reset
sevseg
led_on

Simulation loop : main

```
  while(1){
   // std input of data
   cpt_I_reset(ctx, _get_bool("reset"));
   // step, contains std outputs
   cpt_step(ctx);
 }
```

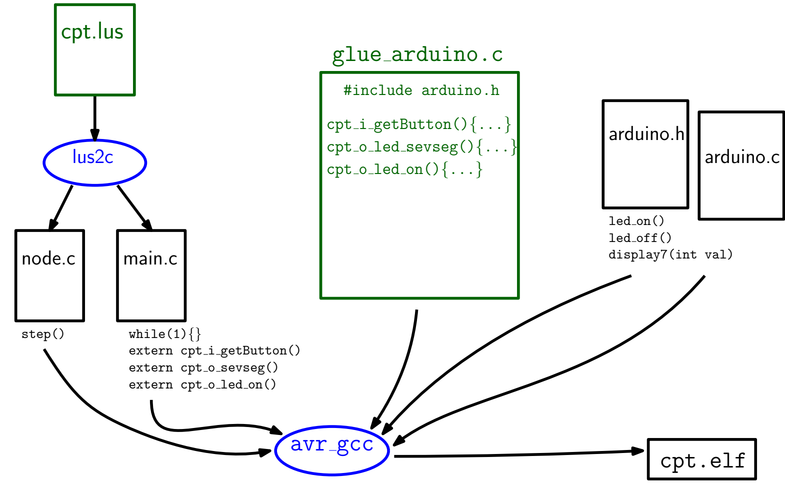
ctx is the state (defined in cpt.c).

What's next ?

- ✔ Write the synchronous program (formal, high abstraction level) ;
- ✔ Compile : it generates C code (medium abstraction level) ;
- Write the **integration program** :
  - Read inputs on sensors ;
  - Call the synchronous program ;
  - Apply outputs to actuators.

  ▶Lab

Compilation chain for arduino backend



Glue code 1/2 : 1 input function

```
/* Main loop */
while(1){
  //input
  cpt_I_reset(ctx, _get_reset_value());
  //step
  cpt_step(ctx);
  //delay
  _delay_ms(1000);
}
```

```
int _get_reset_value(){
  button_state=digitalRead(button);
  return (button_state == HIGH);
}
```


Glue code 2/2 : 2 output functions

```
void cpt_step(cpt_ctx* ctx){
  cpt_O_sevseg(ctx->data, L1);// <-- 7seg command
  cpt_O_led_on(ctx->data, L14); // <-- led command
}
```

7seg-command :

```
void cpt_O_sevseg(void* cdata, _integer _V) {
  displayDigit(_V);// <-- arduino's lib for 7 seg
}
```

Led command :

```
void cpt_O_led_on(void* cdata, _boolean _V) {
  if (_V == 1)
    digitalWrite(led, HIGH);// <-- arduino's lib for led
  else
    digitalWrite(led, LOW);
}
```


Running example

In the lab.

Plan

1. The Synchronous approach
2. The Lustre language
   - Clock sampling
   - Common errors in Lustre
3. Simulation of Lustre programs
4. Compilation of Lustre programs
5. Conclusion

Take-out message

For real-time :

- Programming reactive loops can be error-prone, we favor high level languages.
- Safety is important (see next course). Timing AND functional constraints.

▶Synchronous languages are one (among other) solution.

Credits

Slides used for a common talk with Lionel Morel, INSA Lyon.

- *Julien FORGET (Cristal)* teaching notes
- *Abdoulaye GAMATIE (LIRMM)*, Synchronous Programming of Real-Time Systems with the Signal language, course at Telecom Lille 1, 2012 ;
- *Pascal RAYMOND (Verimag)*, various teaching notes.
- *Florence MARANINCHI (Verimag)*, Arduino inspiration.

Applications Synchrones (CS442)

Lustre: formal verification

Laure Gonnord

Grenoble INP/Esisar

2022-2023

Plan

1. Formal verification of LUSTRE programs
   - Formal verification with synchronous observers
   - Synchronous observers : how is it working ?
   - Bonus : a technique to (model- check) boolean circuits
2. Lustre program verification for numerical properties
3. Conclusion

Formal verification, what for ?

- Reactive/real time systems are **critical**
- We want strong guarantees.

▶Both **functional** and timing properties.

Timing properties

See WCET (next) course. The analysis are all **sound**.

Functional properties, how ?

Different approaches :

- Test the generated code on scenarios ▶not complete
- Formal verification of the generated code ?
- Formal verification of the source code ?
- Something in the middle ?

Recall : the Implicit State Machine

Used implicitely during the compilation process.

$\ldots I_i \ldots$ → Controller $\ldots M_i \ldots$ → $\ldots O_i \ldots$

Synchronous programming model

- The **output function** $O_i = f(I_i, M_i)$;
- The **transition function** $M_{i+1} = g(I_i, M_i)$

This is equivalent to an explicit transition system, where : states are all possible values of the memory and :

$$q \longleftrightarrow^{i/o} q' \text{ iff } q' = g(i, q) \text{and} o = f(i, q)$$

Computing an explicit automaton from a lustre node

```
node edge (b : bool) returns (edge : bool);
let
    edge = false -> b and not pre b;
tel
```

1) By hand (memories==pre==registers)
2) `lus2atg`. Demo. (be careful, there seems to be a bug inside the atg viewer).

▶

Credits

Slides from P. Raymond, Verimag, for the MOSIG M2.

Example: the beacon counter in a train

- Counts the difference between beacons and seconds
- Decides whether the train is late, early or ontime
- Hysteresis to avoid oscillations

```
node b(sec,bea: bool) returns (ontime,late,early: bool);
var diff: int;
let
  diff = 0 -> pre ( diff +
      (if bea then 1 else 0) + (if sec then -1 else 0));
  early = false -> pre (
      (ontime and diff > 3) or (early and diff > 1));
  late = false -> pre(
      (ontime and diff < -3) or (late and diff < -1));
  ontime = not (early or late);
tel
```

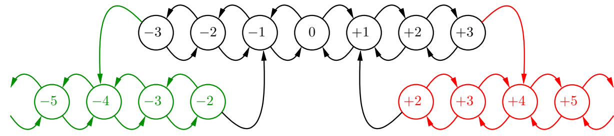

- It's impossible to be late and early
- It's impossible to directly pass from late to early
- It's impossible to remain late only one instant
- If the train stops, it will eventually get late

The 3 first ones are obviously safety, while the one is a typical liveness: it refers to unbounded future

Example: beacon counter

- I = {sec, bea} O = {late, ontime, early}
- A memory for each "`-> pre`" expression,
  (e.g. Plate for "`false -> pre late`"):
  M = {Plate, Pontime, Pearly, Pdiff}
  with M_0 = (*false*, *true*, *false*, 0)
- Functions directly given by the Lustre equations

A small part of the explicit automaton:



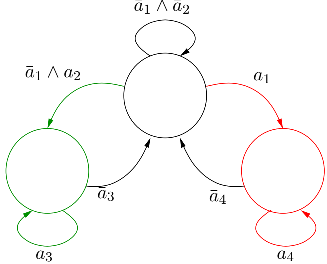
Model and verification

The explicit automaton is the set of behavior,
so exploring the automaton is checking the program
Problem: The automaton may be infinite, or at least enormous,
it is impossible to explore it
Idea: work on a finite (not too big) abstraction of the program N.B. the abstraction must conserve at least some properties (otherwise it's useless)

Example

Abstraction of numerical comparisons in the beacon counter, they become "free" boolean variables:

- a_1 for `diff > 3`
- a_2 for `diff < -3`
- a_3 for `diff < -1`
- a_4 for `diff > 1`



- It's impossible to be late and early (safety)
- It's impossible to directly pass from late to early (safety)

Lost properties

- It's impossible to remain late only one instant (safety)
- If the train stops, it will eventually get late (liveness)

More serious: introduced property

- It's possible to remain late only one instant (liveness): true on the abstraction, false on the real program !

⇒ Important to precisely know what is preserved by the abstraction

Abstraction and safety

- Finite abstraction is a special case of over-approximation
- Anything which is impossible in the abstraction is impossible on the program
- The counterpart is (in general) false

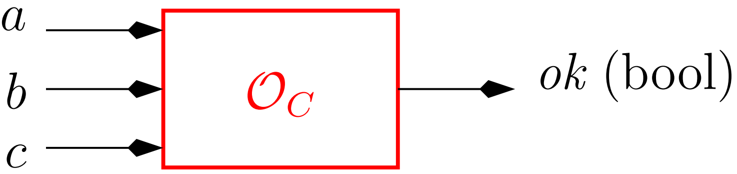
⇒ safeties are preserved or lost, but never introduced

As a consequence, when checking a safety on the abstraction:

- the verification succeeds ⇒ property satisfied
- the verification fails ⇒ inconclusive (it may be a *false negative*)

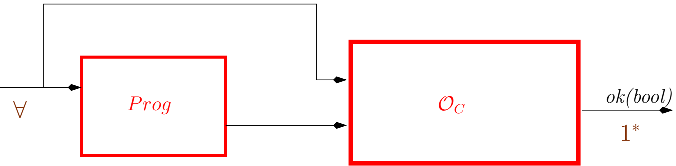
Lustre verification with observers, principle (1)

- A property is specified in the same language as the program.
- The "execution" is done in parallel.



▶**Goal :** prove $L(\neg C) \cap L(Prog) = \emptyset$

General scheme of verification with observers



Model checking of the 20th century - demos

An example with `xlesar` and the node :

```
node edge (b : bool) returns (edge : bool);
let
     edge = false -> b and not pre b;
tel
```

Let us try to prove :

- `true -> (edge => not pre edge)`
- `b => edge`

Example (in Lustre)

- It's impossible to be late and early:
  `ok = not (late and early);`
- It's impossible to directly pass from late to early:
  `ok = true -> not (early and pre late);`
- It's impossible to remain late only one instant:

```
Plate = false -> pre late;
PPlate = false -> pre Plate;
ok = not (not late and Plate and not PPlate);
```

Let see a quick demo ...

Convenient to split property into assumption/conclusion:

"if the train keeps the right speed, it remains on time"

property is simply `ok = ontime`, assumption can be:

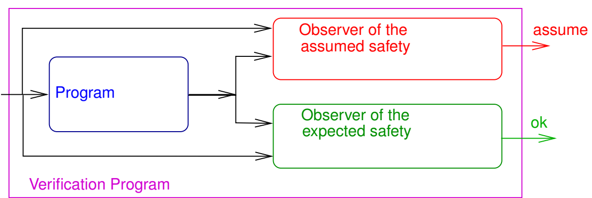
- naive: `assume = (sec = bea);`
- more sophisticated, bea and sec alternate:

```
SF = switch(sec and not bea, bea and not sec);
BF = switch(bea and not sec, sec and not bea);
assume = (SF => not sec) and (BF => not bea);
```

with:

```
node switch(on, off : bool) returns (s: bool);
let s = false -> pre(if s then not off else on); tel
```

General scheme



- We suppose provided such a verification program
- Goal: if assume remains indefinitely true, then ok remains indefinitely true:
  (always `assume`*)* $\Rightarrow$ *(always* `ok`*)*
- Note: it is NOT a "regular" safety, so in a first step, we approximate it by:
  always ((once not `assume`*) or* `ok`*)*
  (the problem will be explained later)

Abstracted verification program

Special case of Boolean synchronous program with 2 "outputs"

- Free variables V, state variables S
- Initial state(s): Init $: \mathbf{B}^{|S|} \to \mathbf{B}$
- Transition functions: $g_k : \mathbf{B}^{|S|} \times \mathbf{B}^{|V|} \to \mathbf{B}$ for $k = 1 \cdots |S|$
- Assumption: $H : \mathbf{B}^{|S|} \times \mathbf{B}^{|V|} \to \mathbf{B}$
- Property: $\phi : \mathbf{B}^{|S|} \times \mathbf{B}^{|V|} \to \mathbf{B}$

(N.B. we identify predicates and sets)

Associated explicit automaton

We note $Q = \mathbf{B}^{|S|}$ the state space
We use "pre" and "post" functions:

- for $q \in Q$, $\text{post}_H(q) = \{q' / \exists v \;\; q \xrightarrow{v} q' \;\wedge\; H(q, v)\}$
- for $X \subseteq Q$, $\text{Post}_H(X) = \bigcup_{q \in X} \text{post}_H(q)$
- for $q \in Q$, $\text{pre}_H(q) = \{q' / \exists v \;\; q' \xrightarrow{v} q \;\wedge\; H(q', v)\}$
- for $X \subseteq Q$, $\text{Pre}_H(X) = \bigcup_{q \in X} \text{pre}_H(q)$

- Initial state(s): $\text{Acc}_0 = \{q / \text{Init}(q)\}$
- Error states: $\text{Err} = \{q / \exists v \quad H(q, v) \wedge \neg\phi(q, v)\}$
- Reachable states: $\text{Acc} = \mu X \cdot (X = \text{Init} \cup \text{Post}_H(X))$
- Bad states: $\text{Bad} = \mu X \cdot (X = \text{Err} \cup \text{Pre}_H(X))$

Goal

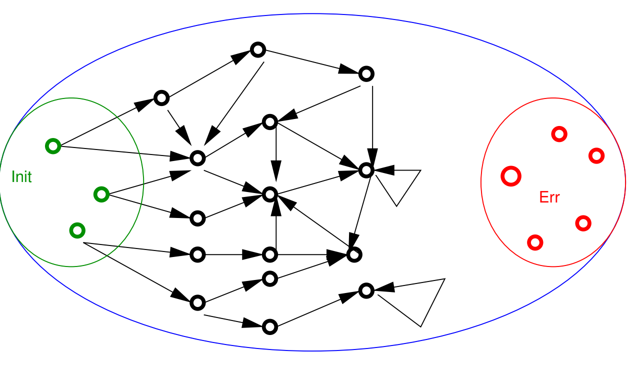
Naive: prove that $\text{Acc} \cap \text{Bad} = \emptyset$
No need to compute both Acc and Bad:

- prove that $\text{Acc} \cap \text{Bad}_0 = \emptyset$ (forward method)
- prove that $\text{Bad} \cap \text{Acc}_0 = \emptyset$ (backward method)

Remark: methods are non symmetric because of determinism

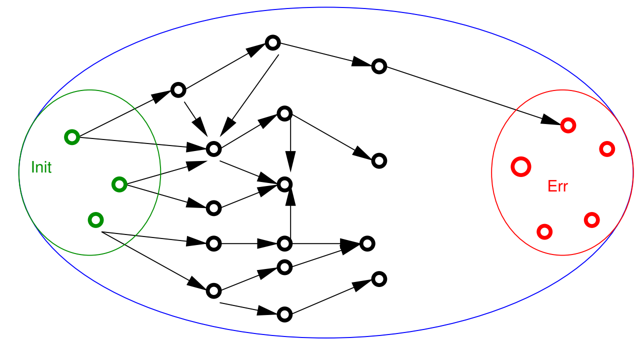
Enumerative (forward) algorithm

```
CurAcc := Init
Done := empty
while it exits q in CurAcc - Done do {
    (* q ∈ CurAcc \ Done *)
    for all q' in post_H(q) do {
        if q' in Bad0 then EXIT(failed)
        put q' in CurAcc
    }
    put q in Done
}
(* we have CurAcc = Done = Acc *)
EXIT(succeed)
```

success

Example of failure (breadth first)



Failure

Notes on implementation

- depth first, breath first, or other
- compact encoding of states
- very costly:
  $|Acc|$ times the cost of post$_H$(q), with $|Acc| \sim 2^{|S|}$
- backward is even worse: pre$_H$(q) is more complex than post$_H$(q)
  (enumerative backward is never used in practice)

Big problem: computing post$_H$(q)

- For a given q, find all v s.t. $H(q, v)$
- Typical decision problem (NP-complete)
- Naive method: try all $2^{|V|}$ possible values
- Need for non trivial, efficient decision procedure

⇒ Digression on efficient decision techniques

Circuit, an example ?

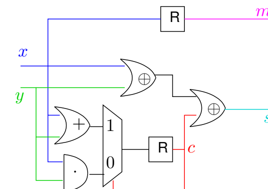
Credits : P. Raymond

Serial adder:

- inputs x, y
- outputs s(um), c(arry)

Shift:

- m encodes $2 \times x$



time (most significant bits)

c	0	0	1	0	
x	0	1	0		(2)
y	1	1	0		(3)
s	1	0	1		(5)
m	0	0	1		(4)

Property: if always $x = y$ then always $s = m$

Serial adder, questions ...

- Give the (implicit) automaton of the system
- Explore the system with the enumerative method (by "hand")
  (and prove that "always(x=y) ⇒ always(s = m)")

- *2 inputs* $V = \{x, y\}$
- *2 memories* $S = \{c, m\}$ *with*
  $c_{init} = 0, g_c = c.(x + y) + \bar{c}.x.y$
  $m_{init} = 0, g_m = x$
- $H \equiv (x = y)$, *and* $\phi \equiv (m = s)$, *where* $s = (c \oplus x \oplus y)$

Enumerative exploration (the "tabular method")

Note: we have "pre-computed" that $x = y$ *are the only possible inputs*

Starting state		Inputs		Output/Prop		Next state	
c	m	x	y	s	ϕ	c'	m'
0	0	0	0	0	1	0	0
		1	1	0	1	1	1
1	1	0	0	1	1	0	0
		1	1	1	1	1	1

Serial adder, questions (cntd)

- Explore the system with the symbolic method
  n.b. hardly feasible by hand, need an helper: `bddc`
- Use `bddc` (basic BDD calculator)
- How it works: reads formula, build (and echo if possible) the corresponding BDD
  ↪ `x or (y xor z);` outputs: `x + y.-z + -y.z`
  ↪ `x => (y => x);` outputs: `1` (canonical form)
- Assign a formula to a "variable"
  ↪ `s := c xor x xor y;`
- Define a function over formulae
  ↪ `Implique(X,Y) := not X or Y;`
- Usefull commands: `help` and `syntax`
- ... quick demo.

- *2 inputs* $V = \{x, y\}$
- *2 memories* $S = \{c, m\}$ *with*
  $c_{init} = 0, g_c = c.(x + y) + \bar{c}.x.y$
  $m_{init} = 0, g_m = x$
- $H \equiv (x = y)$, *and* $\phi \equiv (m = s)$, *where* $s = (c \oplus x \oplus y)$
- *Error states: Err* $\equiv (\exists x, y \;\; H \wedge \neg\Phi) \;\; \equiv (c \oplus m)$

In bddc syntax ...

```
Gm   := x;
Gc   := if c then (x or y) else (x and y);
s    := x xor y xor c;
Init := not c and not m;
H    := (x = y);
Phi  := (m = s) ;
Err  := exist x,y (H and not Phi);
Acc0 := Init;
```


Step 0

- *Check that* $Acc_0 = Init \cap Err = \emptyset$

`Acc0 and Err;`

gives 0*, ok, continue and compute* Acc_1

Step 1

- $Acc_1 = Acc_0 \cup post_H(Acc_0)$
- *Recall the definition of* $Post_H$ *(slide 50)*

`Post(A) := exist x,y,m,c (A and H and (xm = Gm) and (xc = Gc));`

- *Computes:*

`Post(Acc0);`

gives: `xm.xc + -xm.-xc`*, i.e.* `xm = xc`

- *Warning, technical problem: we need a formula on c and m (not xc and xm)*

- *Trick, use a "rename" function:*

`Rnm(a,b,F) := exist a (F and (a = b));`

- *The "right" definition of Post:*

`Postbis(X) := Rnm(xc,c, Rnm(xm,m, Post(X)));`

- *Check that:*

`Postbis(Acc0);` *gives:* `m.c + -m.-c`*, i.e.* `m = c`

- *Compute:*

`Acc1 := Acc0 or Postbis(Acc0);`

- *Are* Acc_0 *and* Acc_1 *the same ?*

`compare(Acc1,Acc0);`

answers 0 (not the same), fixpoint not reached...

- *Check:*

`Acc1 and Err;`

gives empty, no error yet ...

Step 2

- *Compute* Acc_2*:*

```
Acc2 := Acc1 or Postbis(Acc1);
```

- *Check:*

```
compare(Acc2,Acc1);
```

answers 1 (same), fixpoint is reached !

Property satisfied,

we have proven formally *that*

$\forall x, y \in \mathbb{Z} \quad (x = y) \Rightarrow (x + y \; = \; 2x)$

Plan

1. Formal verification of Lustre programs
   - Formal verification with synchronous observers
   - Synchronous observers : how is it working ?
   - Bonus : a technique to (model- check) boolean circuits
2. Lustre program verification for numerical properties
3. Conclusion

And for numeric properties ?

Some ideas :

- The Automaton is infinite in the general case.
- Its construction can be helped by the property to prove (*var* ≤ 2 gives 2 states, ...
- The enumerated forward strategy can be approximated (Abstract interpretation, here `nbac`).

Plan

1. Formal verification of Lustre programs
   - Formal verification with synchronous observers
   - Synchronous observers : how is it working ?
   - Bonus : a technique to (model- check) boolean circuits
2. Lustre program verification for numerical properties
3. Conclusion

Take-out message

- Formal verification through the help of automata
- But we should deal with the *state explosion problem*
- Modern solvers to the rescue !

▶**numerical properties** are out of the scope here.

Applications Synchrones (CS442)

Lustre: timing verification (WCET)

Laure Gonnord

Grenoble INP/Esisar

2022-2023

Plan

1. Introduction
2. WCET d'une tâche
   - Analyse de flot
   - Analyses haut-niveau : calcul final du WCET
   - Analyses bas-niveau : calcul de WCET par bloc
3. Bornes des coûts de préemption

Contexte - Système embarqué

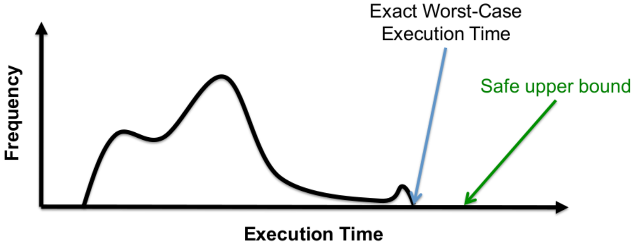
Ce qu'on a fait jusqu'à présent :

- Écrit les tâches (leur fonctionnalité), en Lustre (et vérifié ces tâches en isolation ou en combinaison) *dans CS442*. (+ de tests, de validation dans le contexte du cours SE515 - 5A ISE)
- Ordonnancé en supposant leur période d'exécution connue. *dans le cours de real-time 4A*.

Crédits

- Isabelle Puaut/ Damien Hardy pour Univ. Rennes
- Exemples et Dessins : Claire Maiza (Grenoble), Damien Hardy (Rennes), Reinhard Wilhelm and Jan Reineke (Univ. Saarland).
- Ordonnancement dans les systèmes temps-réel, coord. Maryline Chetto, chapitre WCET (C. Maiza, P. Raymond, C. Rochange)

Définitions : temps d'exécution pire cas



WCET = Worst Case Execution Time

▶Attention à ne pas confondre pire-cas et **estimation** du pire cas.

Dessin D. Hardy (Rennes)

De quoi dépend le temps d'exécution ?

- Des chemins d'exécution dans le programme, qui eux-même dépendent des **entrées** (inconnues).
- De l'architecture matérielle : la durée d'une action dépend du matériel (et de son historique d'exécution).

Validation des systèmes temps-réel durs

Que faut-il garantir ?

- Validation au niveau système : connaissance pire cas de l'impact du système (interruptions, etc.)
- Validation au niveau tâche : exec pire cas de chaque tâche individuelle. Le code est considéré en isolation.

Plan

1 Introduction

2 WCET d'une tâche
   Analyse de flot
   Analyses haut-niveau : calcul final du WCET
   Analyses bas-niveau : calcul de WCET par bloc

3 Bornes des coûts de préemption

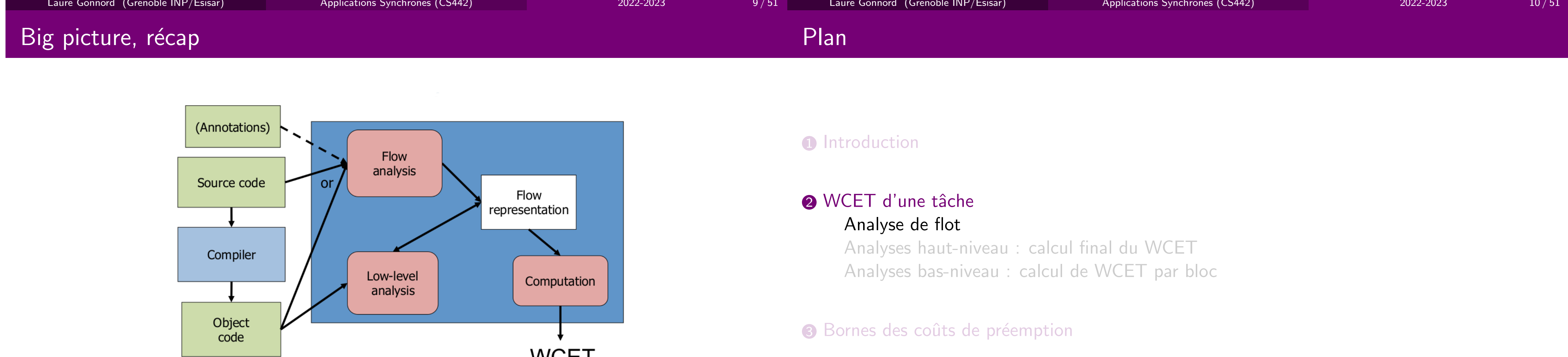
Challenges de l'estimation du WCET

- **Sûreté** (safety) : l'estimation est plus grande que le "vrai" WCET ▶**essentiel** pour les programmes critiques De plus une erreur signifie qu'on se plante dans l'analyse d'ordonnançabilité après.
- **Précision** : en cas de trop grande surapproximation ▶test d'ordonnançabilité plante ou, trop de ressources utilisées.

Principes des analyses statiques de WCET

- Principe : Analyse basée sur la structure du programme (sans exécution).
- 3 composants principaux :
  1. Récupération du flot du programme
  2. Analyse bas niveau ▶récup d'un WCET par block **Analyse hardware-dépendante**
  3. Calcul final en considérant **tous les chemins**

Big picture, récap



Dessin D. Hardy (Rennes) Il manque une flèche entre lowlevel et computation.

Plan

1 Introduction

2 WCET d'une tâche
- Analyse de flot
- Analyses haut-niveau : calcul final du WCET
- Analyses bas-niveau : calcul de WCET par bloc

3 Bornes des coûts de préemption

Analyses de base/ Autres analyses

Rappel :toutes les analyses sont statiques :

- Extraction d'un graphe de flot (à partir du code source ou du binaire). **assez simple**.
- Calcul du nombre d'itération des boucles. **pas simple**
- (Optionnel) Chemins infaisables. **ça dépend**

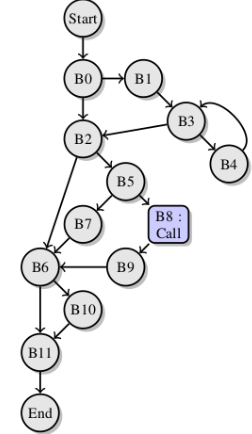
Un programme exemple (step)

Crédit exemple + dessins C. Maiza (Grenoble)

```
void Step () {
  if (init){
    for (i = 0 ; i < 10 ; i ++){
      tab [i]=0;
    }
  }
  if (not (idle)){ // idle et low = etats
    if (low) {     // le systeme n'a plus de batterie
      S=X+Y ;      // X et Y = capteurs
    }
    else {
      insert (X, tab) ;
      S=tab[0]+Y ;
    }
  }
  if (init){
    init=false ;
  }
}
```


Graphe de flot - (programme step)

```
void Step () {
  if (init){
    for (i = 0 ; i < 10 ; i ++){
      tab [i]=0;
    }
  }
  if (not (idle)){
    if (low) {
      S=X+Y ;
    }
    else {
      insert (X, tab) ;
      S=tab[0]+Y ;
    }
  }
  if (init){
    init=false ;
  }
}
```



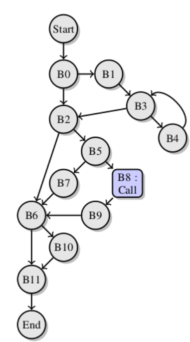
Calcul de borne d'un nid de boucle

```
void Step () {
  if (init){
    for (i = 0 ; i<10 ; i++){
      tab[i]=0;
    }
  }
}
```

Calculer le nombre exact est indécidable. ▶Approximations. Méthodes plus ou moins syntaxiques.
Voir [1] pour un exemple de telle analyse.

Calcul de chemins infaisables (infeasible paths)

```
void Step () {
  if (init){
    ...
   }
  } ...
  if (low) {  ...}
  }
  if (init){
    ...
  }
}
```



- `init` inchangé entre les deux tests ⇒ transitions mutuellement exclusives (B0-B1 et B6-B11)
- Si on sait `init => low`, alors on peut enlever un long chemin. (les outils utilisent des annotations)

▶Voir [3] pour un exemple de tel analyse. On gagne beaucoup.

What's next ?

1. Calcul du WCET par bloc (plus tard).
2. On a le flot (ou l'AST) et des WCET par bloc ▶calculer le WCET total.

On suppose des architectures simples, où la durée d'une instruction dépend seulement de l'instruction et des opérandes. Pas d'*overlap* entre les instructions.

Plan

1 Introduction

2 WCET d'une tâche
   Analyse de flot
   Analyses haut-niveau : calcul final du WCET
   Analyses bas-niveau : calcul de WCET par bloc

3 Bornes des coûts de préemption

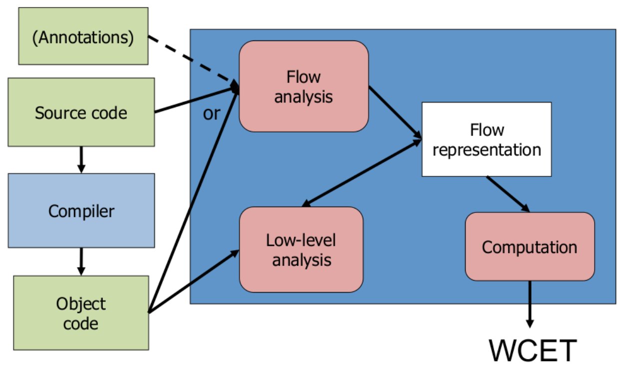
Rappel du problème

Le problème

On a le flot (ou l'AST) et des WCET par bloc. On veut calculer le WCET total.

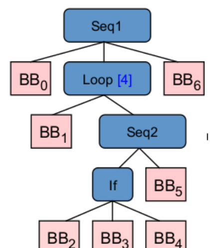
▶2 méthodes

Sur la récap.



Méthode 1 : sur l'AST

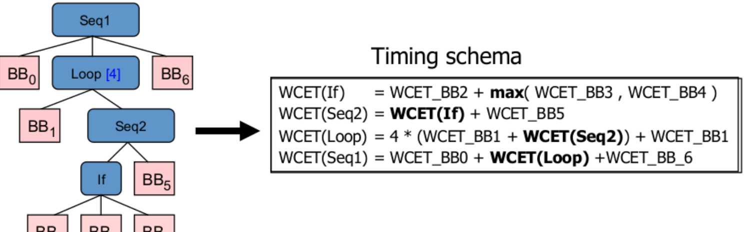
```
int x, p=0; i=0;
for (x=0; x<4;x++){
  if (i mod 2) p++
  else i++;
}
```



On dispose des WCET des blocs, puis :

- WCET d'une séquence : somme des WCET
- WCET (if) : WCET(test) + max des 2 WCET des branches
- WCET(while) : nbboucles * (WCET(test) + WCET (body) + WCET (inc))

Méthode 1 : exemple



Méthode 1 : avantages/inconvénients

- Avantages : pas cher ! passe bien à l'échelle, retour au source facile.
- Inconvénients : pas compatible avec des optimisations "agressives" du compilateur.

Méthode 2 : sur le CFG - IPET

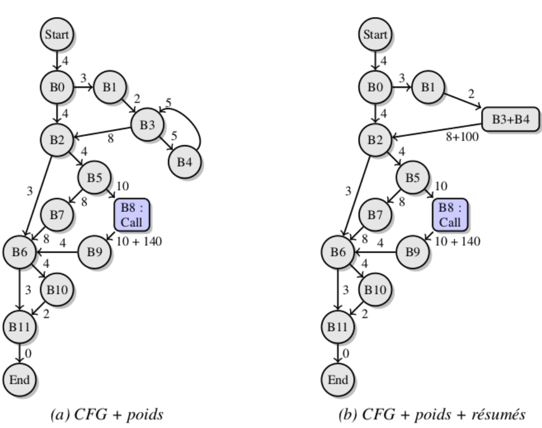
IPET = Implicit path enumeration technique

On écrit une instance d'un **problème linéaire en nombre entiers** (ILP) :

- Données : T_i les WCET des blocs.
- Une variable numérique $x_{i,j}$ par transition qui compte le nombre de fois où l'on passe par celle-ci.
- La fonction objectif est $Max \sum f_i T_i$.
- Contraintes : pour tout bloc, somme des entrants = somme des sortants.
- Cas des boucles : rajouter le nb max dans les contraintes
- Les infos de flot rajoutent des contraintes.

Méthode 2 : exemple

Exemple C. Maiza, livre "Ordonnancement pour les systèmes temps-réel"

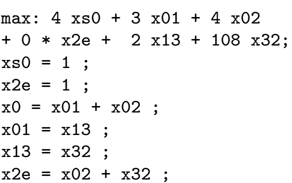


(a) CFG + poids (b) CFG + poids + résumés

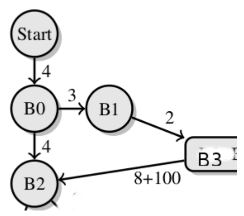
▶Exo : écrire le (début) du Programme Linéaire.

Méthode 2 : problème LP simplifié et lpsolve

(LP solve format)



```
max: 4 xs0 + 3 x01 + 4 x02
+ 0 * x2e +  2 x13 + 108 x32;
xs0 = 1 ;
x2e = 1 ;
x0 = x01 + x02 ;
x01 = x13 ;
x13 = x32 ;
x2e = x02 + x32 ;
```



▶Utilisation de `lpsolve` (si il n'y a pas de boucle la solution est en 0/1) :

```
$ lp_solve mif25.lp

Value of objective function: 117

Actual values of the variables:
xs0                          1
x01                          1
x02                          0
...
```


Méthode 2 : avantages/inconvénients

- Avantages : supporte tous les types de graphes de flots, super optimisés ou complétement instructurés . . .
- Inconvénients : coûte plus cher, moins de feed-back, les annotations sont difficiles à propager (dans le cas binaire).

▶Utilisé dans la plupart des outils commerciaux/académiques

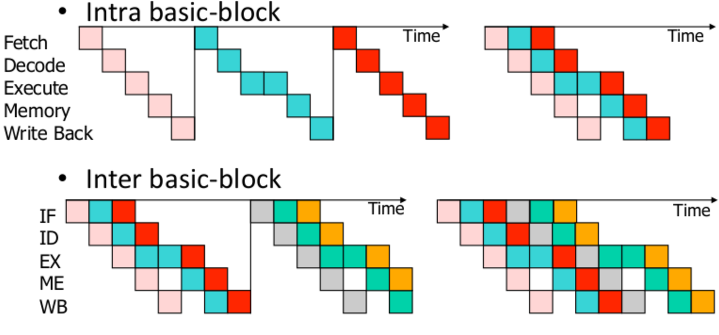
Plan

1. Introduction
2. WCET d'une tâche
   - Analyse de flot
   - Analyses haut-niveau : calcul final du WCET
   - Analyses bas-niveau : calcul de WCET par bloc
3. Bornes des coûts de préemption

Problématiques

- Effets locaux : pipeline (au niveau instruction)
- Effets globaux : cache, prédicteurs de branchement (il faut une connaissance entière du code).

Pipeline 1/3



crédit image D. Hardy (Rennes)

Pipeline 2/3 : intra-basic blocks

Modification du calcul de WCET d'un bloc, notion de table de réservation.

	1	2	3	4	5	6	7	8	9
LI	a	b	c	d					
DLE		a	b	c		d			
EX			a	b		c	d		
M				a	b		c	d	
ER					a	b		c	d

```
a: add r0,r15,#128
b: ldrb r1,[r]
c: cmp r1,#0
d: bne _low0
```

▶On prend en compte le pipeline dans le WCET d'un bloc de base.

crédit image C. Maiza (Grenoble)

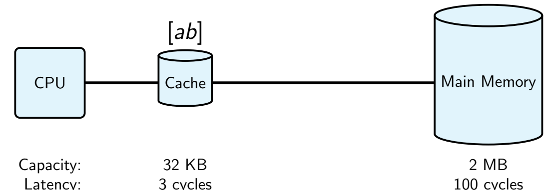
Pipeline 3/3 : inter-basic blocks

Modification du calcul global du WCET :

- (méthode AST) on remplace l'addition des WCET par une addition plus “intelligente”
- (méthode CFG) on ajoute des contraintes négatives sur certaines transitions dans le problème LP.

Caches, rappels d'archi

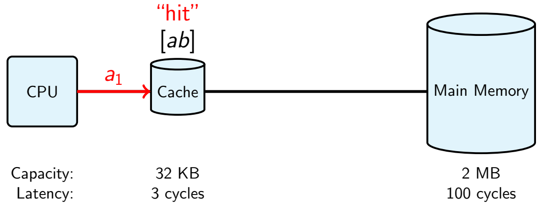
Crédit images cache + analyses de cache : J. Reineke, R. Wilhelm



▶Petites parties de mémoire rapides qui permettent d'exploiter la **localité spaciale** et temporelle.

Caches, rappels d'archi

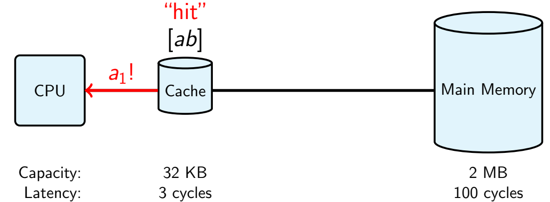
Crédit images cache + analyses de cache : J. Reineke, R. Wilhelm



▶Petites parties de mémoire rapides qui permettent d'exploiter la **localité spaciale** et temporelle.

Caches, rappels d'archi

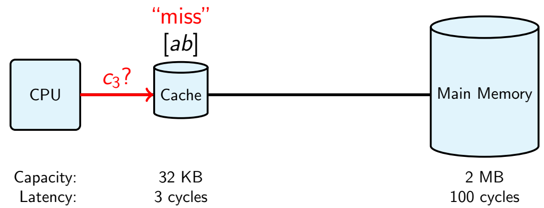
Crédit images cache + analyses de cache : J. Reineke, R. Wilhelm



▶Petites parties de mémoire rapides qui permettent d'exploiter la **localité spaciale** et temporelle.

Caches, rappels d'archi

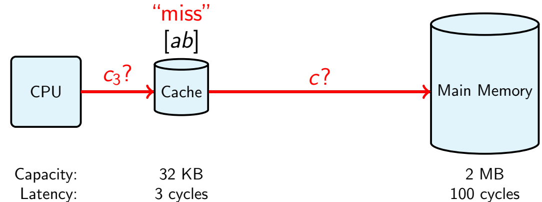
Crédit images cache + analyses de cache : J. Reineke, R. Wilhelm



▶Petites parties de mémoire rapides qui permettent d'exploiter la **localité spaciale** et temporelle.

Caches, rappels d'archi

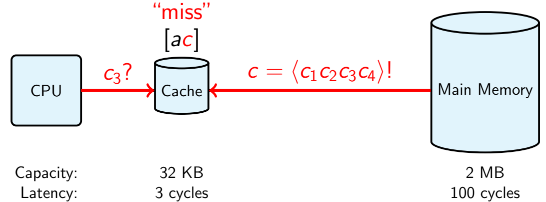
Crédit images cache + analyses de cache : J. Reineke, R. Wilhelm



▶Petites parties de mémoire rapides qui permettent d'exploiter la **localité spaciale** et temporelle.

Caches, rappels d'archi

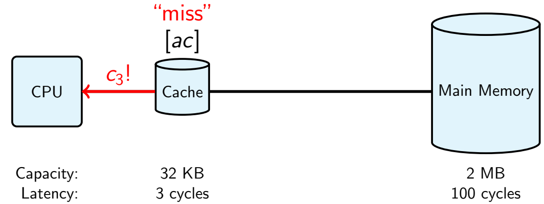
Crédit images cache + analyses de cache : J. Reineke, R. Wilhelm



▶Petites parties de mémoire rapides qui permettent d'exploiter la **localité spaciale** et temporelle.

Caches, rappels d'archi

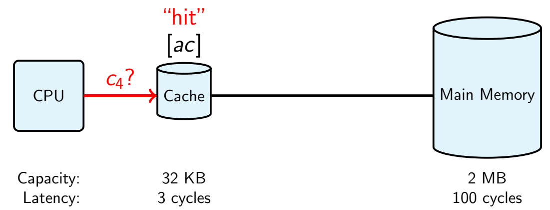
Crédit images cache + analyses de cache : J. Reineke, R. Wilhelm



▶Petites parties de mémoire rapides qui permettent d'exploiter la **localité spaciale** et temporelle.

Caches, rappels d'archi

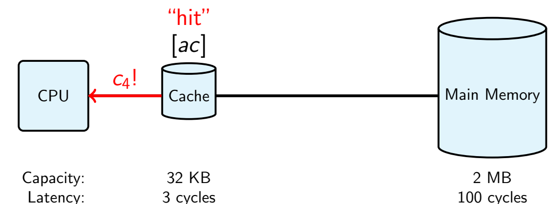
Crédit images cache + analyses de cache : J. Reineke, R. Wilhelm



▶Petites parties de mémoire rapides qui permettent d'exploiter la **localité spaciale** et temporelle.

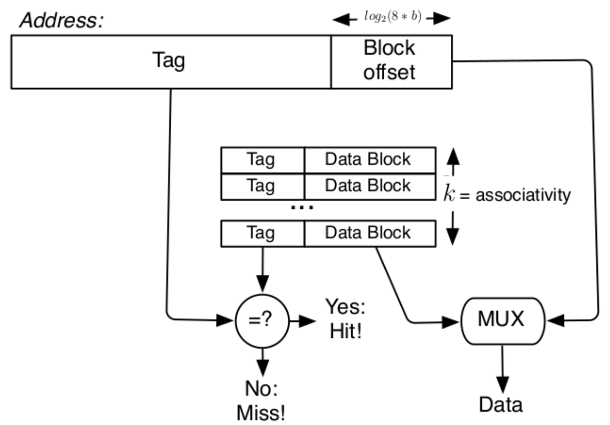
Caches, rappels d'archi

Crédit images cache + analyses de cache : J. Reineke, R. Wilhelm

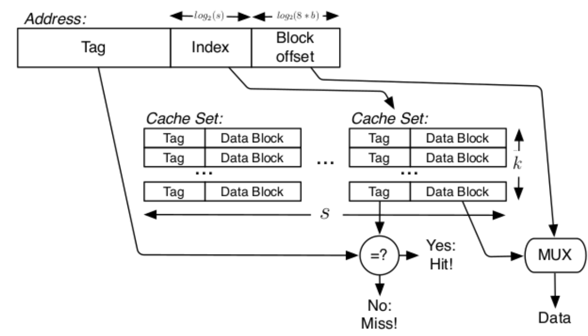


▶Petites parties de mémoire rapides qui permettent d'exploiter la **localité spaciale** et temporelle.

Différents types de caches 1/3



Différents types de caches 2/3



Différents types de caches 3/3

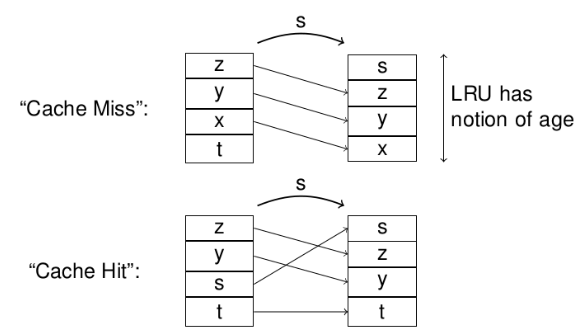
Politiques de remplacement :

- Least-Recently-Used (LRU) : Pentium 1, MIPS
- First-In First-Out (FIFO) : Intel XS CALE , ARM9, ARM11, ...
- Pseudo-LRU (PLRU) : Intel Pentium II-IV et POWER PC 75 X (voir plus tard)
- Most-Recently-Used (MRU) : meilleur dans le cas de "repeated scans over large datasets"

▶On veut prédire les "hit" et les "miss".
On considère que les set associatifs sont des compositions de caches totalement associatifs.

Analyses de cache pour LRU, principes

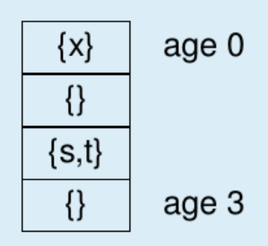
Comportement **concret** pour LRU :



Must analysis : prédire les cache hits

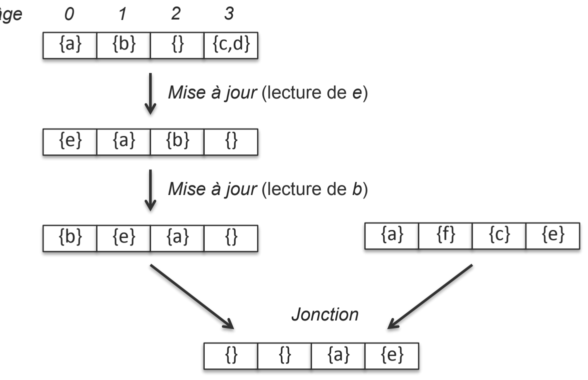
Idée : on maintient des bornes supérieures sur les âges du contenu du cache (si cette borne est inférieure à l'associativité, cela signifie que la donnée est **vraiment** dans le cache)

Exemple de **valeur abstraite**:



▶x a un âge 0, $age(s) \leq 2$, ...

Must analysis, opérations

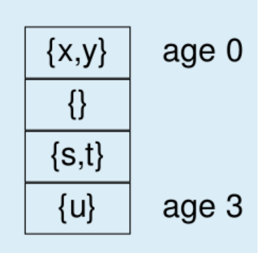


▶Itérer jusqu'à point fixe. Recherche active pour augmenter la précision. Cette technique s'appelle l'interprétation abstraite. C'est le couteau suisse des analyses statiques.

May analysis : prédire les cache misses (défauts)

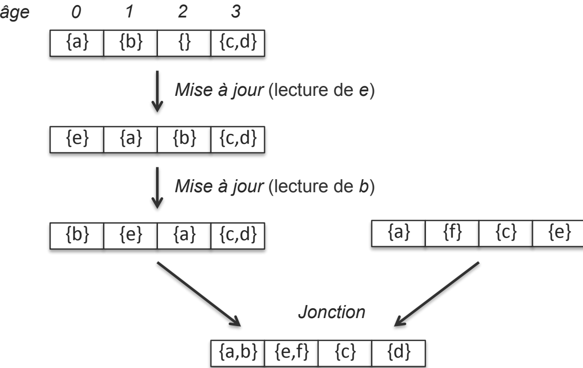
Idée : on maintient des bornes inférieures des âges du contenu du cache (si $\geq$ associativité, cela signifie que la donnée n'est **vraiment pas** dans le cache)

Exemple de **valeur abstraite**:



▶ x, y ont un âge ≥ 0, $age(s,t) \geq 2$, ...

May analysis

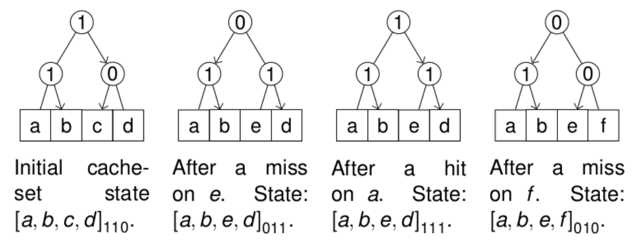


Au delà - MRU

Des bits se rappellent des lignes récemment utilisées :

Au delà - Pseudo LRU

"Sélectionner un élément qui n'a probablement pas été accédé récemment" : arbres de recherche (0 = gauche, 1 = droite)



▶ le voisinage est recalculé $\Rightarrow$ b reste dans le cache.

Plan

1. Introduction
2. WCET d'une tâche
   - Analyse de flot
   - Analyses haut-niveau : calcul final du WCET
   - Analyses bas-niveau : calcul de WCET par bloc
3. Bornes des coûts de préemption

La préemption, pourquoi ? (rappel)

- Souvent la préemption permet l'ordonnançabilité d'un ensemble de tâches.
- Souvent les tâches à échéance courte ne sont pas ordonnançables sans préemption.

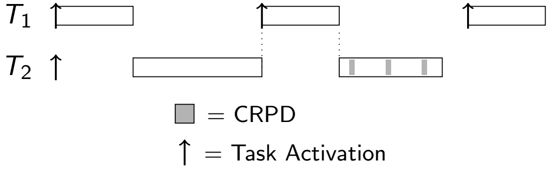
Exemple : Soient les tâches T_1 et T_2, de périodes $P_1 = 2, P_2 = 8$, échéances $D_1 = P_1, D_2 = P_2$, et temps d'exécution $C_1 = 1, C_2 = 3$.

Dessins !

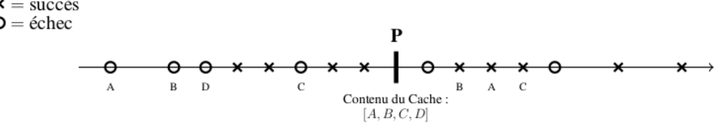
Rappel : il existe des jeux de tâches avec des taux d'utilisation très bas qui ne sont schedulables qu'avec un ordonnanceur préemptif.

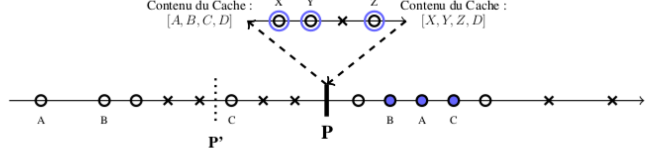
La préemption n'est pas gratuite

- La tâche qui préempte perturbe les états jouant sur la performance, comme les caches et les pipelines.
- Lors de la reprise d'exécution de la tâche prémptée, cette perturbation cause des défauts de cache supplémentaires.
- On appelle le temps supplémentaire dû à ces défaut de cache le *cache-related preemption delay* (CRPD).



Exemple d'effet de la préemption





○ = échecs supplémentaires dus à la préemption (CRPD)

▶ici : $CRPD = 3 * cost(miss)$

Image C. Maiza - livre "Ordonnancement des systèmes temps-réel."

Où prendre en compte le coût de préemption ?

- Intégrer dans l'analyse de WCET: [4]
  - supposer des cache misses partout
  - usage facile dans les analyses d'ordonnançabilité. (mais très pessimiste)
- WCET Analysis + CRPD Analysis: [2]
  - $WCET_{bound} + n \cdot CRPD_{bound} \geq$ temps d'exécution avec "jusqu'à n préemptions".
  - plus précis mais très peu d'analyses d'ordonnançabilité savent prendre en compte.

Analyses de CRPD, quoi faire?

- Tâche préemptée:
  combien de blocs *utiles* dans le cache ?
- Tâche qui préempte:
  quel dommage cause-t-elle sur le cache de *n'importe quelle autre* tâche ?
- Les deux :
  quel dommage la tâche qui préempte cause-t-elle sur le cache utile de la préemptée ?

Conclusion

L'analyse de WCET est un domaine de recherche très foisonnant, en particulier :

- Prise en compte des caractéristiques architecturales plus complexes;
- Le couplage avec l'ordonnancement;
- Les machines parallèles.

Bibliography I

Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord.
Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs.
In *Static Analysis Symposium*, Perpignan, France, September 2010.

Sebastian Altmeyer, Robert I. Davis, and Claire Maiza.
Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems.
Real-Time Systems, 48(5):499–526, 2012.

Julien Henry, Mihail Asavoae, David Monniaux, and Claire Maiza.
How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics.
In Youtao Zhang and Prasad Kulkarni, editors, *LCTES*, pages 43–52. ACM, 2014.

Bibliography II

Joerg Schneider.
Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems.
In *Real-Time Systemes Symposium*, 2000.