Polyhedral Scheduling and Relaxation of Synchronous Reactive Systems

Guillaume Iooss, Albert Cohen, Dumitru Potop-Butucaru, Marc Pouzet, Vincent Bregeon, Jean Souyris, Philippe Beaufreton

INRIA, Google, ENS, Airbus, Safran

June 20th, 2022

Application considered

Critical embedded real-time synchronous applications (ex: avionic)

Application considered

Critical embedded real-time synchronous applications (ex: avionic)

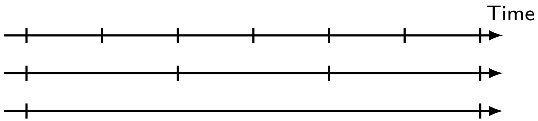
- **Critical system:** correction is (very) important
- **Real-time:** hard temporal deadlines
- **Synchronous:**
  - Manipulate infinite streams of data
  - Computations can have different cadences of production
  - Rhythm of computations synchronized with a global clock

Integration program

- **Integration program:** top-most part of the application. Orchestrates the exchange of data between computations

Integration program

- **Integration program:** top-most part of the application. Orchestrates the exchange of data between computations
- Assumed properties:

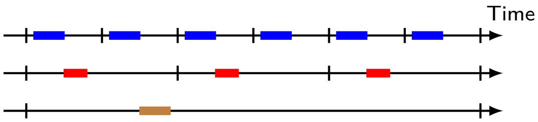


Harmonic periods

Integration program

- **Integration program:** top-most part of the application. Orchestrates the exchange of data between computations

- Assumed properties:

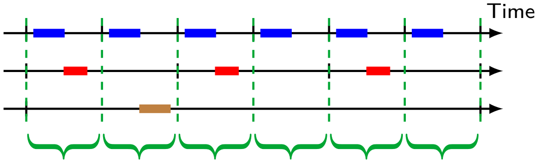


Harmonic periods
Computation: once per period

Integration program

- **Integration program:** top-most part of the application. Orchestrates the exchange of data between computations

Code generation:

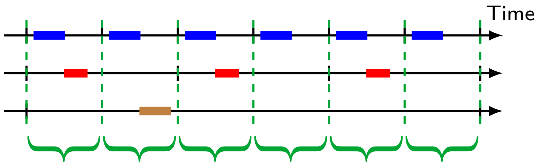


Code generation: infinite while loop (with if statements)
1 iteration = 1 tick of fastest period

Integration program

- **Integration program:** top-most part of the application. Orchestrates the exchange of data between computations

Code generation:



Code generation: infinite while loop (with if statements)
1 iteration = 1 tick of fastest period

- **Phase:** in which iteration should a computation be? Must respect timing constraints
  - Deadline at the end of a tick
  - (Additional) Latency constraints

Contributions

1. How are integration programs written in Lustre ?
   - Specialization: 1-synchronous clocks
   - Issue: phase (= schedule) has to be provided manually.

Contributions

1. How are integration programs written in Lustre ?
    - Specialization: 1-synchronous clocks
    - Issue: phase (= schedule) has to be provided manually.

2. Unknown phases, to be inferred by the compiler:
    - Gather affine constraints from typing system
    - ILP formulation (and tricks) to find them
    - Objective function: load-balancing

Contributions

1. How are integration programs written in Lustre ?
   - Specialization: 1-synchronous clocks
   - Issue: phase (= schedule) has to be provided manually.
2. Unknown phases, to be inferred by the compiler:
   - Gather affine constraints from typing system
   - ILP formulation (and tricks) to find them
   - Objective function: load-balancing
3. Under-specified operators: fuzzy dataflow
   - Possible to relax some dependency constraints

Background - Lustre programming language

- Equational language for synchronous program:
  - Variables/expressions are stream of data
  - Equations define the values of variables

Background - Lustre programming language

- Equational language for synchronous program:
  - Variables/expressions are stream of data
  - Equations define the values of variables

x	0	1	1	2	...
y	4	-2	1	4	...
42	42	42	42	42	...
$x + y$	4	-1	2	6	...
42 `fby` y	42	4	-2	1	...

- `fby` operator: use the previous value (memory)

Background - Clocks

- **Clock**: $x :: clk$
  - Boolean stream to encode if a value is present at a tick
  - Arbitrary in general
  - Will consider only *periodic* clocks (ex: (T F T T))
- Clocking rules: check that the clocks match

Background - Clocks

- **Clock**: $x :: clk$
  - Boolean stream to encode if a value is present at a tick
  - Arbitrary in general
  - Will consider only *periodic* clocks (ex: (T F T T))
- Clocking rules: check that the clocks match
- **Sub/Over-sampling:** `when` / (`merge` + `current`)
  - `when`: filter the values of a (faster) stream
  - `merge`: combine 2 streams
  - `current`: repeat (faster) a value produced by a slower stream

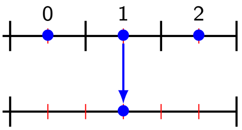
x	$:: c$	0	1	1	2	...
$b = (TF)$	$:: c$	T	F	T	F	...
$z = x$ `when` b	$:: c$ `on` b	0	–	1	–	...
y	$:: c$ `on not` b	–	42	–	64	...
`merge` $b\ z\ y$	$:: c$	0	42	1	64	...
`current(b,0,z)`	$:: c$	0	0	1	1	...

1-synchronous clocks and temporal operators

- **1-synchronous clock** : for integration program
  - Periodic + only one activation per period
  - $(F^k.T.F^{n-k-1})$: $n =$ period, $k =$ phase, $0 \leq k < n$
    Compact representation: $[k, n]$

1-synchronous clocks and temporal operators

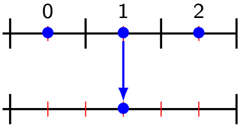
- **1-synchronous clock** : for integration program
  - Periodic + only one activation per period
  - $(F^k.T.F^{n-k-1})$: $n =$ period, $k =$ phase, $0 \leq k < n$
    Compact representation: $[k, n]$
- Temporal operators:
  - Specialization of `when`/`current` operators.
  - `delay(d)` / `delayfby(d)` operators



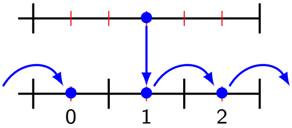
y = x `when` [1,3]

1-synchronous clocks and temporal operators

- **1-synchronous clock** : for integration program
  - Periodic + only one activation per period
  - $(F^k.T.F^{n-k-1})$: $n =$ period, $k =$ phase, $0 \leq k < n$
    Compact representation: $[k, n]$
- Temporal operators:
  - Specialization of `when`/`current` operators.
  - `delay(d)` / `delayfby(d)` operators



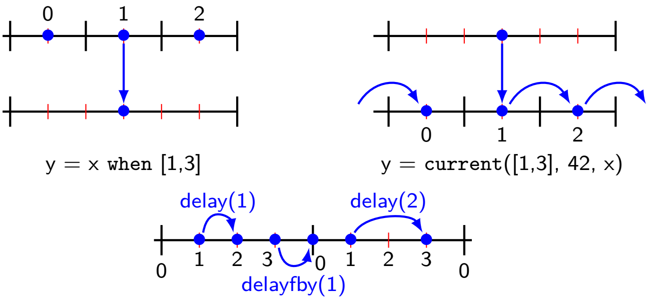
y = x `when` [1,3]



y = `current`([1,3], 42, x)

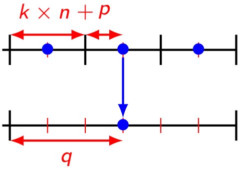
1-synchronous clocks and temporal operators

- **1-synchronous clock** : for integration program
  - Periodic + only one activation per period
  - $(F^k.T.F^{n-k-1})$: n = period, k = phase, $0 \leq k < n$
    Compact representation: $[k, n]$
- Temporal operators:
  - Specialization of `when`/`current` operators.
  - `delay(d)` / `delayfby(d)` operators



Clock analysis

- 1-synchronous operators: add a constant to the phase



y = x `when` [k,r]

where:

- p/q are phases, n/m periods
- k is the number of the sampled occurrence

Extension - Unknown phases

- 1-synchronous clocks with unknown phases.
  - Period is still specified
  - Compiler will need to find a set of valid phases

Extension - Unknown phases

- 1-synchronous clocks with unknown phases.
  - Period is still specified
  - Compiler will need to find a set of valid phases
- New operators: `buffer`/`bufferfby`:
  - Similar to `delay(d)` / `delayfby(d)` with less information
  - `buffer` clock rule: dependence constraint

Extension - Unknown phases

- 1-synchronous clocks with unknown phases.
  - Period is still specified
  - Compiler will need to find a set of valid phases
- New operators: `buffer`/`bufferfby`:
  - Similar to `delay(d)` / `delayfby(d)` with less information
  - `buffer` clock rule: dependence constraint
- **Clock analysis:** now manipulates symbolic phases p.

Extension - Unknown phases

- 1-synchronous clocks with unknown phases.
  - Period is still specified
  - Compiler will need to find a set of valid phases
- New operators: `buffer`/`bufferfby`:
  - Similar to `delay(d)` / `delayfby(d)` with less information
  - `buffer` clock rule: dependence constraint
- **Clock analysis:** now manipulates symbolic phases p.
- Gather affine constraints from it:
  - Boundary condition on phases: $0 \leq p < n$
  - Dependence constraints from buffer: $p_1 + C \leq p_2$
  - Unification constraints from ops/equation: $p_1 = p_2 + C$
  - (Additional) latency constraints: $p_1 - p_2 \leq C$

Example

Example of program:

```
model main(...)  returns (...)
var
  a ::  [..,1]; b ::  [..,2];
  c ::  [..,6]; d ::  [..,2]; e ::  [..,1];
let
  b = buffer f1(a when [1,2]);
  c = buffer f2(b when [0,3]);
  d = f3(current([2,3], 0.0, buffer c));
  e = f4(current([0,2], 0.0, buffer d));
tel
```

Example

Example of program:

```
model main(...)  returns (...)
var
  a ::  [..,1]; b ::  [..,2];
  c ::  [..,6]; d ::  [..,2]; e ::  [..,1];
let
  b = buffer f1(a when [1,2]);
  c = buffer f2(b when [0,3]);
  d = f3(current([2,3], 0.0, buffer c));
  e = f4(current([0,2], 0.0, buffer d));
tel
```

Extracted constraints:

- Bounds from variable declarations:
  $0 \leq p_a, p_e < 1$, $0 \leq p_b, p_d < 2$ and $0 \leq p_c < 6$

Example

Example of program:

```
model main(...)  returns (...)
var
  a ::  [..,1]; b ::  [..,2];
  c ::  [..,6]; d ::  [..,2]; e ::  [..,1];
let
  b = buffer f1(a when [1,2]);
  c = buffer f2(b when [0,3]);
  d = f3(current([2,3], 0.0, buffer c));
  e = f4(current([0,2], 0.0, buffer d));
tel
```

Extracted constraints:

- Bounds from variable declarations:
  $0 \leq p_a, p_e < 1$, $0 \leq p_b, p_d < 2$ and $0 \leq p_c < 6$
- Constraints from `buffer`:
  $p_a + 1 \leq p_b$

Example

Example of program:

```
model main(...)  returns (...)
var
  a ::  [..,1]; b ::  [..,2];
  c ::  [..,6]; d ::  [..,2]; e ::  [..,1];
let
  b = buffer f1(a when [1,2]);
  c = buffer f2(b when [0,3]);
  d = f3(current([2,3], 0.0, buffer c));
  e = f4(current([0,2], 0.0, buffer d));
tel
```

Extracted constraints:

- Bounds from variable declarations:
  $0 \leq p_a, p_e < 1$, $0 \leq p_b, p_d < 2$ and $0 \leq p_c < 6$
- Constraints from `buffer`:
  $p_a + 1 \leq p_b \qquad p_b \leq p_c$

Example

Example of program:

```
model main(...)  returns (...)
var
  a ::  [..,1]; b ::  [..,2];
  c ::  [..,6]; d ::  [..,2]; e ::  [..,1];
let
  b = buffer f1(a when [1,2]);
  c = buffer f2(b when [0,3]);
  d = f3(current([2,3], 0.0, buffer c));
  e = f4(current([0,2], 0.0, buffer d));
tel
```

Extracted constraints:

- Bounds from variable declarations:
  $0 \leq p_a, p_e < 1$, $0 \leq p_b, p_d < 2$ and $0 \leq p_c < 6$
- Constraints from `buffer`:
  $p_a + 1 \leq p_b \qquad p_b \leq p_c \qquad p_c - 4 \leq p_d \qquad p_d \leq p_e$

Underspecified operators - Motivation

- `when`/`current` operator: which occurrence is sampled?
  - Critical to have a deterministic dataflow

Underspecified operators - Motivation

- `when`/`current` operator: which occurrence is sampled?
    - Critical to have a deterministic dataflow
- But, engineers do not know that information. . .

Underspecified operators - Motivation

- `when`/`current` operator: which occurrence is sampled?
  - Critical to have a deterministic dataflow
- But, engineers do not know that information. . .
- . . . How is that possible?
  - Equations coming from a physical model
  - Some stream have a physical meaning, with low temporal variability (ex: temperature)

  $\Rightarrow$ Just need a "fresh enough" value ($\neq$ the most recent one)

Underspecified operators - Motivation

- `when`/`current` operator: which occurrence is sampled?
  - Critical to have a deterministic dataflow
- But, engineers do not know that information...
- ...How is that possible?
  - Equations coming from a physical model
  - Some stream have a physical meaning, with low temporal variability (ex: temperature)

  $\Rightarrow$ Just need a "fresh enough" value ($\neq$ the most recent one)

$\Rightarrow$ Provide this property to relax the scheduling problem

Underspecified operators - Operators

- Underspecified operators:
  - `(e when? r)` $=$ any `(e when [k,r])`, $0 \leq k < r$
  - `current?(r,i,e)` $=$ any `current([k,r],i,e)`, $0 \leq k < r$

Underspecified operators - Operators

- Underspecified operators:
  - `(e when? r)` $=$ any `(e when [k,r])`, $0 \leq k < r$
  - `current?(r,i,e)` $=$ any `current([k,r],i,e)`, $0 \leq k < r$
  - `(c fby? e)` $=$ `e` or `(c fby e)`
  - `(c bufferfby? e)` $=$ `(buffer e)` or `(c bufferfby e)`

Underspecified operators - Operators

- Underspecified operators:
  - `(e when? r)` = any `(e when [k,r])`, $0 \leq k < r$
  - `current?(r,i,e)` = any `current([k,r],i,e)`, $0 \leq k < r$
  - `(c fby? e)` = `e` or `(c fby e)`
  - `(c bufferfby? e)` = `(buffer e)` or `(c bufferfby e)`
- **Clock constraint:** add *decision variables* d_i
  Corresponds to the choice of dataflow made

Underspecified operators - Operators

- Underspecified operators:
  - `(e when? r)` = any `(e when [k,r])`, $0 \leq k < r$
  - `current?(r,i,e)` = any `current([k,r],i,e)`, $0 \leq k < r$
  - `(c fby? e)` = `e` or `(c fby e)`
  - `(c bufferfby? e)` = `(buffer e)` or `(c bufferfby e)`
- **Clock constraint:** add *decision variables* d_i
  Corresponds to the choice of dataflow made
- Add constraints on the d_i to avoid causality loops.

ILP solving

- Cost function: load balancing across all phases
  - Each computation T has a weight (WCET) W_T

ILP solving

- Cost function: load balancing across all phases
  - Each computation T has a weight (WCET) W_T
  - Binary representation of p_T: $\delta_{k,T}$ ($= 1$ iff $p_T = k$, else $= 0$)
- Link binary - integer representation:

$$
\begin{aligned}
(\forall T \text{ computation}) \quad & \sum_k \delta_{k,T} = 1 \\
(\forall T \text{ computation}) \quad & \sum_k k.\delta_{k,T} = p_T
\end{aligned}
$$

ILP solving

- Cost function: load balancing across all phases
  - Each computation T has a weight (WCET) W_T
  - Binary representation of p_T: $\delta_{k,T}$ ($= 1$ iff $p_T = k$, else $= 0$)
- Link binary - integer representation:

$$\begin{array}{ll}(\forall T \text{ computation}) & \sum_k \delta_{k,T} = 1 \\ (\forall T \text{ computation}) & \sum_k k.\delta_{k,T} = p_T\end{array}$$

- Load balancing constraint (minimize W_{max})

$$(\forall k \text{ phase}) \quad \sum_T (\delta_{k \ mod \ per(T),T} \times W_T) \leq W_{max}$$

ILP solving

- Cost function: load balancing across all phases
  - Each computation T has a weight (WCET) W_T
  - Binary representation of p_T: $\delta_{k,T}$ ($= 1$ iff $p_T = k$, else $= 0$)
- Link binary - integer representation:

$$(\forall T \text{ computation}) \quad \sum_k \delta_{k,T} = 1$$
$$(\forall T \text{ computation}) \quad \sum_k k.\delta_{k,T} = p_T$$

- Load balancing constraint (minimize W_{max})

$$(\forall k \text{ phase}) \quad \sum_T (\delta_{k \ mod \ per(T),T} \times W_T) \leq W_{max}$$

- Redundancy of encoding of phase value (binary + integer)

  $\Rightarrow$ Alternative formulation with binary only

Experimental results

- Two real-world industrial use cases:
  - **UC1:** (control command - Airbus) 5124 nodes, 32k variables 4 harmonic periods (10/20/40/120 ms)
  - **UC2:** (motor regulation - Safran) 142 nodes, 5 harmonic periods (15/30/60/120/240 ms)

Experimental results

- Two real-world industrial use cases:
  - **UC1:** (control command - Airbus) 5124 nodes, 32k variables 4 harmonic periods (10/20/40/120 ms)
  - **UC2:** (motor regulation - Safran) 142 nodes, 5 harmonic periods (15/30/60/120/240 ms)

ILP problem		# of lines	# of vars	Solving time
UC1	No cost function	28742	5124	0.05 s
	Load balancing integral	79128	45249	17.75 s
	Load balancing binary	200746	40125	9.05 s
UC2	No cost function	3908	143	0.01 s
	Load balancing integral	5106	1039	0.03 s
	Load balancing binary	17689	897	0.05 s

- Solutions found are $\leq 0.01\%$ above rational optimal.

Injecting the solution in the program

- Once we found a solution, reinject it in the program
- With decision variable values: `op?` $\Rightarrow$ `op`
- With phase values:
  - `buffer` $\Rightarrow$ `delay(d)`
  - `[..,n]` $\Rightarrow$ `[k,n]`

$\Rightarrow$ After substitution, fully specified 1-synchronous program

Conclusion

- In summary: Three language extensions to Lustre:
  - 1-synchronous clocks...
  - ...With unknown phases
  - Underspecified operators.

Conclusion

- In summary: Three language extensions to Lustre:
  - 1-synchronous clocks...
  - ...With unknown phases
  - Underspecified operators.

- Remain to be done: load balancing with parallelism?

Conclusion

- In summary: Three language extensions to Lustre:
  - 1-synchronous clocks. . .
  - . . . With unknown phases
  - Underspecified operators.

- Remain to be done: load balancing with parallelism?

- Takeaway message: Semantic properties can be useful

  $\Rightarrow$ Check how an application is specified/used in practice

Thank you for listening. . .

. . . Do you have any questions?