Scheduling and compiling rate-synchronous programs with end-to-end latency constraints

Timothy Bourke

Inria Paris — PARKAS Team
École normale supérieure, PSL University

21 October 2025, Systèmes réactifs synchrones MPRI 2-23-1

Plan

Context

- Standard practice: design an application as a set of periodically executed tasks that communicate through shared variables.
- Read data from sensors via a bus, compute through sequences of cyclic tasks, and write to actuators via the bus.

Context

- Standard practice: design an application as a set of periodically executed tasks that communicate through shared variables.
- Read data from sensors via a bus, compute through sequences of cyclic tasks, and write to actuators via the bus.

Airbus project "All-in-Lustre"

- *Current system*: task = Lustre node ($\approx 5\,000$), separate constraints on order and latency.
- *Desired system*: "All-in-Lustre": compose nodes into a single Lustre program with new features for specifying periods and execution constraints.
- Generate sequential code for cyclic execution on a single-processor platform.
- Base period = 5ms. Tasks at 10ms, 20ms, 40ms, and 120ms.
- Tasks are already chopped up into small pieces.

Slow flows

```
node read() returns (y:int);
node write(x:int) returns ();
node filter(x:int) returns (y:int);

node main() returns ()
var s0, s1, s2, s3 : int :: 1/3;
let
    s0 = read();
    s1 = filter(s0);
    s2 = filter(s1);
    s3 = s1 + s2;
    () = write(s3);
tel
```

- Declare variables of rate 1/3 (period = 3)
- Calculate each one once every three cycles

```
$ presseail example1.ail -v --glpk --print
```

Slow flows

```
node read() returns (y:int);
node write(x:int) returns ();
node filter(x:int) returns (y:int);

node main() returns ()
var s0, s1, s2, s3 : int :: 1/3;
let
    s0 = read();
    s1 = filter(s0);
    s2 = filter(s1);
    s3 = s1 + s2;
    () = write(s3);
tel
```

- Declare variables of rate 1/3 (period = 3)
- Calculate each one once every three cycles

```
$ presseail example1.ail --glpk --compile 1 --print
```

Slow flows

```
node read() returns (y:int);
node write(x:int) returns ();
node filter(x:int) returns (y:int);

node main() returns ()
var s0, s1, s2, s3 : int :: 1/3;
let
    s0 = read();
    s1 = filter(s0);
    s2 = filter(s1);
    s3 = s1 + s2;
    () = write(s3);
tel
```

- Declare variables of rate 1/3 (period = 3)
- Calculate each one once every three cycles
- The 5 calculations here are synchronous relative to the period
  even if they are not necessarily simultaneous relative to the base clock
- `s3 = s1 + s2` is well clocked
  since `s1 :: 1/3`, `s2 :: 1/3`, and `s3 :: 1/3`.

```
$ presseail example1.ail --glpk --compile 1 --print
```

Slow flows

```
node read() returns (y:int);
node write(x:int) returns ();
node filter(x:int) returns (y:int);

node main() returns ()
var s0, s1, s2, s3 : int :: 1/3;
let
    s0 = read();
    s1 = filter(s0);
    s2 = filter(s1);
    s3 = s1 + s2;
    () = write(s3);
tel
```

- Declare variables of rate 1/3 (period = 3)
- Calculate each one once every three cycles
- The 5 calculations here are synchronous relative to the period
  even if they are not necessarily simultaneous relative to the base clock
- `s3 = s1 + s2` is well clocked
  since `s1 :: 1/3`, `s2 :: 1/3`, and `s3 :: 1/3`.
- Causality applies 'across' a period
  and 'within' an instant: $s_0 \to s_1 \to s_2 \to s_3 \to ()$

```
$ presseail example1.ail --glpk --compile 1 --print
```

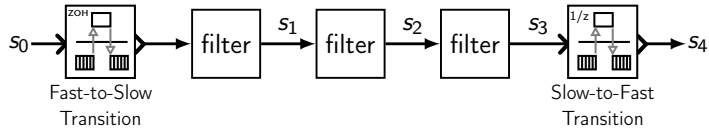
Changing speeds: 1

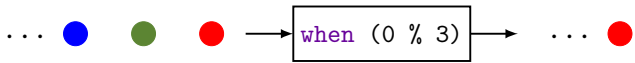
```
resource cpu : int

node filter(x : int)
  returns  (y : int);

node main(s0 : int)
returns (s4 : int)
var s1, s2 : int :: 1/3;
    s3 : int :: 1/3 last = 0;
let
    s1 = filter(s0 when (0 % 3));
    s2 = filter(s1);
    s3 = filter(s2);
    s4 = current(s3, (2 % 3));
tel
```

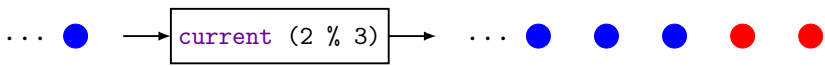
- `x when c`
  - » c is for '(sampling) choice'
  - » sub-sampling of a stream
  - » fast-to-slow rate change
- `current(x, c)`
  - » stutter stream elements
  - » must declare an initial `last` value
  - » slow-to-fast rate change

```
y = merge c x ((0 fby y) when not c)
```

```
$ presseail example2.ail --glpk --compile 1
```


s_0
ZOH
Fast-to-Slow
Transition
filter
s_1
filter
s_2
filter
s_3
1/z
Slow-to-Fast
Transition
s_4


when (0 % 3)


current (2 % 3)

Changing speeds: 2

```
r = w when (i % n)
```

- `(i % n)`: take the `i`th of every `n` elements.
- `n` is the rate of `w` relative to `r`
  E.g., for `w :: 1/4` and `r :: 1/8`, `n` is 2.
- It can be deduced from the clocks, but is useful for readability.
- It implies a lower bound on the scheduling of the equation.

Changing speeds: 2

```
r = w when (i % n)
```

- `(i % n)`: take the `i`th of every `n` elements.
- `n` is the rate of `w` relative to `r`
  E.g., for `w :: 1/4` and `r :: 1/8`, `n` is 2.
- It can be deduced from the clocks, but is useful for readability.
- It implies a lower bound on the scheduling of the equation.

```
r = current(w, (i % n))
```

- `(i % n)`: repeat the initial `last` value `i` times, then repeat each `w` value `n` times.
- `n` is the rate of `r` relative to `w`
  E.g., for `r :: 1/4` and `w :: 1/8`, `n` is 2.
- It implies an upper bound on the scheduling of the equation.

Changing speeds: 2

`r = w when (i % n)`

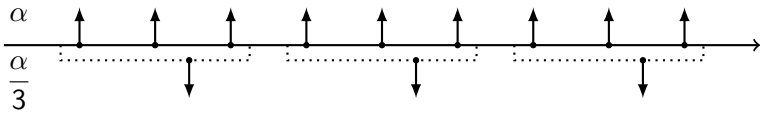
- `(i % n)`: take the `i`th of every `n` elements.
- `n` is the rate of `w` relative to `r`
  E.g., for `w :: 1/4` and `r :: 1/8`, `n` is 2.
- It can be deduced from the clocks, but is useful for readability.
- It implies a lower bound on the scheduling of the equation.

`r = current(w, (i % n))`

- `(i % n)`: repeat the initial `last` value `i` times, then repeat each `w` value `n` times.
- `n` is the rate of `r` relative to `w`
  E.g., for `r :: 1/4` and `w :: 1/8`, `n` is 2.
- It implies an upper bound on the scheduling of the equation.

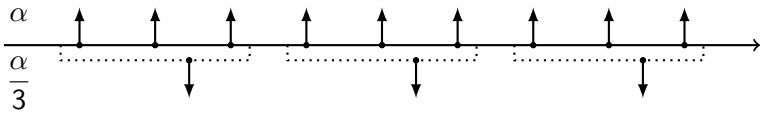
- Write `(? % n)` if we don't care when values are sampled/updated.
- The schedule decides when sampling/updating occur; fixed at compile time.

Rate-synchronous Model

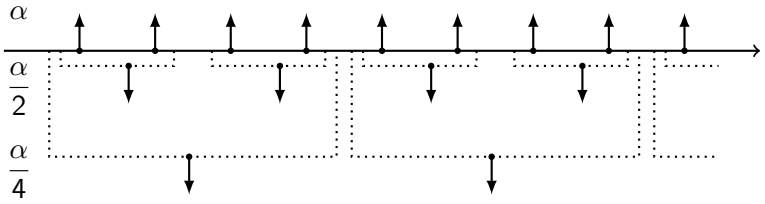


One slow tick is synchronous with three fast ones.

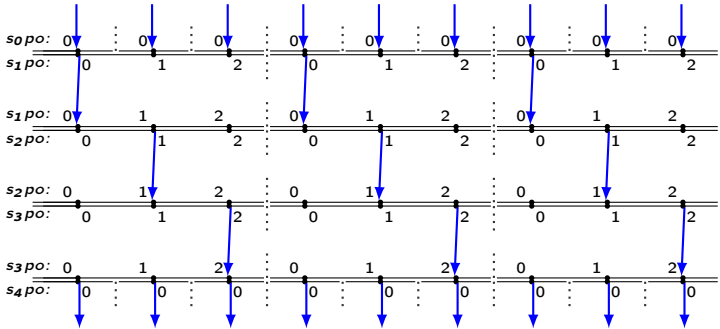
Rate-synchronous Model



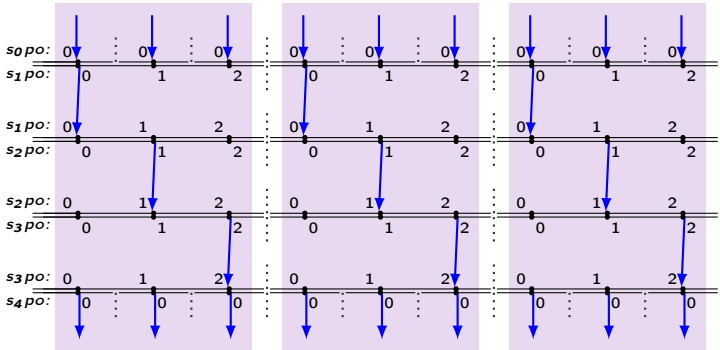
One slow tick is synchronous with three fast ones.



- Calculations are synchronous relative to their periods
  but not necessarily simultaneous relative to execution cycles
- The compiler assigns computations to phases, buffering values if necessary

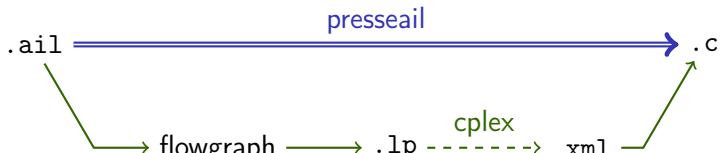



```
node main (s0 : int) returns (s4 : int)
var s1, s2 : int :: 1 / 3,
    s3 : int :: 1 / 3 last = 0;
let
  label(filter)   phase(0 % 3) s1 = filter(s0 when (0 % 3));
  label(filter_0) phase(1 % 3) s2 = filter(s1);
  label(filter_1) phase(2 % 3) s3 = filter(s2);
  label(s4_2)     phase(0 % 1) s4 = current(s3, (2 % 3));
tel
```



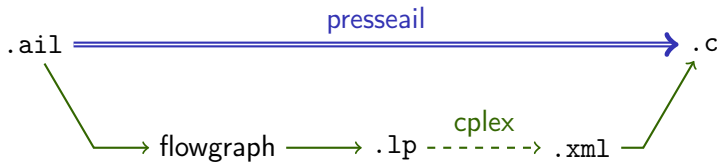

```
node main (s0 : int) returns (s4 : int)
var s1, s2 : int :: 1 / 3,
    s3 : int :: 1 / 3 last = 0;
let
  label(filter)   phase(0 % 3) s1 = filter(s0 when (0 % 3));
  label(filter_0) phase(1 % 3) s2 = filter(s1);
  label(filter_1) phase(2 % 3) s3 = filter(s2);
  label(s4_2)     phase(0 % 1) s4 = current(s3, (2 % 3));
tel
```

Overview: compilation using Integer Linear Programming (ILP)



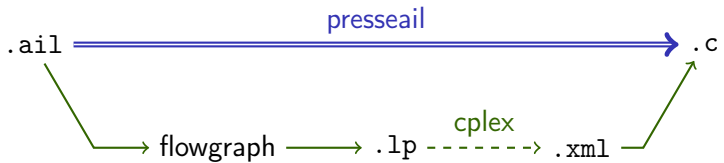
Overview: compilation using Integer Linear Programming (ILP)

- Like Prelude [Forget, Boniol, Lesens, and Pagetti (2010): A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems]
  But, no WCET, no deadlines, no real-time tasks
- Rates expressed as $1/n$ of the base clock



Overview: compilation using Integer Linear Programming (ILP)

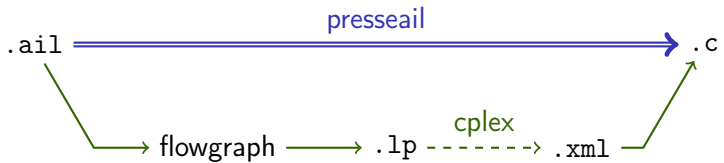
- Like Prelude [Forget, Boniol, Lesens, and Pagetti (2010): A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems]
  But, no WCET, no deadlines, no real-time tasks
- Rates expressed as $1/n$ of the base clock
- One or more `step` functions
- Called cyclically at the base rate



Overview: compilation using Integer Linear Programming (ILP)

- Like Prelude [Forget, Boniol, Lesens, and Pagetti (2010): A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems]
  But, no WCET, no deadlines, no real-time tasks
- Rates expressed as $1/n$ of the base clock

- One or more `step` functions
- Called cyclically at the base rate

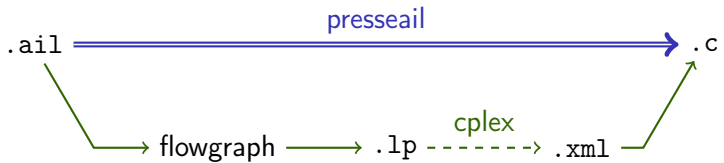


- Vertex = equation
- Arc from producer to consumer
- Independent of source language

Overview: compilation using Integer Linear Programming (ILP)

- Like Prelude [Forget, Boniol, Lesens, and Pagetti (2010): A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems]
  But, no WCET, no deadlines, no real-time tasks
- Rates expressed as $1/n$ of the base clock

- One or more `step` functions
- Called cyclically at the base rate



- Vertex = equation
- Arc from producer to consumer
- Independent of source language

- Data dependencies
- Load balancing
- End-to-end latency

Aside: `fby` or `last`

```
x = c fby e;
P
```

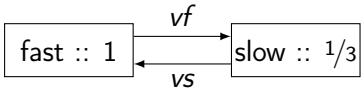
```
var nx : T last = c

nx = e;
P{last nx/x}
```

- `c fby e` initialized unit delay / register / `delay c e`
- `last x` previous value of initialized variable

  [Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]

- Here: easier to work with `last x`

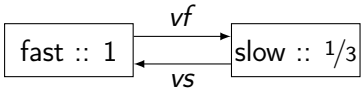
Rate-synchronous Model




```
node eg1() returns ()
var vf : int :: 1;
    vs : int :: 1/3 last = 0;
     n : int :: 1 last = 0;
let
  n = (last n) + 1;
  vf = n + current(vs, (2 % 3));
  vs = (vf when (1 % 3)) + 5;
tel
```

vf	1	2	10	11	12	23	24	25	39	···
n	1	2	3	4	5	6	7	8	9	···
vs	7			17			30			···

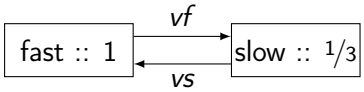
Rate-synchronous Model




```
node eg1() returns ()
var vf : int :: 1;
    vs : int :: 1/3 last = 0;
     n : int :: 1 last = 0;
let
  n = (last n) + 1;
  vf = n + current(vs, (2 % 3));
  vs = (vf when (1 % 3)) + 5;
tel
```

vf	1	2	10	11	12	23	24	25	39	···
n	1	2	3	4	5	6	7	8	9	···
vs	7			17			30			···

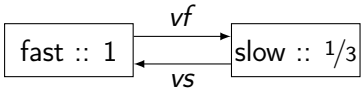
Rate-synchronous Model




```
node eg1() returns ()
var vf : int :: 1;
    vs : int :: 1/3 last = 0;
     n : int :: 1 last = 0;
let
  n = (last n) + 1;
  vf = n + current(vs, (2 % 3));
  vs = (vf when (1 % 3)) + 5;
tel
```

vf	1	2	10	11	12	23	24	25	39	···
n	1	2	3	4	5	6	7	8	9	···
vs		7			17			30		···

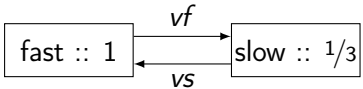
Rate-synchronous Model




```
node eg1() returns ()
var vf : int :: 1;
    vs : int :: 1/3 last = 0;
     n : int :: 1 last = 0;
let
  n = (last n) + 1;
  vf = n + current(vs, (2 % 3));
  vs = (vf when (1 % 3)) + 5;
tel
```

vf	1	2	10	11	12	23	24	25	39	···
n	1	2	3	4	5	6	7	8	9	···
vs		7			17			30		···

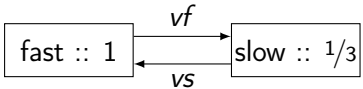
Rate-synchronous Model




```
node eg1() returns ()
var vf : int :: 1;
    vs : int :: 1/3 last = 0;
     n : int :: 1 last = 0;
let
  n = (last n) + 1;
  vf = n + current(vs, (2 % 3));
  vs = (vf when (1 % 3)) + 5;
tel
```

vf	1	2	10	11	12	23	24	25	39	···
n	1	2	3	4	5	6	7	8	9	···
vs		7			17			30		···

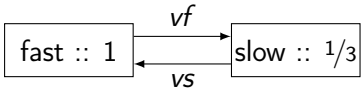
Rate-synchronous Model




```
node eg1() returns ()
var vf : int :: 1;
    vs : int :: 1/3 last = 0;
     n : int :: 1 last = 0;
let
  n = (last n) + 1;
  vf = n + current(vs, (2 % 3));
  vs = (vf when (1 % 3)) + 5;
tel
```

vf	1	2	10	11	12	23	24	25	39	···
n	1	2	3	4	5	6	7	8	9	···
vs		7			17			30		···

Rate-synchronous Model




```
node eg1() returns ()
var vf : int :: 1;
    vs : int :: 1/3 last = 0;
     n : int :: 1 last = 0;
let
  n = (last n) + 1;
  vf = n + current(vs, (2 % 3));
  vs = (vf when (1 % 3)) + 5;
tel
```

vf	1	2	10	11	12	23	24	25	39	···
n	1	2	3	4	5	6	7	8	9	···
vs		7			17			30		···

Syntax

$$
\begin{aligned}
eq &::= x \texttt{ = } e \mid x^* \texttt{ = } f\texttt{(}e^*\texttt{)} \\
e &::= c \mid x \mid \diamond\ e \mid e \oplus e \mid \texttt{if } e \texttt{ then } e \texttt{ else } e \mid \texttt{last } x \\
&\quad \mid x \texttt{ when } s \mid \texttt{(last } x\texttt{) when } s \mid \texttt{current(}x\texttt{, } s\texttt{)} \\
s &::= \texttt{(}c \texttt{ \% } c\texttt{)} \mid \texttt{(? \% } c\texttt{)} \\
p &::= (d\ \texttt{;})^* \\
d &::= \texttt{resource } x \texttt{ : } ty \\
&\quad \mid \texttt{node } f\texttt{(}\ (x \texttt{ : } ty\ \texttt{;})^*\texttt{)} \texttt{ returns (}\ (x \texttt{ : } ty\ \texttt{;})^*\texttt{)} \texttt{ requires (}\ (x \texttt{ = } c\ \texttt{;})^*\texttt{)} \\
&\quad \mid \texttt{node } f\texttt{(}\ (x \texttt{ : } ty \texttt{ :: } ck\ [\texttt{last = } c]\ \texttt{;})^*\texttt{)} \texttt{ returns (}\ (x \texttt{ : } ty \texttt{ :: } ck\ [\texttt{last = } c]\ \texttt{;})^*\texttt{)} \\
&\quad\ \ \texttt{var}\ (x \texttt{ : } ty \texttt{ :: } ck\ [\texttt{last = } c]\ \texttt{;})^* \texttt{ let}\ (((pragmas\ eq) \mid cst)\ \texttt{;})^+ \texttt{ tel} \\
pragmas &::= [\texttt{label(}x\texttt{)}]\ [\texttt{phase(}c \texttt{ \% } c\texttt{)}] \\
cst &::= \texttt{resource balance } x \\
&\quad \mid \texttt{resource } x\ rel\ c \\
&\quad \mid \texttt{latency (exists} \mid \texttt{forward} \mid \texttt{backward)}\ rel\ c\ \texttt{(}\ x\ \texttt{,}\ x\ (\texttt{,}\ x)^*\texttt{)} \\
rel &::= \texttt{<=} \mid \texttt{<} \mid \texttt{=} \mid \texttt{>} \mid \texttt{>=}
\end{aligned}
$$

Valid programs are defined by clock typing

$$\frac{e_1 :: 1/n \qquad e_2 :: 1/n}{e_1 \oplus e_2 :: 1/n}$$

$$\frac{x :: 1/n}{\texttt{last}\ x :: 1/n}$$

$$\frac{x :: 1/m}{x\ \texttt{when}\ (\cdot \%\, n) :: 1/mn}$$

$$\frac{x :: 1/mn}{\texttt{current}(x, (\cdot \%\, n)) :: 1/m}$$

- No phase offsets in clock types, unlike
  - » Prelude: `rate(100, 0)`
    [Forget, Boniol, Lesens, and Pagetti (2010): A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems]
  - » Lucy-n: `(010), 00(00100)`
    [Cohen, Duranton, Eisenbeis, Pagetti, Plateau, and Pouzet (2006): N-Synchronous Kahn networks: a relaxed model of synchrony for real-time systems]
  - » 1-Synchronous: `[0, 2]`
    [Iooss, Pouzet, Cohen, Potop-Butucaru, Souyris, Bregeon, and Baufreton (2020): 1-Synchronous Programming of Large Scale, Multi-Periodic Real-Time Applications with Functional Degrees of Freedom]
  - » Simulink: `[Ts, To]`
- Dataflow semantics: independent of phase offsets
- Generated code: phase offsets implement data dependencies.

Valid programs are defined by clock typing

$$\frac{e_1 :: 1/n \qquad e_2 :: 1/n}{e_1 \oplus e_2 :: 1/n}$$

$$\frac{x :: 1/n}{\texttt{last}\ x :: 1/n}$$

$$\frac{x :: 1/m}{x\ \texttt{when}\ (\cdot \% n) :: 1/mn}$$

$$\frac{x :: 1/mn}{\texttt{current}(x, (\cdot \% n)) :: 1/m}$$

- No phase offsets in clock types, unlike
  - » Prelude: `rate(100, 0)`
    [Forget, Boniol, Lesens, and Pagetti (2010): A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems]
  - » Lucy-n: `(010), 00(00100)`
    [Cohen, Duranton, Eisenbeis, Pagetti, Plateau, and Pouzet (2006): N-Synchronous Kahn networks: a relaxed model of synchrony for real-time systems]
  - » 1-Synchronous: `[0, 2]`
    [Iooss, Pouzet, Cohen, Potop-Butucaru, Souyris, Bregeon, and Baufreton (2020): 1-Synchronous Programming of Large Scale, Multi-Periodic Real-Time Applications with Functional Degrees of Freedom]
  - » Simulink: `[Ts, To]`
- Dataflow semantics: independent of phase offsets
- Generated code: phase offsets implement data dependencies.

Related Work: Lucy-n

- Model and language [Cohen, Duranton, Eisenbeis, Pagetti, Plateau, and Pouzet (2006): N-Synchronous Kahn networks: a relaxed model of synchrony for real-time systems] [Mandel, Plateau, and Pouzet (2010): Lucy-n: a n-Synchronous extension of Lustre]
- Flexible scheduling patterns (0010(010)) and buffering
- Notion of jitter with clock envelopes [Cohen, Mandel, Plateau, and Pouzet (2008): Abstraction of Clocks in Synchronous Data-flow Systems]
- Sophisticated type-based analysis for causality and buffer sizes
- Less focus on code generation

Our work

- Less flexible scheduling
- Buffering is implicit and very limited
- Less clock typing, more causality
- Generate imperative code

Related Work: Iooss et al.

- "1-synchronous" programs [Iooss, Pouzet, Cohen, Potop-Butucaru, Souyris, Bregeon, and Baufreton (2020): 1-Synchronous Programming of Large Scale, Multi-Periodic Real-Time Applications with Functional Degrees of Freedom]
- Two-element clocks: $[phase, period]$
  $(0^k 10^{n-k-1}$ or $0^k(10^{n-1})$, where n is the period and $0 \leq k < n$ is the phase
- Related to work on affine clocks
  - » [Curic (2005): Implementing Lustre Programs on Distributed Platforms with Real-Time Constraints]
  - » [Smarandache, Gautier, and Le Guernic (1999): Validation of Mixed Signal-Alpha Real-Time Systems through Affine Calculus on Clock Synchronisation Constraints]
- Several operators: `when`, `current`, `delay`, `delayfby`, `buffer`, `bufferfby`
- Prototype in Heptagon: introduces (lots of) `whens` and `merges`

Our work

- Simpler clocks, fewer operators, implicit buffering
- Generate imperative code directly

Prelude: Multi-periodic Sync. Prog. [Forget, Boniol, Lesens, and Pagetti (2008): A Multi-Periodic Synchronous Data-Flow Language]

- Language [Forget, Boniol, Lesens, and Pagetti (2010): A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems] and compiler [Pagetti, Forget, Boniol, Cordovilla, and Lesens (2011): Multi-task implementation of multi-periodic synchronous programs]
- Extend Lustre with task periods/phases and WCET.
- Compose real-time primitives to express communication patterns.
- Generate and schedule a set of real-time tasks
  - » WCET, release times, deadlines
  - » Adapt existing scheduling algorithms to respect data dependencies

- "Don't Care" [Wyss, Boniol, Forget, and Pagetti (2012): A Synchronous Language with Partial Delay Specification for Real-Time Systems Programming],
  Let the compiler decide if `c dc x` (`c fby? x`) is
  - » `c fby x`
  - » `x`

Stream-based Semantics

$$[\![e_1 \oplus e_2]\!](i) = [\![e_1]\!](i) \oplus [\![e_2]\!](i)$$

$$[\![\texttt{last}\ x]\!](i) = \begin{cases} x_{\text{-}1} & \text{if } i = 0 \\ [\![x]\!](i-1) & \text{otherwise} \end{cases}$$

$$[\![x\ \texttt{when}\ (s\,\%\,n)]\!](i) = [\![x]\!](n \cdot i + s)$$

$$[\![\texttt{current}(x, (s\,\%\,n))]\!](i) = \begin{cases} x_{\text{-}1} & \text{if } i < s \\ [\![x]\!](\lfloor {}^{i-s}/_{n} \rfloor) & \text{otherwise} \end{cases}$$

- Recursive equations on streams: $\mathbb{N} \to \mathbb{V}$
- $x_{\text{-}1}$ is the initial `last` value
- No explicit presence or absence
- Cf. Prelude (tagged-signal model)

Declare and constrain resources

```
resource cpu : int

node read() returns (y:int);
node write(x:int) returns ();
node filter(x:int) returns (y:int)
  requires (cpu = 4);

node main() returns ()
var s0, s1, s2, s3 : int :: 1/3;
let
    resource cpu <= 4;
    s0 = read();
    s1 = filter(s0);
    s2 = filter(s1);
    s3 = filter(s2);
    () = write(s3);
tel
```

- Declare *named weights* to represent resources required per cycle
  - » Simple proxies for worst-case execution time
  - » Network bus accesses
- Use to constrain scheduling
- normally: `resource balance cpu`

Declare and constrain resources

```
resource cpu : int

node read() returns (y:int);
node write(x:int) returns ();
node filter(x:int) returns (y:int)
  requires (cpu = 4);

node main() returns ()
var s0, s1, s2, s3 : int :: 1/3;
let
    resource cpu <= 4;
    s0 = read();
    s1 = filter(s0);
    s2 = filter(s1);
    s3 = filter(s2);
    () = write(s3);
tel
```

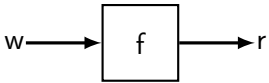
- Declare *named weights* to represent resources required per cycle
  - » Simple proxies for worst-case execution time
  - » Network bus accesses
- Use to constrain scheduling
- normally: `resource balance cpu`

Also: trade-off *resource balancing* against *latency*:

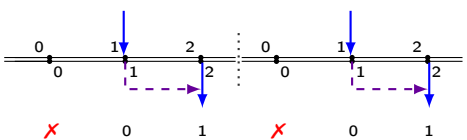
```
latency_chain forward <= 0 (s0 -> s1 -> s2 -> s3);
```

Direct Communications

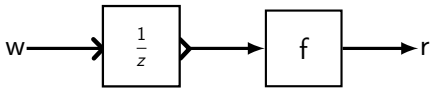
```
r = f(w)
```



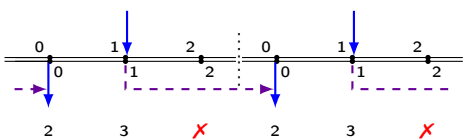
- D_f^w: Direct Write-before-read
  (forward concomitance)
- Dependency constraint: p:$w \leq$ p:r
- $0 \leq$ p:$r -$ p:w




```
r = f(last w)
```



- D_b^r: Direct Read-before-write
  (backward concomitance)
- Dependency constraint: p:$r \leq$ p:w
- $0 \leq$ p:$w -$ p:r

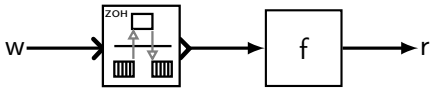


Rate Transitions

`r = f(w when (1 % 3))`

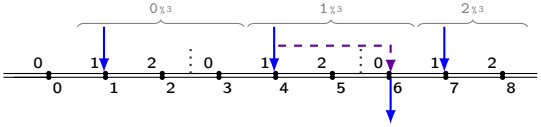
`(i % n)`: take value `i` of every `n`

`(? % n)`: take any of every `n` values



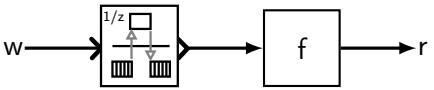
Fast-to-Slow Transition

- $/_{nf}$: Fast-to-slow
  (forward concomitance)



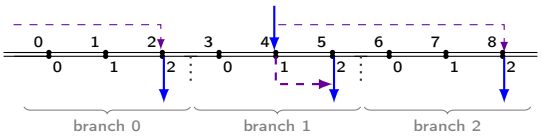
`r = f(current(w, (1 % 3)))`

`(i % n)`: `i` initial values, then repeat `n` times



Slow-to-Fast Transition

- $*_{nf}$ | $*_{nb}$ | $*_{n?}$: Slow-to-fast
  (forward or backward concomitance)



Macro-scheduling using Integer Linear Programming (ILP)

Usual Workflow

1. `$ presseail example2.ail --write-lp example2.lp`
   writes the scheduling constraints to a file
2. Call `cplex`
3. `$ presseail example2.ail --read-sol example2.sol --compile 1`
   reads the solution and generates code

Macro-scheduling using Integer Linear Programming (ILP)

Usual Workflow

1. `$ presseail example2.ail --write-lp example2.lp`
   writes the scheduling constraints to a file
2. Call `cplex`
3. `$ presseail example2.ail --read-sol example2.sol --compile 1`
   reads the solution and generates code

Testing simple examples

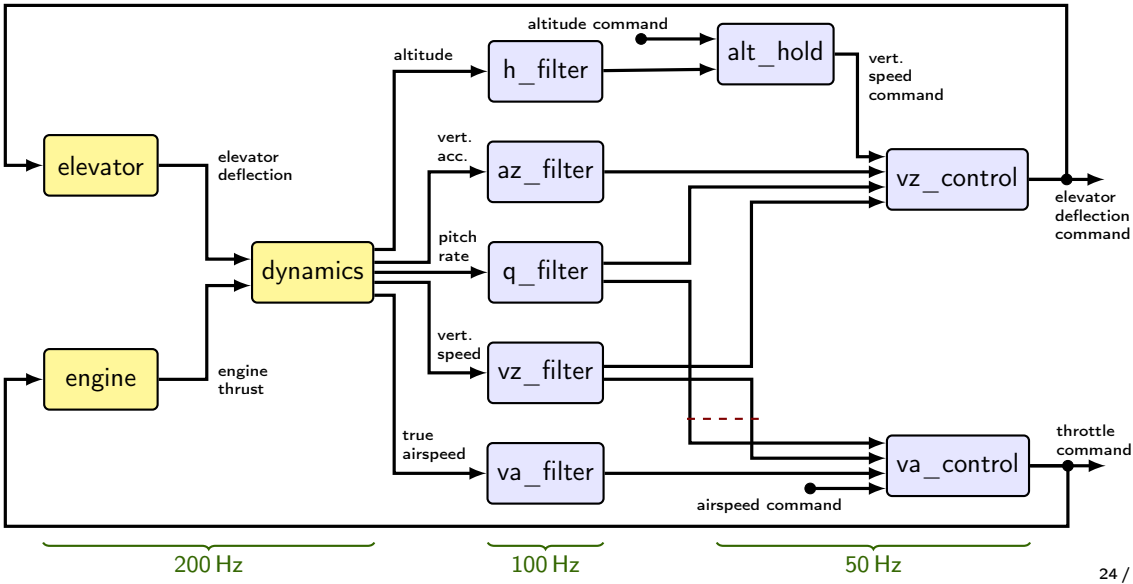
- `$ presseail example2.ail --glpk --compile 1`

```
Minimize
  rmax.equ

Subject to
  pw.def0.filter: pw.0.filter + pw.1.filter + pw.2.filter = 1
  pw.def1.filter: 2 pw.2.filter + pw.1.filter - p.filter = 0
  ...
  depd.wr.p.read.p.filter_5: p.filter - p.read >= 0
  ...
  rbnd.cpu_8: 5 pw.0.filter_1 + 5 pw.0.filter_0 + 5 pw.0.filter <= 8
  rbnd.cpu_7: 5 pw.1.filter_1 + 5 pw.1.filter_0 + 5 pw.1.filter <= 8
  rbnd.cpu_6: 5 pw.2.filter_1 + 5 pw.2.filter_0 + 5 pw.2.filter <= 8

Bounds                          General
  0 <= p.read < 3                 p.read p.filter ...
  0 <= p.filter < 3             Binary
  ...                             pw.0.read pw.1.read pw.2.read pw.0.filter ...
End
```

Plan

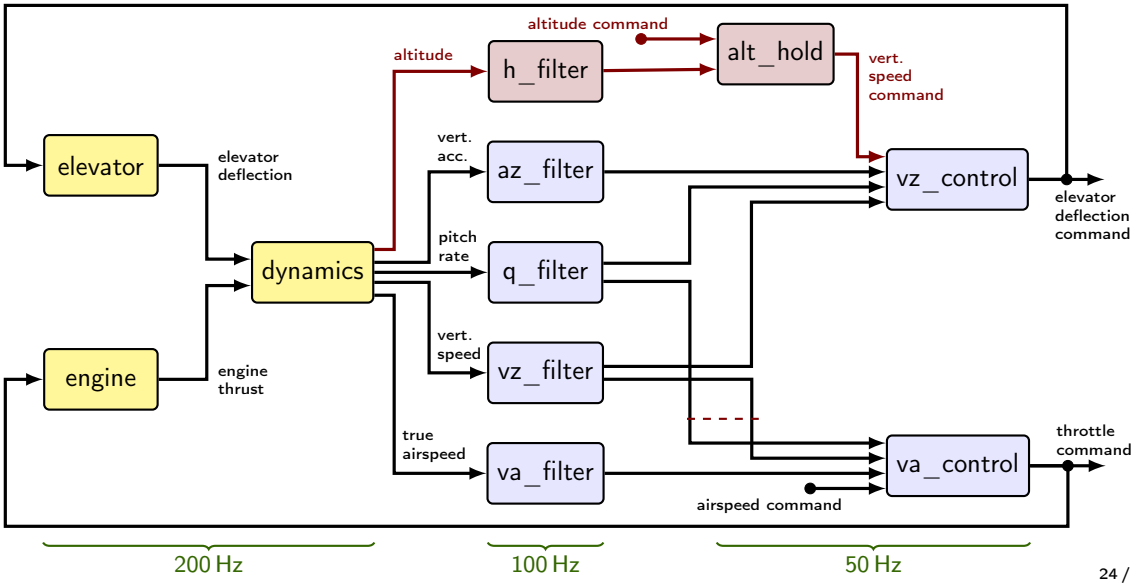
The ROSACE Case Study

[Pagetti, Saussié, Gratia, Noulard, and Siron (2014): The ROSACE Case Study: From Simulink Specification to Multi/Many-Core Execution]



The ROSACE Case Study

[Pagetti, Saussié, Gratia, Noulard, and Siron (2014): The ROSACE Case Study: From Simulink Specification to Multi/Many-Core Execution]



The ROSACE Case Study

[Pagetti, Saussié, Gratia, Noulard, and Siron (2014): The ROSACE Case Study: From Simulink Specification to Multi/Many-Core Execution]

```
resource ops : int
node alt_hold (h_c, h_f : float) returns (vz_c : float) requires (ops = 201);
...
const H200 : rate = 1 / 2 (* base clock = 400Hz *)
const H100 : rate = 1 / 4
const H50  : rate = 1 / 8
const H10  : rate = 1 / 40

node assemblage1( h_c : float :: H10 last = 0.;   (* altitude command *)
                  va_c : float :: H10 last = 0.)  (* airspeed command *)
returns (d_th_c : float :: H50 last = 1.6402;     (* throttle command *)
         d_e_c  : float :: H50 last = 0.0186)     (* elevator deflection command *)
var  h_f : float :: H100; (* altitude *)
    vz_c : float :: H50;  (* vertical speed command *)
    ...
let
   h_f =  h_filter( h when (? % 2) );
  vz_c = alt_hold( current(h_c, (? % 5)), h_f when (? % 2) );
  ...
  resource balance ops;
  latency assemblage exists <= 2
    (dynamics, h_filter, alt_hold, vz_control, elevator);
tel
```

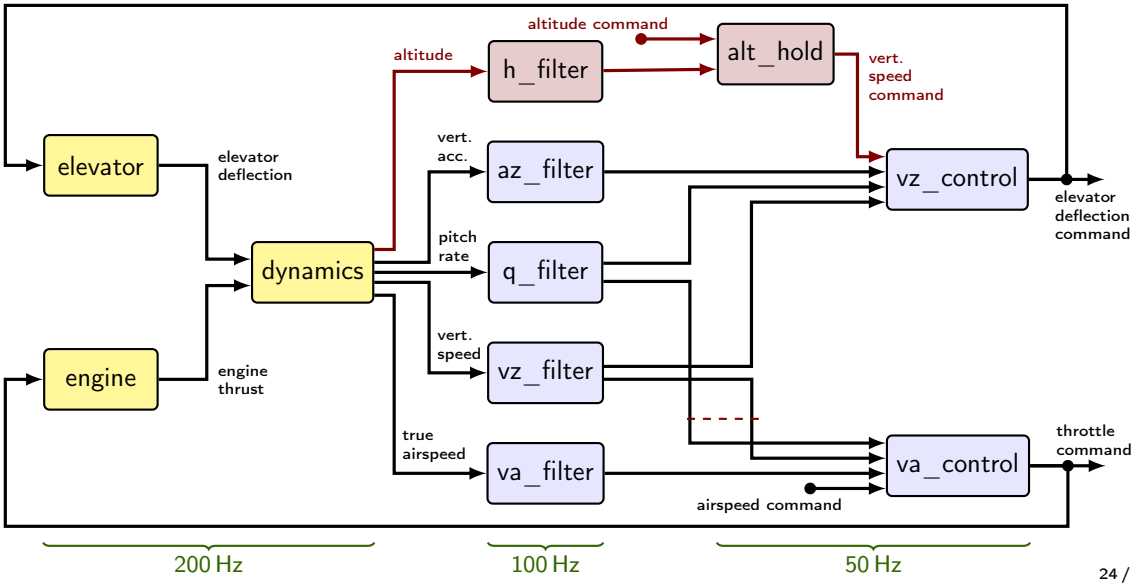
elevator deflection command

throttle command

200 Hz | 100 Hz | 50 Hz

The ROSACE Case Study

[Pagetti, Saussié, Gratia, Noulard, and Siron (2014): The ROSACE Case Study: From Simulink Specification to Multi/Many-Core Execution]

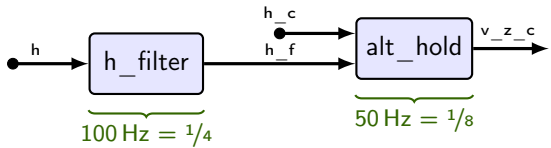


```
node assemblage(h_c, va_c : float rate(100, 0))
returns (d_th_c, d_e_c : float)
var vz_c : float;
    d_e, th, h, az, va, q, vz : float;
    vz_f, va_f, h_f, az_f, q_f : float;
let
  (* 200Hz *)
  d_e = elevator(d_e_c *^ 4);
  th = engine(d_th_c *^ 4);
  (va, az, q, vz, h) = dynamics(1.6402 fby th, 0.0186 fby d_e);
  (* 100Hz *)
   h_f =  h_filter( h /^ 2);
  az_f = az_filter(az /^ 2); ...
  (* 50Hz *)
  vz_c = alt_hold(h_c *^ 5, h_f /^ 2);
  d_e_c = vz_control(vz_c, vz_f /^ 2, q_f ^/ 2,
                     az_f /^ 2);
  d_th_c = va_control(va_c *^ 5, va_f /^ 2,
                      q_f /^ 2, vz_f /^ 2);

  (* dynamics -> h_filter -> alt_hold -> vz_control -> elevator <= 2 *)
  (* scheduled as real-time tasks with WCET, precedence, deadlines *)
tel
```

```
node assemblage(h_c, va_c : float :: 1/40 last = 0.)
returns (d_th_c, d_e_c : float :: 1/8 last = (1.6402, 0.0186))
var vz_c : float :: 1/8;
    d_e, th, h, az, va, q, vz : float :: 1/2;
    vz_f, va_f, h_f, az_f, q_f : float :: 1/4;
let
  (* 200Hz = 1/2 *)
  d_e = elevator(current(d_e_c, (? % 4)));
  th = engine(current(d_th_c, (? % 4)));
  (va, az, q, vz, h) = dynamics(th, d_e);
  (* 100Hz = 1/4 *)
   h_f =  h_filter( h when (? % 2));
  az_f = az_filter(az when (? % 2)); ...
  (* 50Hz = 1/8 *)
  vz_c = alt_hold(current(h_c, (? % 5)), h_f when (? % 2));
  d_e_c = vz_control(vz_c, vz_f when (? % 2), q_f when (? % 2),
                     az_f when (? % 2));
  d_th_c = va_control(current(va_c, (? % 5)), va_f when (? % 2),
                      q_f when (? % 2), vz_f when (? % 2));

  latency exists <= 2 (dynamics, h_filter, alt_hold, vz_control, elevator);
  resource balance ops;
tel
```

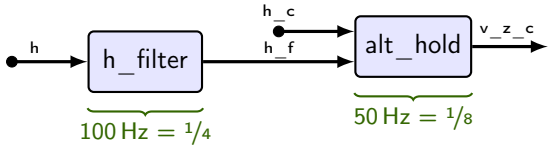
Compilation: Model ⇒ Scheduling ⇒ Generated Code

```
 h_f =  h_filter(h when (? % 2));
vz_c = alt_hold(current(h_c, (? % 5)),
                h_f when (? % 2));
```



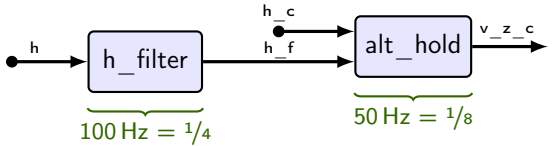
Compilation: Model ⇒ Scheduling ⇒ Generated Code

```
 h_f =  h_filter(h when (? % 2));
vz_c = alt_hold(current(h_c, (? % 5)),
                h_f when (? % 2));
```



Compilation: Model ⇒ Scheduling ⇒ Generated Code

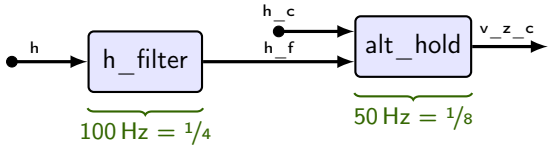
```
h_f =  h_filter(h when (? % 2));
vz_c = alt_hold(current(h_c, (? % 5)),
                h_f when (? % 2));
```




```
static int c_30 = 0;

void step0()
{
    if (c_30 % 2 == 0) {
        if (c_30 % 4 == 2) {
            h_filter();      // ***
            ...
        }
    } else {
        ...
    }
    switch (c_30) {
    case 2: va_control(); break;
    case 6: alt_hold();      // ***
            vz_control();
            break;
    }
    c_30 = (c_30 + 1) % 8;
}
```

Compilation: Model ⇒ Scheduling ⇒ Generated Code

```
h_f =  h_filter(h when (? % 2));
vz_c = alt_hold(current(h_c, (? % 5)),
                h_f when (? % 2));
```

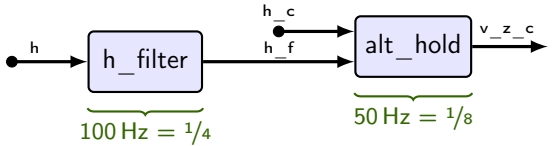


- Source: dataflow semantics
- Target: C code implicitly writing and reading static variables

```
static int c_30 = 0;

void step0()
{
    if (c_30 % 2 == 0) {
        if (c_30 % 4 == 2) {
            h_filter();      // ***
            ...
        }
    } else {
        ...
    }
    switch (c_30) {
    case 2: va_control(); break;
    case 6: alt_hold();      // ***
            vz_control();
            break;
    }
    c_30 = (c_30 + 1) % 8;
}
```

Compilation: Model ⇒ Scheduling ⇒ Generated Code

```
h_f =  h_filter(h when (? % 2));
vz_c = alt_hold(current(h_c, (? % 5)),
                h_f when (? % 2));
```



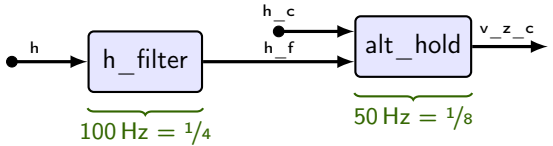
- Source: dataflow semantics
- Target: C code implicitly writing and reading static variables

```
static int c_30 = 0;

void step0()
{
    if (c_30 % 2 == 0) {
        if (c_30 % 4 == 2) {
            h_filter();      // ***
            ...
        }
    } else {
        ...
    }
    switch (c_30) {
    case 2: va_control(); break;
    case 6: alt_hold();      // ***
            vz_control();
            break;
    }
    c_30 = (c_30 + 1) % 8;
}
```

f (concomitance)

Compilation: Model ⇒ Scheduling ⇒ Generated Code

```
h_f =  h_filter(h when (? % 2));
vz_c = alt_hold(current(h_c, (? % 5)),
                h_f when (? % 2));
```

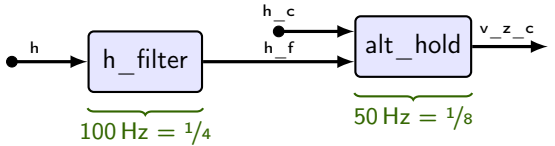


- Source: dataflow semantics
- Target: C code implicitly writing and reading static variables

```
static int c_30 = 0;

void step0()
{
    if (c_30 % 2 == 0) {
        if (c_30 % 4 == 2) {
            h_filter();       // ***
            ...
        }
    } else {
        ...
    }
    switch (c_30) {
    case 0: va_control(); break;
    case 6: alt_hold();       // ***
            vz_control();
            break;
    }
    c_30 = (c_30 + 1) % 8;
}
```

Compilation: Model ⇒ Scheduling ⇒ Generated Code

```
h_f =  h_filter(h when (? % 2));
vz_c = alt_hold(current(h_c, (? % 5)),
                h_f when (? % 2));
```

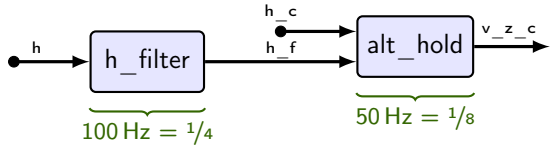


- Source: dataflow semantics
- Target: C code implicitly writing and reading static variables

```
static int c_30 = 0;

void step0()
{
    switch (c_30) {
    case 2: va_control(); break;
    case 6: alt_hold();      // ***
            vz_control();
            break;
    }
    if (c_30 % 2 == 0) {
        if (c_30 % 4 == 2) {
            h_filter();      // ***
            ...
        }
    } else {
        ...
    }
    c_30 = (c_30 + 1) % 8;
}
```

Compilation: Model ⇒ Scheduling ⇒ Generated Code

```
h_f =  h_filter(h when (? % 2));
vz_c = alt_hold(current(h_c, (? % 5)),
                h_f when (? % 2));
```



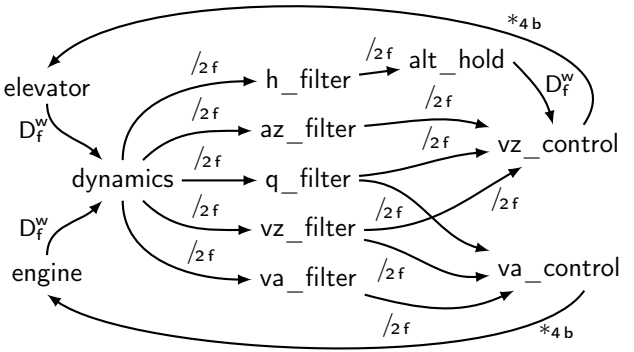
- Source: dataflow semantics
- Target: C code implicitly writing and reading static variables

```
static int c_30 = 0;

void step0()
{
    switch (c_30) {
    case 2: va_control(); break;
    case 6: alt_hold();      // ***
            vz_control();
            break;
    }
    if (c_30 % 2 == 0) {
        if (c_30 % 4 == 2) {
            h_filter();      // ***
            ...
        }
    } else {
        ...
    }
    c_30 = (c_30 + 1) % 8;
}
```

b (concomitance)

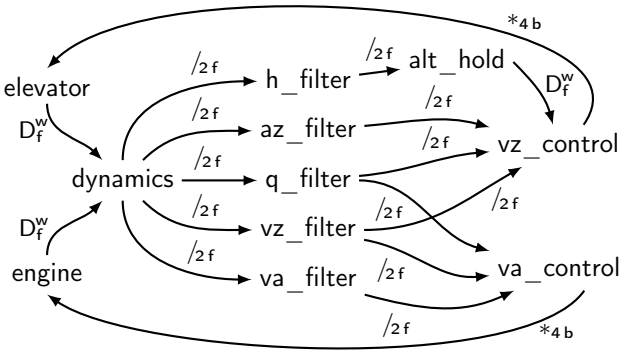
Plan

Flowgraphs



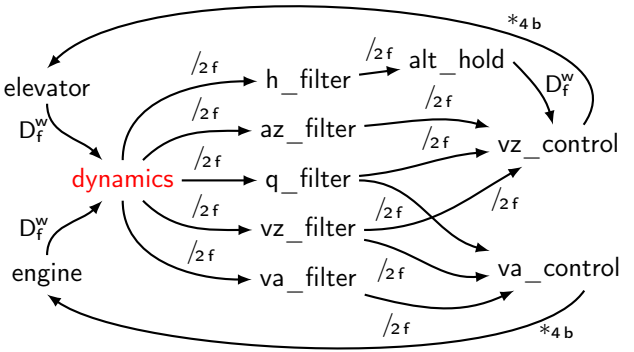
- Generate flowgraph from program
  - `r = w + 1` becomes $w \rightarrow r$
  - `r = last w + 1` also becomes $w \rightarrow r$
- Annotate edges with
  - Rate-transitions and data dependencies
  - *Concomitance*: order within cycle
- Analyze the graph to identify cycles (strongly-connected components)
- Eliminate cycles
  - Changing the concomitance
  - Dropping data dependencies

Flowgraphs



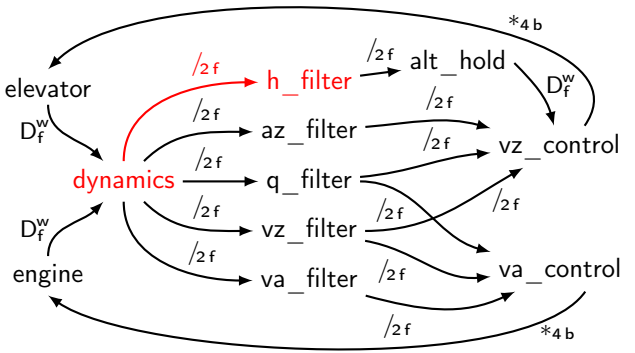
- Generate flowgraph from program
  - `r = w + 1` becomes $w \rightarrow r$
  - `r = last w + 1` also becomes $w \rightarrow r$
- Annotate edges with
  - Rate-transitions and data dependencies
  - *Concomitance*: order within cycle
- Analyze the graph to identify cycles (strongly-connected components)
- Eliminate cycles
  - Changing the concomitance
  - Dropping data dependencies

Flowgraphs



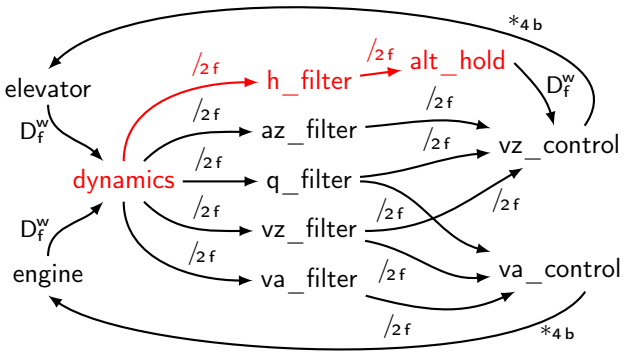
- Generate flowgraph from program
  - `r = w + 1` becomes $w \rightarrow r$
  - `r = last w + 1` also becomes $w \rightarrow r$
- Annotate edges with
  - Rate-transitions and data dependencies
  - *Concomitance*: order within cycle
- Analyze the graph to identify cycles (strongly-connected components)
- Eliminate cycles
  - Changing the concomitance
  - Dropping data dependencies

Flowgraphs



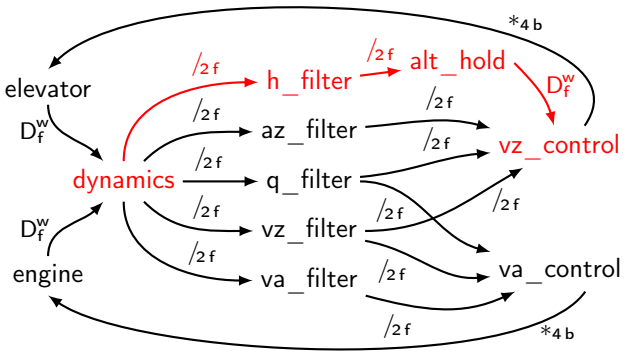
- Generate flowgraph from program
  - `r = w + 1` becomes $w \to r$
  - `r = last w + 1` also becomes $w \to r$
- Annotate edges with
  - Rate-transitions and data dependencies
  - *Concomitance*: order within cycle
- Analyze the graph to identify cycles (strongly-connected components)
- Eliminate cycles
  - Changing the concomitance
  - Dropping data dependencies

Flowgraphs



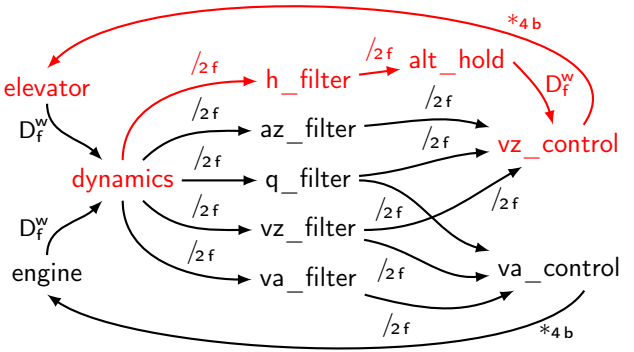
- Generate flowgraph from program
  - `r = w + 1` becomes $w \rightarrow r$
  - `r = last w + 1` also becomes $w \rightarrow r$
- Annotate edges with
  - Rate-transitions and data dependencies
  - *Concomitance*: order within cycle
- Analyze the graph to identify cycles (strongly-connected components)
- Eliminate cycles
  - Changing the concomitance
  - Dropping data dependencies

Flowgraphs



- Generate flowgraph from program
  - `r = w + 1` becomes $w \rightarrow r$
  - `r = last w + 1` also becomes $w \rightarrow r$
- Annotate edges with
  - Rate-transitions and data dependencies
  - *Concomitance*: order within cycle
- Analyze the graph to identify cycles (strongly-connected components)
- Eliminate cycles
  - Changing the concomitance
  - Dropping data dependencies

Flowgraphs



- Generate flowgraph from program
  - `r = w + 1` becomes $w \to r$
  - `r = last w + 1` also becomes $w \to r$
- Annotate edges with
  - Rate-transitions and data dependencies
  - *Concomitance*: order within cycle
- Analyze the graph to identify cycles (strongly-connected components)
- Eliminate cycles
  - Changing the concomitance
  - Dropping data dependencies

Flowgraph links

x	$eq_w \xrightarrow{\mathsf{D}^{\mathsf{w}}_{\mathsf{f}}} eq_r$	
`last` x	$eq_w \xrightarrow{\mathsf{D}^{\mathsf{r}}_{\mathsf{b}}} eq_r$	
unconstrained	$eq_w \xrightarrow{\mathsf{D}^{?}_{\mathsf{f}}} eq_r$	
x `when` $(\cdot \,\%\, n)$	$eq_w \xrightarrow{/_n \mathsf{f}} eq_r$	
(`last` x) `when` $(\cdot \,\%\, n)$	$eq_w \xrightarrow{/^{\mathsf{L}}_n \mathsf{b}} eq_r$	
`current`$(x, (\cdot \,\%\, n))$	$eq_w \xrightarrow{*_n \mathsf{f}} eq_r$	(by default)
	$eq_w \xrightarrow{*_n \mathsf{b}} eq_r$	(if 'fast-first')
`current`(`last` $x, (\cdot \,\%\, n)$)	*forbidden*	

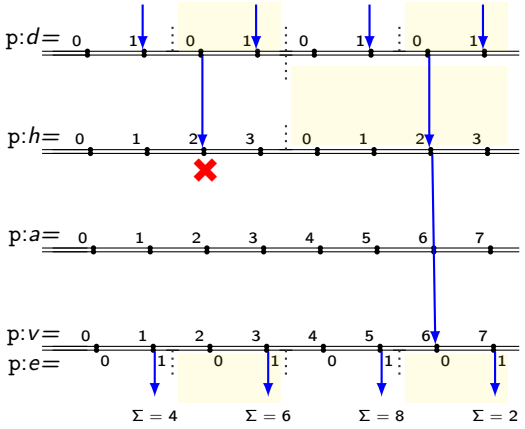
Schedule

	ops	*phase*	
`elevator`	98	$1 \bmod 2$	(p:*e*)
`engine`	82	$0 \bmod 2$	
`dynamics`	1174	$1 \bmod 2$	(p:*d*)
`h_filter`	38	$2 \bmod 4$	(p:*h*)
`az_filter`	37	$2 \bmod 4$	
`q_filter`	37	$2 \bmod 4$	
`vz_filter`	37	$2 \bmod 4$	
`va_filter`	38	$2 \bmod 4$	
`alt_hold`	201	$6 \bmod 8$	(p:*a*)
`vz_control`	88	$6 \bmod 8$	(p:*v*)
`va_control`	90	$2 \bmod 8$	

- Our ROSACE implementation
  - » Cycle period = 2.5 ms (400 Hz)
  - » Allow load balancing of fastest components (200 Hz)
  - » The ops resource estimates the computations required
- Assign each component a phase relative to its period (in terms of base cycles)
- Balance ops per cycle
- Respect end-to-end latency

Schedule

	ops	*phase*	
elevator	98	1 %2	(p:*e*)
engine	82	0 %2	
dynamics	1174	1 %2	(p:*d*)
h_filter	38	2 %4	(p:*h*)
az_filter	37	2 %4	
q_filter	37	2 %4	
vz_filter	37	2 %4	
va_filter	38	2 %4	
alt_hold	201	6 %8	(p:*a*)
vz_control	88	6 %8	(p:*v*)
va_control	90	2 %8	

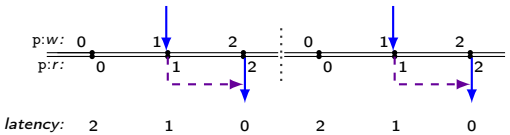



```
latency exists <= 2 (dynamics, h_filter, alt_hold, vz_control, elevator);
latency exists <= 2 (d,        h,        a,        v,          e);
```

Plan

Minimum Pairwise Latency: same period

Direct communication: `r = w`

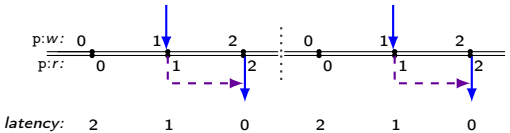


$\mathtt{r}, \mathtt{w} :: 1/3$
$\mathrm{p}{:}w = 1$
$\mathrm{p}{:}r = 2$
$lat. = 1$

- $eq_w \xrightarrow{\mathrm{D}^{\mathrm{w}}_{\mathrm{f}}} eq_r$
- Write-before-read: $\mathrm{p}{:}w \leq \mathrm{p}{:}r$
- $0 \leq \mathrm{p}{:}r - \mathrm{p}{:}w < period$

Minimum Pairwise Latency: same period

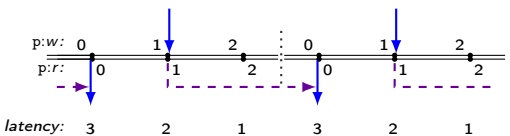
Direct communication: `r = w`



$\mathtt{r}, \mathtt{w} :: 1/3$
$\mathrm{p}{:}w = 1$
$\mathrm{p}{:}r = 2$
$lat. = 1$

- $eq_w \xrightarrow{\mathrm{D}^{\mathrm{w}}_{\mathrm{f}}} eq_r$
- Write-before-read: $\mathrm{p}{:}w \leq \mathrm{p}{:}r$
- $0 \leq \mathrm{p}{:}r - \mathrm{p}{:}w < period$

Delayed communication: `r = last w`

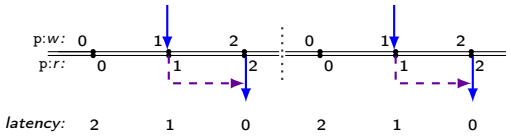


$\mathtt{r}, \mathtt{w} :: 1/3$
$\mathrm{p}{:}w = 1$
$\mathrm{p}{:}r = 0$
$lat. = 2$

- $eq_w \xrightarrow{\mathrm{D}^{\mathrm{r}}_{\mathrm{b}}} eq_r$
- Read-before-write: $\mathrm{p}{:}r \leq \mathrm{p}{:}w$
- $0 < \mathrm{p}{:}r - \mathrm{p}{:}w + period \leq period$

Minimum Pairwise Latency: same period

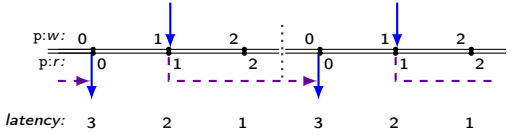
Direct communication: `r = w`



$\mathtt{r}, \mathtt{w} :: 1/3$
$\mathrm{p}{:}w = 1$
$\mathrm{p}{:}r = 2$
$lat. = 1$

- $eq_w \xrightarrow{\mathrm{D}^{\mathrm{w}}_{\mathrm{f}}} eq_r$
- Write-before-read: $\mathrm{p}{:}w \leq \mathrm{p}{:}r$
- $0 \leq \mathrm{p}{:}r - \mathrm{p}{:}w < period$

Delayed communication: `r = last w`



$\mathtt{r}, \mathtt{w} :: 1/3$
$\mathrm{p}{:}w = 1$
$\mathrm{p}{:}r = 0$
$lat. = 2$

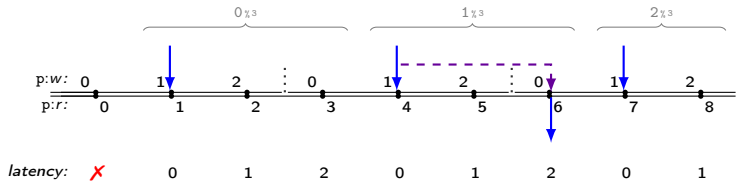
- $eq_w \xrightarrow{\mathrm{D}^{\mathrm{r}}_{\mathrm{b}}} eq_r$
- Read-before-write: $\mathrm{p}{:}r \leq \mathrm{p}{:}w$
- $0 < \mathrm{p}{:}r - \mathrm{p}{:}w + period \leq period$

Unconstrained communication (`r = last? w`)

- $eq_w \xrightarrow{\mathrm{D}^{?}_{\mathrm{f}}} eq_r$: $(\mathrm{p}{:}r - \mathrm{p}{:}w + \mathrm{period}(w)) \bmod \mathrm{period}(w)$
- $eq_w \xrightarrow{\mathrm{D}^{?}_{\mathrm{b}}} eq_r$: $((\mathrm{p}{:}r - \mathrm{p}{:}w + \mathrm{period}(w) - 1) \bmod \mathrm{period}(w)) + 1$

Minimum Pairwise Latency: fast-to-slow

`r = w when (? % 3)` $\quad eq_w \xrightarrow{/_{nf}} eq_r \quad$ $(\mathrm{p}{:}r - \mathrm{p}{:}w) \bmod \mathrm{period}(w)$



$\mathtt{w} :: 1/3$

$\mathtt{r} :: 1/9$

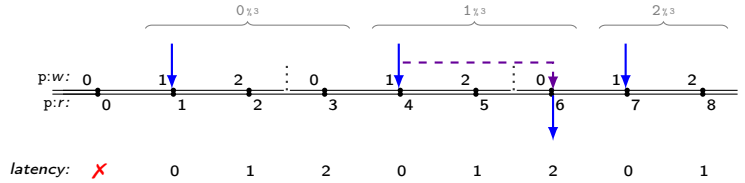
$\mathrm{p}{:}w = 1$

$\mathrm{p}{:}r = 6$

$lat. = 2$

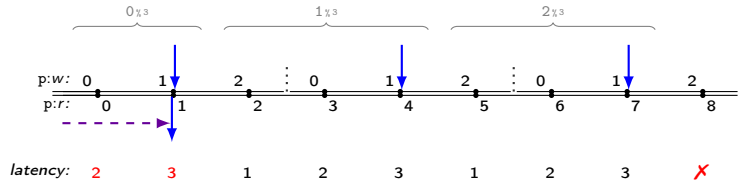
Minimum Pairwise Latency: fast-to-slow

`r = w when (? % 3)` $eq_w \xrightarrow{/_{nf}} eq_r$ $(\mathrm{p}{:}r - \mathrm{p}{:}w) \bmod \mathrm{period}(w)$



`r = (last e) when (0 % 3)` $eq_w \xrightarrow{/_{nb}} eq_r$

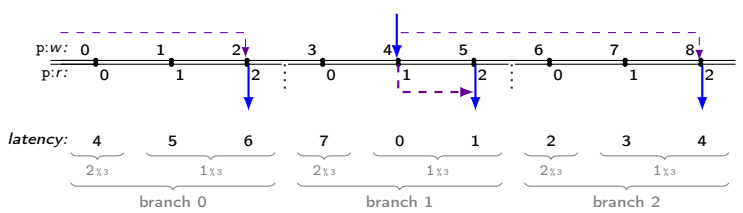
$((\mathrm{period}(w) + \mathrm{p}{:}r - \mathrm{p}{:}w - 1) \bmod \mathrm{period}(w)) + 1$



Minimum Pairwise Latency: slow-to-fast

`r = current(w, (1 % 3))` $\quad eq_w \xrightarrow{*_{nf}} eq_r$

$((branch \cdot \mathrm{period}(r) + \mathrm{period}(w) + \mathrm{p}{:}r - \mathrm{p}{:}w - 1) \bmod \mathrm{period}(w)) + 1$

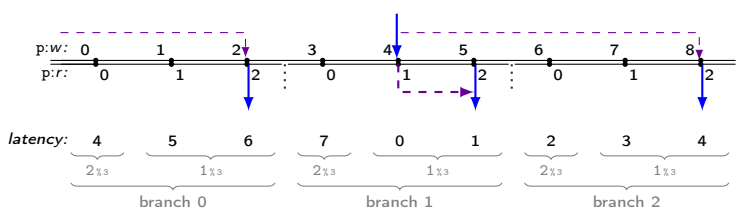


`w` :: $1/9$
`r` :: $1/3$
$\mathrm{p}{:}w = 4$
$\mathrm{p}{:}r = 2$
$lat. = 1, 4$

Minimum Pairwise Latency: slow-to-fast

`r = current(w, (1 % 3))` $\quad eq_w \xrightarrow{*_{nf}} eq_r$

$((branch \cdot \mathrm{period}(r) + \mathrm{period}(w) + \mathrm{p}{:}r - \mathrm{p}{:}w - 1) \bmod \mathrm{period}(w)) + 1$



`w` :: $1/9$
`r` :: $1/3$
p:$w = 4$
p:$r = 2$
$lat. = 1, 4$

`r = current(last w, (? % 3))`?

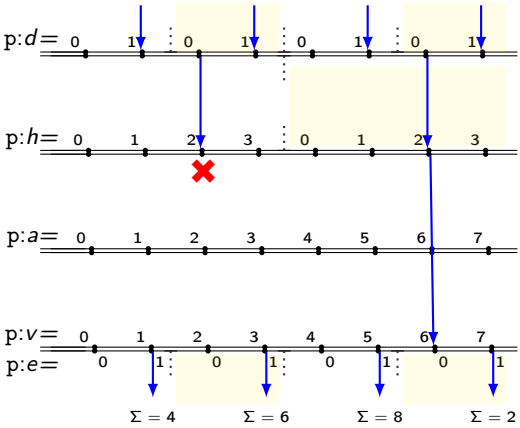
- Not allowed. Not enough ‘memories’.
- Must be normalized to

```
r = current(t, (? % 3));
t = last w;
```

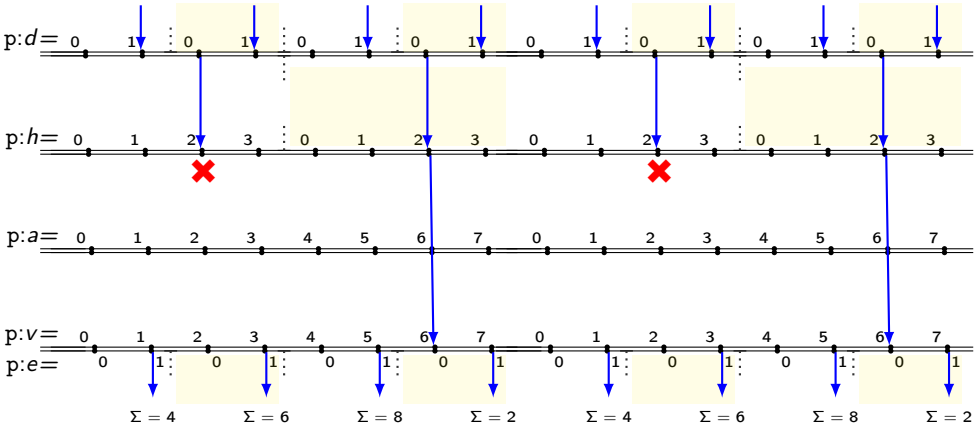
End-to-End Latency: the wrong way

- Define pairwise latencies for each link type.
- Chain them together into a sequence.
- Difficult to handle branching and dead ends.
- Difficult to explain.
- Complicated formulas.
- There's a better way...

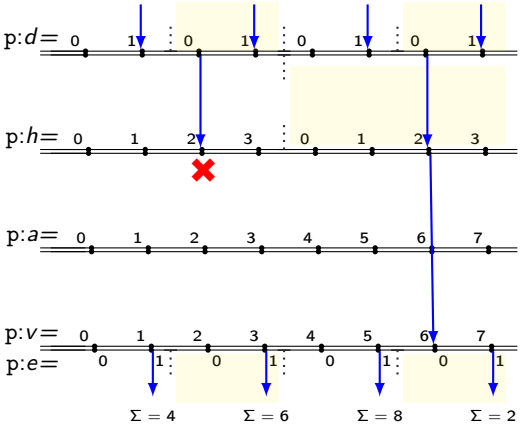
Constraining End-to-end Latency



Constraining End-to-end Latency

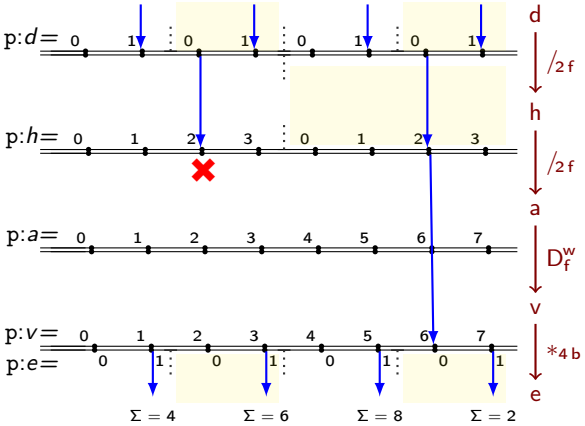


Constraining End-to-end Latency



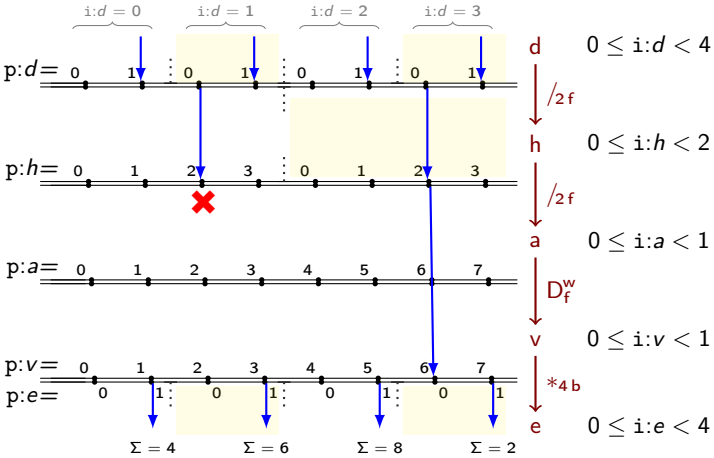
(View online at https://www.tbrk.org/dataflow/showlatency)

Constraining End-to-end Latency



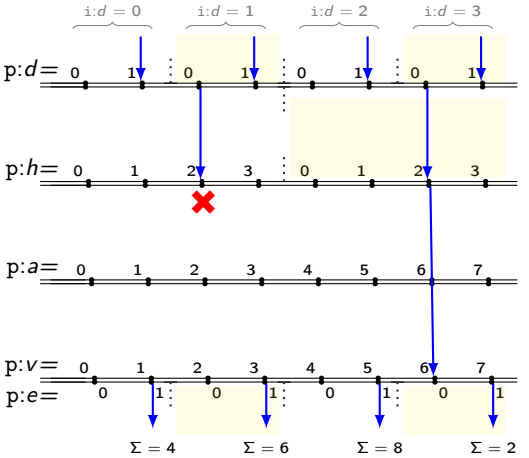
(View online at https://www.tbrk.org/dataflow/showlatency)

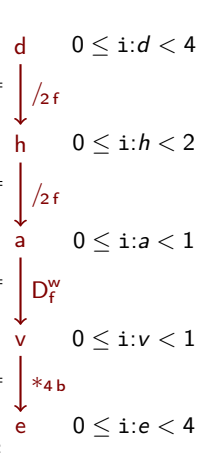
Constraining End-to-end Latency



(View online at https://www.tbrk.org/dataflow/showlatency)

Constraining End-to-end Latency





$0 \le \mathrm{i}{:}d < 4$

$0 \le \mathrm{i}{:}h < 2$

$0 \le \mathrm{i}{:}a < 1$

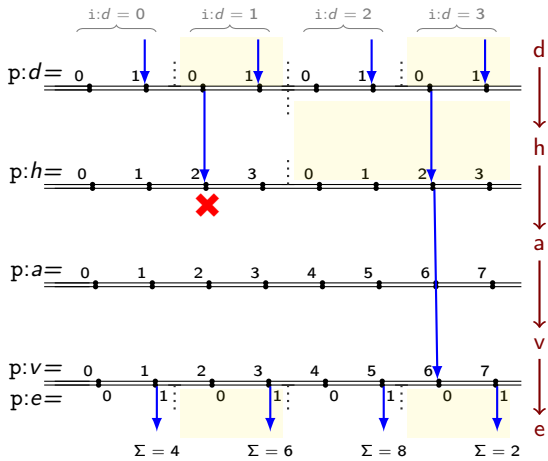
$0 \le \mathrm{i}{:}v < 1$

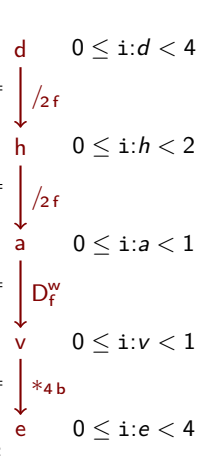
$0 \le \mathrm{i}{:}e < 4$

$$0 \le \mathtt{lat}_{d,h} < 2$$
$$2 \cdot \mathrm{i}{:}d + \mathrm{p}{:}d + \mathtt{lat}_{d,h} = 4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h$$

(View online at https://www.tbrk.org/dataflow/showlatency)

Constraining End-to-end Latency





$0 \le \mathrm{i}{:}d < 4$

$0 \le \mathrm{i}{:}h < 2$

$0 \le \mathrm{i}{:}a < 1$

$0 \le \mathrm{i}{:}v < 1$

$0 \le \mathrm{i}{:}e < 4$

$0 \le \mathtt{lat}_{d,h} < 2$

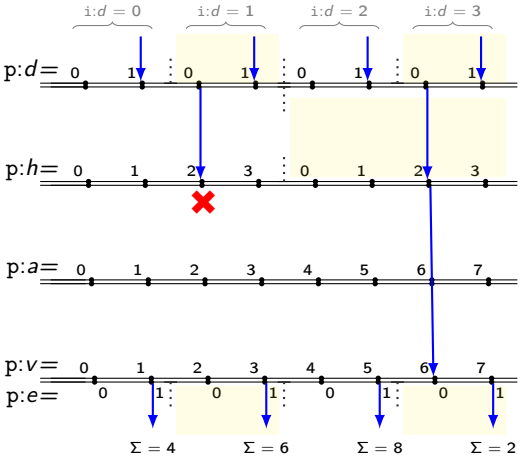
$2 \cdot \mathrm{i}{:}d + \mathrm{p}{:}d + \mathtt{lat}_{d,h} = 4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h$

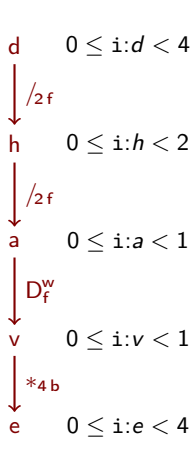
$0 \le \mathtt{lat}_{h,a} < 4$

$4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h + \mathtt{lat}_{h,a} = 8 \cdot \mathrm{i}{:}a + \mathrm{p}{:}a$

(View online at https://www.tbrk.org/dataflow/showlatency)

Constraining End-to-end Latency





$0 \le \mathrm{i}{:}d < 4$

$0 \le \mathrm{i}{:}h < 2$

$0 \le \mathrm{i}{:}a < 1$

$0 \le \mathrm{i}{:}v < 1$

$0 \le \mathrm{i}{:}e < 4$

$0 \le \mathtt{lat}_{d,h} < 2$

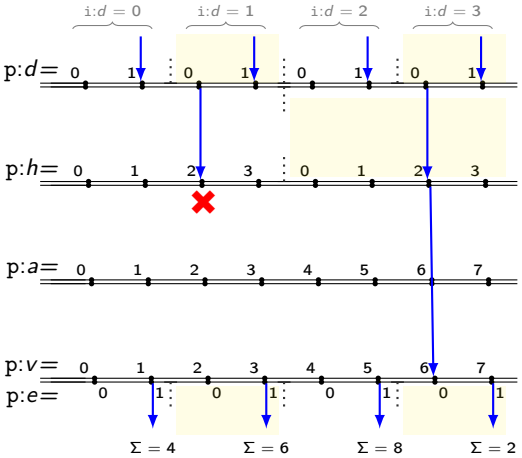
$2 \cdot \mathrm{i}{:}d + \mathrm{p}{:}d + \mathtt{lat}_{d,h} = 4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h$

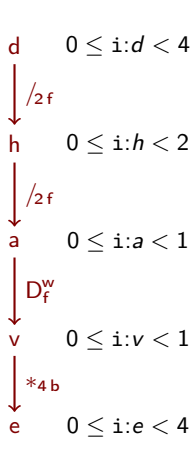
$0 \le \mathtt{lat}_{h,a} < 4$

$4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h + \mathtt{lat}_{h,a} = 8 \cdot \mathrm{i}{:}a + \mathrm{p}{:}a$

(View online at https://www.tbrk.org/dataflow/showlatency)

Constraining End-to-end Latency





$$0 \le \mathrm{i}{:}d < 4$$

$$0 \le \mathrm{lat}_{d,h} < 2$$
$$2 \cdot \mathrm{i}{:}d + \mathrm{p}{:}d + \mathrm{lat}_{d,h} = 4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h$$

$$0 \le \mathrm{i}{:}h < 2$$

$$0 \le \mathrm{lat}_{h,a} < 4$$
$$4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h + \mathrm{lat}_{h,a} = 8 \cdot \mathrm{i}{:}a + \mathrm{p}{:}a$$

$$0 \le \mathrm{i}{:}a < 1$$

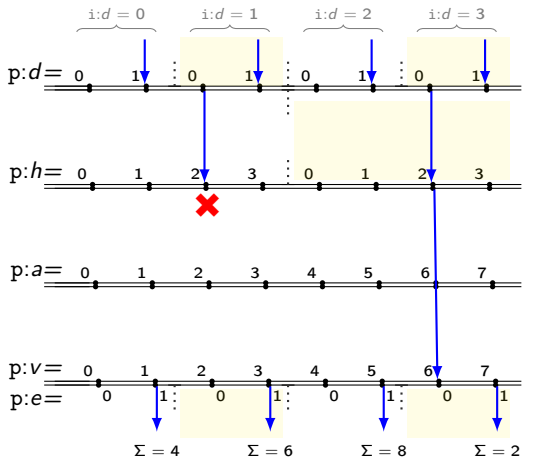
$$0 \le \mathrm{lat}_{a,v} < 8$$
$$8 \cdot \mathrm{i}{:}a + \mathrm{p}{:}a + \mathrm{lat}_{a,v} = 8 \cdot \mathrm{i}{:}v + \mathrm{p}{:}v$$

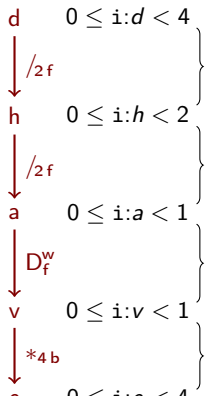
$$0 \le \mathrm{i}{:}v < 1$$

$$0 \le \mathrm{i}{:}e < 4$$

(View online at https://www.tbrk.org/dataflow/showlatency)

Constraining End-to-end Latency





$$0 \le \mathrm{i}{:}d < 4$$

$$0 \le \mathrm{i}{:}h < 2$$

$$0 \le \mathrm{i}{:}a < 1$$

$$0 \le \mathrm{i}{:}v < 1$$

$$0 \le \mathrm{i}{:}e < 4$$

$$0 \le \mathtt{lat}_{d,h} < 2$$
$$2 \cdot \mathrm{i}{:}d + \mathrm{p}{:}d + \mathtt{lat}_{d,h} = 4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h$$

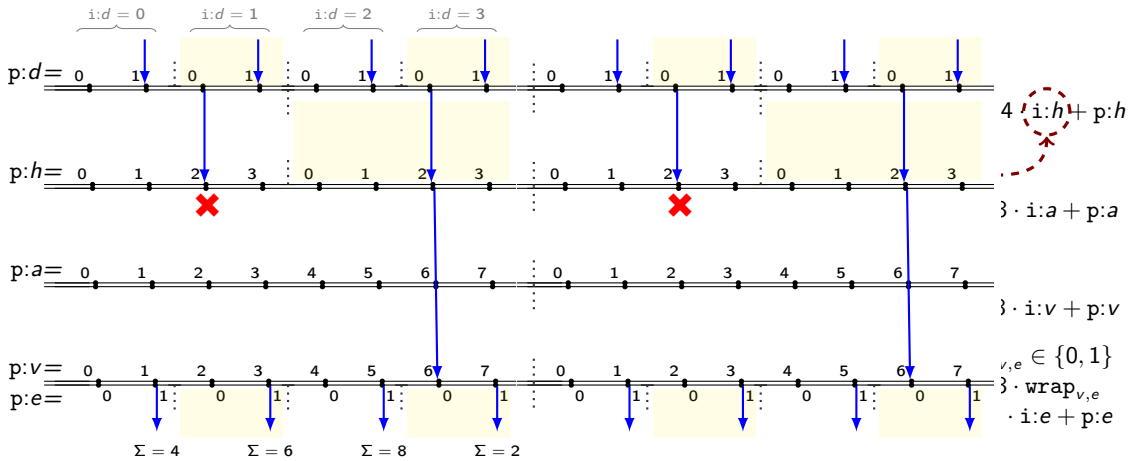
$$0 \le \mathtt{lat}_{h,a} < 4$$
$$4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h + \mathtt{lat}_{h,a} = 8 \cdot \mathrm{i}{:}a + \mathrm{p}{:}a$$

$$0 \le \mathtt{lat}_{a,v} < 8$$
$$8 \cdot \mathrm{i}{:}a + \mathrm{p}{:}a + \mathtt{lat}_{a,v} = 8 \cdot \mathrm{i}{:}v + \mathrm{p}{:}v$$

$$0 < \mathtt{lat}_{v,e} \le 8 \qquad \mathtt{wrap}_{v,e} \in \{0, 1\}$$
$$8 \cdot \mathrm{i}{:}v + \mathrm{p}{:}v + \mathtt{lat}_{v,e} - 8 \cdot \mathtt{wrap}_{v,e} = 2 \cdot \mathrm{i}{:}e + \mathrm{p}{:}e$$

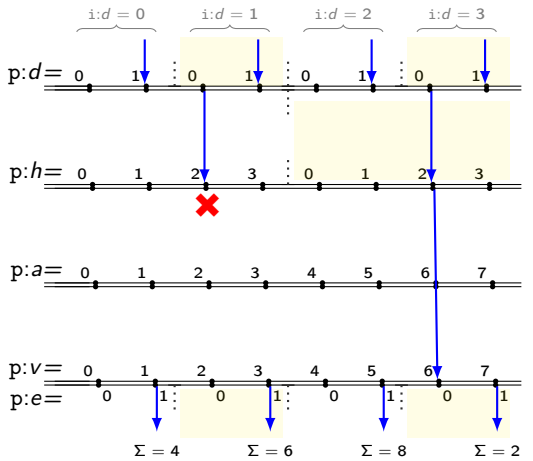
(View online at https://www.tbrk.org/dataflow/showlatency)

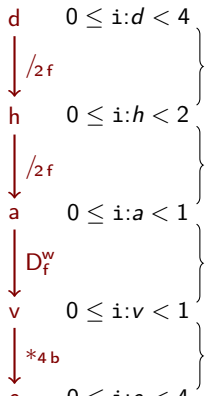
Constraining End-to-end Latency



(View online at https://www.tbrk.org/dataflow/showlatency)

Constraining End-to-end Latency





$$0 \le \mathrm{i}{:}d < 4$$

$$0 \le \mathrm{i}{:}h < 2$$

$$0 \le \mathrm{i}{:}a < 1$$

$$0 \le \mathrm{i}{:}v < 1$$

$$0 \le \mathrm{i}{:}e < 4$$

$$0 \le \mathtt{lat}_{d,h} < 2$$
$$2 \cdot \mathrm{i}{:}d + \mathrm{p}{:}d + \mathtt{lat}_{d,h} = 4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h$$

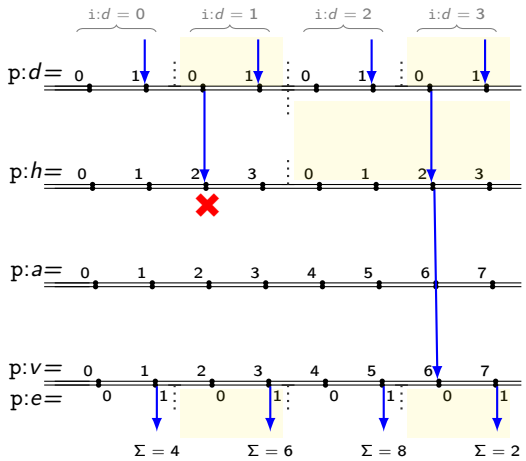
$$0 \le \mathtt{lat}_{h,a} < 4$$
$$4 \cdot \mathrm{i}{:}h + \mathrm{p}{:}h + \mathtt{lat}_{h,a} = 8 \cdot \mathrm{i}{:}a + \mathrm{p}{:}a$$

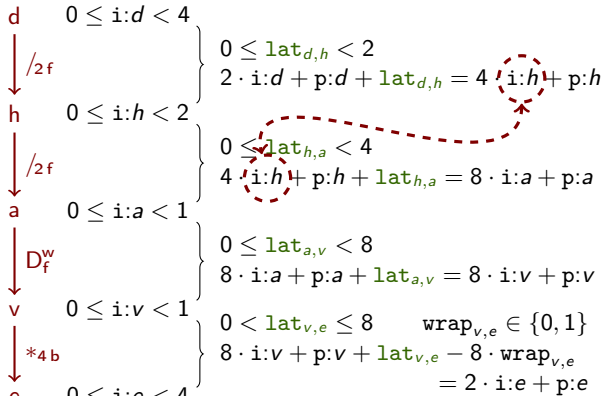
$$0 \le \mathtt{lat}_{a,v} < 8$$
$$8 \cdot \mathrm{i}{:}a + \mathrm{p}{:}a + \mathtt{lat}_{a,v} = 8 \cdot \mathrm{i}{:}v + \mathrm{p}{:}v$$

$$0 < \mathtt{lat}_{v,e} \le 8 \qquad \mathtt{wrap}_{v,e} \in \{0, 1\}$$
$$8 \cdot \mathrm{i}{:}v + \mathrm{p}{:}v + \mathtt{lat}_{v,e} - 8 \cdot \mathtt{wrap}_{v,e} = 2 \cdot \mathrm{i}{:}e + \mathrm{p}{:}e$$

(View online at https://www.tbrk.org/dataflow/showlatency)

Constraining End-to-end Latency





$\mathtt{lat}_{d,h} + \mathtt{lat}_{h,a} + \mathtt{lat}_{a,v} + \mathtt{lat}_{v,e} \leq 2$

(View online at https://www.tbrk.org/dataflow/showlatency)

Chains of constraints

`latency forward/backward` $\leq B\ (\ldots, w, r, \ldots)$

Chains of constraints

`latency forward/backward` $\leq B\ (\ldots, w, r, \ldots)$

For each link $w \xrightarrow{s,c} r$,

Chains of constraints

`latency` `forward`/`backward` $\leq B\ (\ldots, w, r, \ldots)$

For each link $w \xrightarrow{s,c} r$,

$$0 \leq \mathtt{i}{:}r < {}^{hp}/_{\mathrm{period}(r)}$$

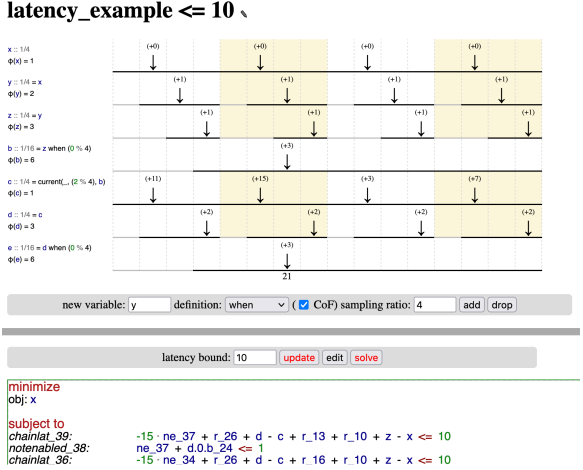
$$\begin{array}{ll} 0 \leq \mathtt{lat}_{w,r} < L & \text{for } c = \mathsf{f} \\ 0 < \mathtt{lat}_{w,r} \leq L & \text{for } c = \mathsf{b} \end{array} \quad \left\{ \begin{array}{ll} \text{where} & L = \mathrm{period}(r) \text{ if } \texttt{forward} \\ \text{and} & L = \mathrm{period}(w) \text{ if } \texttt{backward} \end{array} \right.$$

$$0 \leq \mathtt{wrap}_{w,r} \leq 1 \qquad \begin{array}{l} \text{if } \texttt{forward} \text{ and } s \notin \{\mathsf{D}^{\mathsf{w}}, *_n\} \\ \text{or if } \texttt{backward} \text{ and } s \notin \{\mathsf{D}^{\mathsf{w}}, /_n\} \end{array}$$

$$\mathrm{period}(w) \cdot \mathtt{i}{:}w + \mathtt{p}{:}w + \mathtt{lat}_{w,r} - hp \cdot \mathtt{wrap}_{w,r} = \mathrm{period}(r) \cdot \mathtt{i}{:}r + \mathtt{p}{:}r.$$

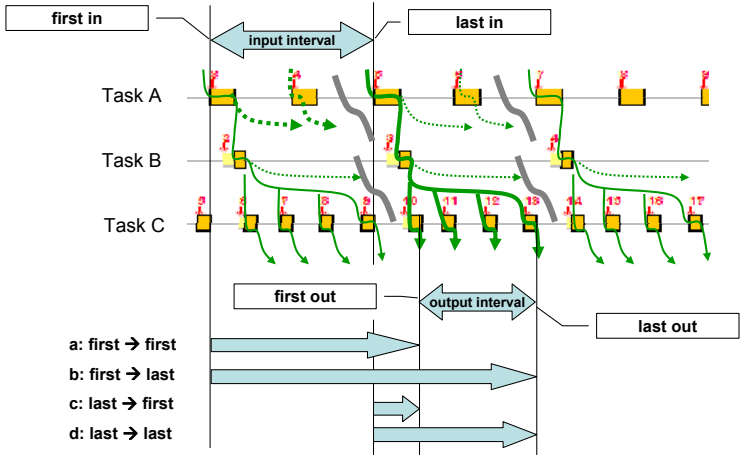
- Each intermediate instance is both a reader and a writer, and thus constrained by two equations.
- `forward`: attach chains from the top
- `backward`: attach chains to the bottom

Showlatency demo



https://www.di.ens.fr/~bourke/showlat/showlatency.html

End-to-End Latency



[Feiertag, Richter, Nordlander, and Jonsson (2008): A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics]

[Girault, Prévot, Quinton, Henia, and Sordon (2018): Improving and Estimating the Precision of Bounds on the Worst-Case Latency of Task Chains], . . .

Plan

Manipulating flow graphs

Before scheduling

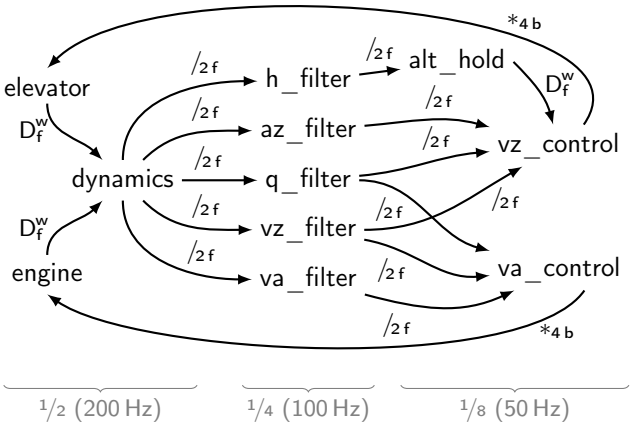
1. Construct flow graph from program
2. Remove potential cycles by flipping the concomitance
3. Use to generate ILP constraints (causality, end-to-end latency)

After scheduling: convert to dependency graph

1. Drop edges between equations that cannot execute in any phase.
2. Flip the cobackward (b) edges.
3. Use with standard algorithm to schedule equations within a step function.

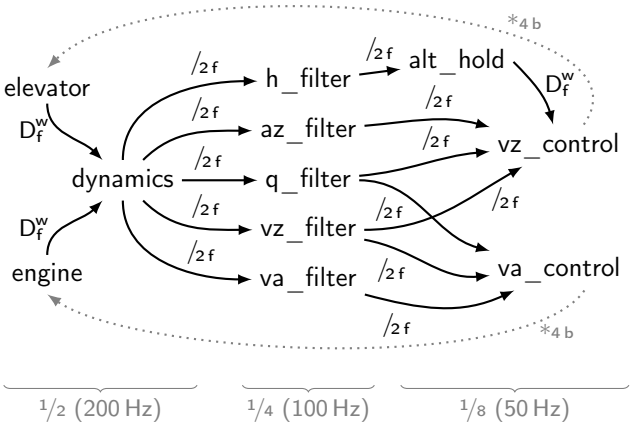
ROSACE example: flow graph → dependency graph

	ops	*phase*
`elevator`	98	$1 \bmod 2$
`engine`	82	$0 \bmod 2$
`dynamics`	1174	$1 \bmod 2$
`h_filter`	38	$2 \bmod 4$
`az_filter`	37	$2 \bmod 4$
`q_filter`	37	$2 \bmod 4$
`vz_filter`	37	$2 \bmod 4$
`va_filter`	38	$2 \bmod 4$
`alt_hold`	201	$6 \bmod 8$
`vz_control`	88	$6 \bmod 8$
`va_control`	90	$2 \bmod 8$



ROSACE example: flow graph → dependency graph

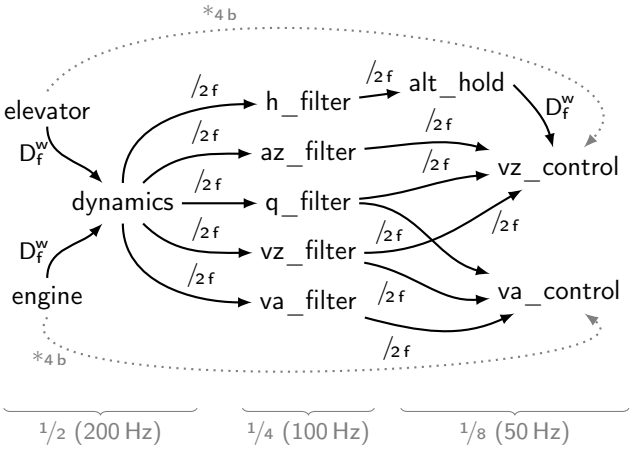
	ops	*phase*
`elevator`	98	$1_{\%2}$
`engine`	82	$0_{\%2}$
`dynamics`	1174	$1_{\%2}$
`h_filter`	38	$2_{\%4}$
`az_filter`	37	$2_{\%4}$
`q_filter`	37	$2_{\%4}$
`vz_filter`	37	$2_{\%4}$
`va_filter`	38	$2_{\%4}$
`alt_hold`	201	$6_{\%8}$
`vz_control`	88	$6_{\%8}$
`va_control`	90	$2_{\%8}$



- Remove all edges between equations that can never execute in the same phase.
- Reverse edges whose concomitance is *b*.

ROSACE example: flow graph → dependency graph

	ops	*phase*
`elevator`	98	$1_{\%2}$
`engine`	82	$0_{\%2}$
`dynamics`	1174	$1_{\%2}$
`h_filter`	38	$2_{\%4}$
`az_filter`	37	$2_{\%4}$
`q_filter`	37	$2_{\%4}$
`vz_filter`	37	$2_{\%4}$
`va_filter`	38	$2_{\%4}$
`alt_hold`	201	$6_{\%8}$
`vz_control`	88	$6_{\%8}$
`va_control`	90	$2_{\%8}$



- Remove all edges between equations that can never execute in the same phase.
- Reverse edges whose concomitance is b.

ROSACE example: generated code

```
static int c = 0;
static float h_c = 0, d_th_c = 1.6402, d_e_c = 0.0186, ...;
static float vz_c, ..., q_f;

void step0()
{
  if (c % 2 == 0) {
    engine();
    if (c % 4 == 2) {
      vz_filter(); h_filter(); va_filter(); q_filter(); az_filter();
    }
  } else {
    elevator(); dynamics();
  }
  switch (c) {
  case 2: va_control(); break;
  case 6: alt_hold(); vz_control(); break;
  }
  c = (c + 1) % 8;
}
```

Code generation

Generalize the clock-directed scheme

[Biernacki, Colaço, Hamon, and Pouzet (2008): Clock-directed modular code generation for synchronous data-flow languages]

- `--compile n` generates n step functions
  - » For the ith step function, $step_i$, List.filter_map equations by phase offset.
  - » Generate dependency graph ignoring variables not in $step_i$ —macro-scheduling guarantees they will already have been calculated.
  - » Micro-schedule equations in $step_i$ w.r.t. dependencies and phase offset/rate.
- Generate multiple Obc step methods, buffer values in state variables.
- Optimize the Obc by joining adjacent case statements.

Code generation: 2

Specialized case construct

```
case (state(c_3) mod 3) {
  0: { skip }
  1: { state(s2) := filter(state(s1)) }
  2: { skip }
  else undefined
};
case (state(c_3) mod 3) {
  0: { state(s1) := filter(s0) }
  1: { skip }
  2: { skip }
  else undefined
};
```

```
case (state(c_3) mod 3) {
  0: { state(s1) := filter(s0) }
  1: { state(s2) := filter(state(s1)) }
  2: { skip }
  else undefined
};
```

Code generation: 2

The 'else undefined' simplifies optimisation under (implicit) invariants

```
case (state(c_3) mod 24) {
  7:  { state(x) := read_real() }
  23: { y := read_real() }
  else undefined }
```

```
if (state(c_3) mod 24 = 7) {
    state(x) := read_real()
} else {
    y := read_real()
}
```

```
case (state(c_3) mod 24) {
  7:  { state(x) := read_real() }
  15: { skip }
  23: { y := read_real() }
  else undefined }
```

```
case (state(c_3) mod 24) {
  7:  { state(x) := read_real() }
  15: { skip }
  23: { y := read_real() }
  else undefined }
```

Plan

More Information

Scheduling and Compiling Rate-Synchronous Programs with End-To-End Latency Constraints

Timothy Bourke
Inria Paris, France
École normale supérieure, PSL University, CNRS, Paris, France

Vincent Bregeon
Airbus Operations S.A.S., Toulouse, France

Marc Pouzet
École normale supérieure, PSL University, CNRS, Paris, France
Inria Paris, France

Abstract

We present an extension of the synchronous-reactive model for specifying multi-rate systems. A set of periodically executed components and their communication dependencies are expressed in a Lustre-like programming language with features for load balancing, resource limiting, and specifying end-to-end latencies. The language abstracts from execution time and phase offsets. This permits simple clock typing rules and a stream-based semantics, but requires each component to execute within an overall base period. A program is compiled to a single periodic task in two stages. First, Integer Linear Programming is used to determine phase offsets using standard encodings for dependencies and load balancing, and a novel encoding for end-to-end latency. Second, a code generation scheme is adapted to produce step functions. As a result, components are synchronous relative to their respective rates, but not necessarily simultaneous relative to the base period. This approach has been implemented in a prototype compiler and validated on an industrial application.

2012 ACM Subject Classification Computer systems organization → Real-time languages; Computer systems organization → Embedded software

Keywords and phrases synchronous-reactive, integer linear programming, code generation

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.1

1 Introduction

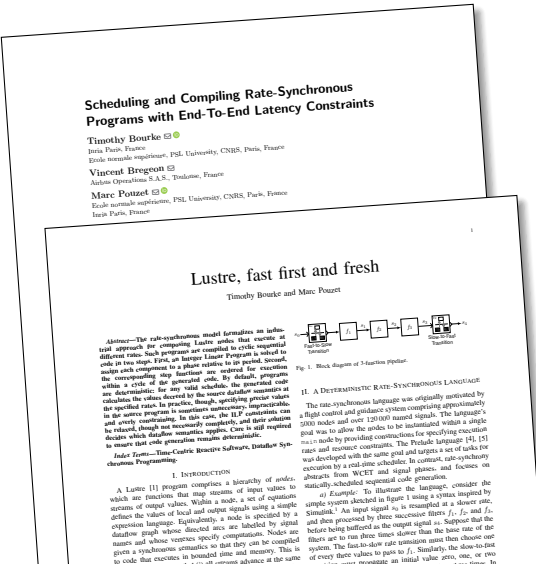
Embedded control software is often designed as a set of components that each repeatedly sample inputs, compute a transition function, and update outputs. Such components must be scheduled so as to share processor resources while respecting timing and communication requirements. Scheduling determines how data propagates along chains of components from sensor acquisitions, through successive computations, to corresponding actuator emissions. The end-to-end latencies of such chains are crucial to overall system performance.

We characterize and extend an approach for developing avionics software based on the synchronous-reactive languages Lustre [29] and Scade [13]. Our application model comprises (i) a set of components whose execution rates are specified as unit fractions ($1/n$) of a base rate, and (ii) a graph of data flow between components. The Worst-Case Execution Time (WCET) of each component must be less than the base period. This is a significant restriction, but one that is acceptable for safety-critical avionics applications. The implementation target is one or more sequential step functions called cyclically to, in turn, call individual component step functions. Data is exchanged by reading and writing static variables.

- Details of language and ILP encoding
- Encoding for end-to-end latency constraints
- Adapt clock-directed modular code generation
  [Biernacki, Colaço, Hamon, and Pouzet (2008): Clock-directed modular code generation for synchronous data-flow languages]
- Decide concomitance before ILP scheduling

ECRTS 2023 Article

More Information



Scheduling and Compiling Rate-Synchronous Programs with End-To-End Latency Constraints

Timothy Bourke
Inria Paris, France
École normale supérieure, PSL University, CNRS, Paris, France

Vincent Bregeon
Airbus Operations S.A.S., Toulouse, France

Marc Pouzet
École normale supérieure, PSL University, CNRS, Paris, France
Inria Paris, France

Lustre, fast first and fresh

Timothy Bourke and Marc Pouzet

ECRTS 2023 Article

- Details of language and ILP encoding
- Encoding for end-to-end latency constraints
- Adapt clock-directed modular code generation

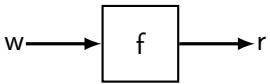
  [Biernacki, Colaço, Hamon, and Pouzet (2008): Clock-directed modular code generation for synchronous data-flow languages]
- Decide concomitance before ILP scheduling

TCRS 2024 article

- Decide concomitance during ILP scheduling
- Extra constraints to avoid cycles in sequencing
- The fast first convention to avoid cycles

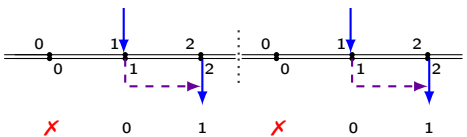
Direct Communications

```
r = f(w)
```

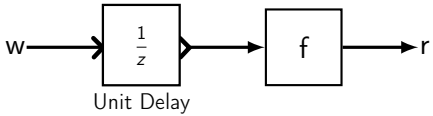


$\mathrm{p}{:}w < \mathrm{p}{:}r + \mathrm{cc}_{w,r}$

$$\begin{cases} \mathrm{p}{:}w \leq \mathrm{p}{:}r & \text{if } \mathrm{cc}_{w,r} = 1 \text{ (forward)} \\ \mathrm{p}{:}w < \mathrm{p}{:}r & \text{if } \mathrm{cc}_{w,r} = 0 \text{ (backward)} \end{cases}$$

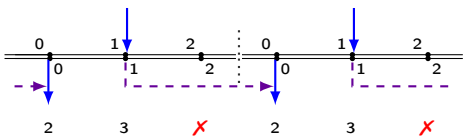



```
r = f(last w)
```



$\mathrm{p}{:}r + \mathrm{cc}_{w,r} \leq \mathrm{p}{:}w$

$$\begin{cases} \mathrm{p}{:}r < \mathrm{p}{:}w & \text{if } \mathrm{cc}_{w,r} = 1 \text{ (forward)} \\ \mathrm{p}{:}r \leq \mathrm{p}{:}w & \text{if } \mathrm{cc}_{w,r} = 0 \text{ (backward)} \end{cases}$$

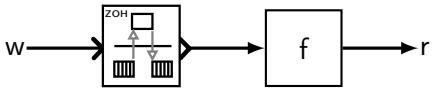


Rate Transitions

`r = f(w when (1 % 3))`

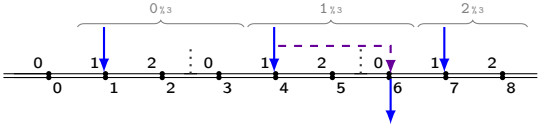
`(i % n)`: take value `i` of every `n`

`(? % n)`: take any of every `n` values



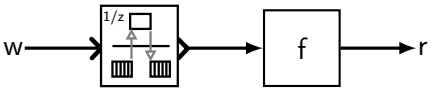
Fast-to-Slow Transition

$$i \cdot \mathrm{period}(w) + \mathrm{p}{:}w < \mathrm{p}{:}r + \mathrm{cc}_{w,r} \leq (i+1) \cdot \mathrm{period}(w) + \mathrm{p}{:}w$$



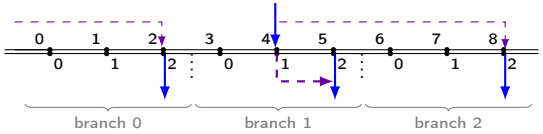
`r = f(current(w, (1 % 3)))`

`(i % n)`: `i` initial values, then repeat `n` times



Slow-to-Fast Transition

$$(i-1) \cdot \mathrm{period}(r) + \mathrm{p}{:}r \leq \mathrm{p}{:}w - \mathrm{cc}_{w,r} < i \cdot \mathrm{period}(r) + \mathrm{p}{:}r$$

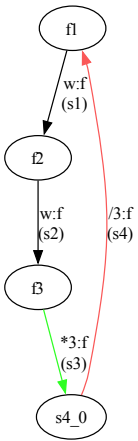


Cycles

```
s1 = f1(s0 when (? % 3), s4 when (? % 3));
s2 = f2(s1);
s3 = f3(s2);
s4 = current(s3, (? % 3));
```

Cycles

```
s1 = f1(s0 when (? % 3), s4 when (? % 3));
s2 = f2(s1);
s3 = f3(s2);
s4 = current(s3, (? % 3));
```



- Every flow has forward concomitance (`:f`)
- If all equations are scheduled in the same phase, then code generation fails because it cannot break the cycle.

Cycles: excluding via ILP Encodings

Classic encodings: see, e.g., [Baharev, Schichl, Neumaier, and Achterberg (2021): An Exact Method for the Minimum Feedback Arc Set Problem] §3

$$\min_{y} \sum_{j=1}^{n} \left(\sum_{k=1}^{j-1} c_{k,j} y_{k,j} + \sum_{\ell=j+1}^{n} c_{\ell,j}(1 - y_{j,\ell}) \right)$$

subject to

$$\begin{aligned} y_{i,j} + y_{j,k} - y_{i,k} &\leq 1, \quad 1 \leq i < j < k \leq n \\ -y_{i,j} - y_{j,k} + y_{i,k} &\leq 0, \quad 1 \leq i < j < k \leq n \\ y_{i,j} &= \{0, 1\}, \quad 1 \leq i < j \leq n. \end{aligned}$$

($c_{i,j} = 0$ if $(i,j) \notin E$ $\quad$ $y_{i,j} = 0$: i precedes j in ordering)

Cycles: excluding via ILP Encodings

Classic encodings: see, e.g., [Baharev, Schichl, Neumaier, and Achterberg (2021): An Exact Method for the Minimum Feedback Arc Set Problem] §3

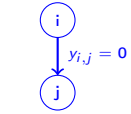
$$\min_{y} \sum_{j=1}^{n} \left(\sum_{k=1}^{j-1} c_{k,j} y_{k,j} + \sum_{\ell=j+1}^{n} c_{\ell,j} (1 - y_{j,\ell}) \right)$$

subject to

$$\begin{aligned} y_{i,j} + y_{j,k} - y_{i,k} &\leq 1, \quad 1 \leq i < j < k \leq n \\ -y_{i,j} - y_{j,k} + y_{i,k} &\leq 0, \quad 1 \leq i < j < k \leq n \\ y_{i,j} &= \{0,1\}, \quad 1 \leq i < j \leq n. \end{aligned}$$

($c_{i,j} = 0$ if $(i,j) \notin E$ $\quad$ $y_{i,j} = 0$: i precedes j in ordering)

- Encode linear ordering as a graph:



- $O(n^2)$ binaries and $O(n^3)$ constraints...
- branch-and-cut:

[Grötschel, Jünger, and Reinelt (1984): A Cutting Plane Algorithm for the Linear Ordering Problem]

Cycles: excluding via ILP Encodings

Classic encodings: see, e.g., [Baharev, Schichl, Neumaier, and Achterberg (2021): An Exact Method for the Minimum Feedback Arc Set Problem] §3

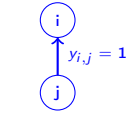
$$\min_{y} \sum_{j=1}^{n} \left(\sum_{k=1}^{j-1} c_{k,j} y_{k,j} + \sum_{\ell=j+1}^{n} c_{\ell,j} (1 - y_{j,\ell}) \right)$$

subject to

$$\begin{aligned} y_{i,j} + y_{j,k} - y_{i,k} &\leq 1, \quad 1 \leq i < j < k \leq n \\ -y_{i,j} - y_{j,k} + y_{i,k} &\leq 0, \quad 1 \leq i < j < k \leq n \\ y_{i,j} &= \{0,1\}, \quad 1 \leq i < j \leq n. \end{aligned}$$

($c_{i,j} = 0$ if $(i,j) \notin E$ $\quad$ $y_{i,j} = 0$: i precedes j in ordering)

- Encode linear ordering as a graph:



- $O(n^2)$ binaries and $O(n^3)$ constraints...
- branch-and-cut:

[Grötschel, Jünger, and Reinelt (1984): A Cutting Plane Algorithm for the Linear Ordering Problem]

Cycles: excluding via ILP Encodings

Classic encodings: see, e.g., [Baharev, Schichl, Neumaier, and Achterberg (2021): An Exact Method for the Minimum Feedback Arc Set Problem] §3

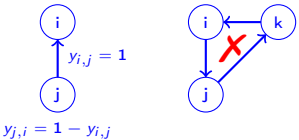
$$\min_y \sum_{j=1}^{n} \left(\sum_{k=1}^{j-1} c_{k,j} y_{k,j} + \sum_{\ell=j+1}^{n} c_{\ell,j}(1 - y_{j,\ell}) \right)$$

subject to

$$\begin{aligned} y_{i,j} + y_{j,k} - y_{i,k} &\leq 1, \quad 1 \leq i < j < k \leq n \\ -y_{i,j} - y_{j,k} + y_{i,k} &\leq 0, \quad 1 \leq i < j < k \leq n \\ y_{i,j} &= \{0,1\}, \quad 1 \leq i < j \leq n. \end{aligned}$$

($c_{i,j} = 0$ if $(i,j) \notin E$ $\quad$ $y_{i,j} = 0$: i precedes j in ordering)

- Encode linear ordering as a graph:



- $O(n^2)$ binaries and $O(n^3)$ constraints...
- branch-and-cut:

[Grötschel, Jünger, and Reinelt (1984): A Cutting Plane Algorithm for the Linear Ordering Problem]

Cycles: excluding via ILP Encodings

Classic encodings: see, e.g., [Baharev, Schichl, Neumaier, and Achterberg (2021): An Exact Method for the Minimum Feedback Arc Set Problem] §3

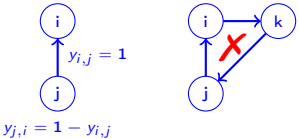
$$\min_y \sum_{j=1}^{n} \left(\sum_{k=1}^{j-1} c_{k,j} y_{k,j} + \sum_{\ell=j+1}^{n} c_{\ell,j}(1 - y_{j,\ell}) \right)$$

subject to

$$\begin{aligned} y_{i,j} + y_{j,k} - y_{i,k} &\leq 1, \quad 1 \leq i < j < k \leq n \\ -y_{i,j} - y_{j,k} + y_{i,k} &\leq 0, \quad 1 \leq i < j < k \leq n \\ y_{i,j} &= \{0,1\}, \quad 1 \leq i < j \leq n. \end{aligned}$$

($c_{i,j} = 0$ if $(i,j) \notin E$ $\quad$ $y_{i,j} = 0$: i precedes j in ordering)

- Encode linear ordering as a graph:



- $O(n^2)$ binaries and $O(n^3)$ constraints...
- branch-and-cut:

[Grötschel, Jünger, and Reinelt (1984): A Cutting Plane Algorithm for the Linear Ordering Problem]

Cycles: excluding via ILP Encodings

Classic encodings: see, e.g., [Baharev, Schichl, Neumaier, and Achterberg (2021): An Exact Method for the Minimum Feedback Arc Set Problem] §3

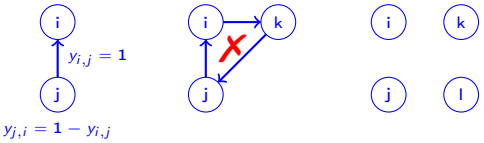
$$\min_y \sum_{j=1}^{n} \left(\sum_{k=1}^{j-1} c_{k,j} y_{k,j} + \sum_{\ell=j+1}^{n} c_{\ell,j} (1 - y_{j,\ell}) \right)$$

subject to

$$\begin{aligned} y_{i,j} + y_{j,k} - y_{i,k} &\leq 1, \quad 1 \leq i < j < k \leq n \\ -y_{i,j} - y_{j,k} + y_{i,k} &\leq 0, \quad 1 \leq i < j < k \leq n \\ y_{i,j} &= \{0,1\}, \quad 1 \leq i < j \leq n. \end{aligned}$$

($c_{i,j} = 0$ if $(i,j) \notin E$ $\quad$ $y_{i,j} = 0$: i precedes j in ordering)

- Encode linear ordering as a graph:



- $O(n^2)$ binaries and $O(n^3)$ constraints...
- branch-and-cut:

[Grötschel, Jünger, and Reinelt (1984): A Cutting Plane Algorithm for the Linear Ordering Problem]

Cycles: excluding via ILP Encodings

Classic encodings: see, e.g., [Baharev, Schichl, Neumaier, and Achterberg (2021): An Exact Method for the Minimum Feedback Arc Set Problem] §3

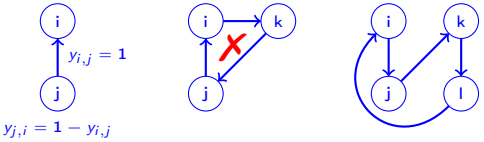
$$\min_y \sum_{j=1}^{n} \left(\sum_{k=1}^{j-1} c_{k,j} y_{k,j} + \sum_{\ell=j+1}^{n} c_{\ell,j}(1 - y_{j,\ell}) \right)$$

subject to

$$\begin{aligned} y_{i,j} + y_{j,k} - y_{i,k} &\leq 1, \quad 1 \leq i < j < k \leq n \\ -y_{i,j} - y_{j,k} + y_{i,k} &\leq 0, \quad 1 \leq i < j < k \leq n \\ y_{i,j} &= \{0,1\}, \quad 1 \leq i < j \leq n. \end{aligned}$$

($c_{i,j} = 0$ if $(i,j) \notin E$ $\quad$ $y_{i,j} = 0$: i precedes j in ordering)

- Encode linear ordering as a graph:



- $O(n^2)$ binaries and $O(n^3)$ constraints...
- branch-and-cut:

[Grötschel, Jünger, and Reinelt (1984): A Cutting Plane Algorithm for the Linear Ordering Problem]

Cycles: excluding via ILP Encodings

Classic encodings: see, e.g., [Baharev, Schichl, Neumaier, and Achterberg (2021): An Exact Method for the Minimum Feedback Arc Set Problem] §3

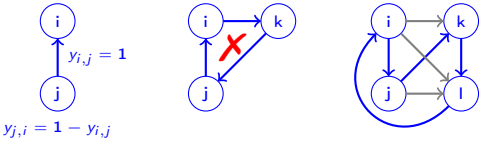
$$\min_y \sum_{j=1}^{n} \left(\sum_{k=1}^{j-1} c_{k,j} y_{k,j} + \sum_{\ell=j+1}^{n} c_{\ell,j}(1 - y_{j,\ell}) \right)$$

subject to

$$\begin{aligned} y_{i,j} + y_{j,k} - y_{i,k} &\leq 1, \quad 1 \leq i < j < k \leq n \\ -y_{i,j} - y_{j,k} + y_{i,k} &\leq 0, \quad 1 \leq i < j < k \leq n \\ y_{i,j} &= \{0,1\}, \quad 1 \leq i < j \leq n. \end{aligned}$$

($c_{i,j} = 0$ if $(i,j) \notin E$ $\quad$ $y_{i,j} = 0$: i precedes j in ordering)

- Encode linear ordering as a graph:



- $O(n^2)$ binaries and $O(n^3)$ constraints...
- branch-and-cut:

[Grötschel, Jünger, and Reinelt (1984): A Cutting Plane Algorithm for the Linear Ordering Problem]

Cycles: excluding via ILP Encodings

Classic encodings: see, e.g., [Baharev, Schichl, Neumaier, and Achterberg (2021): An Exact Method for the Minimum Feedback Arc Set Problem] §3

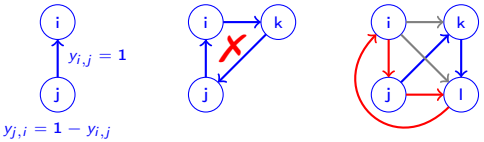
$$\min_y \sum_{j=1}^{n} \left(\sum_{k=1}^{j-1} c_{k,j} y_{k,j} + \sum_{\ell=j+1}^{n} c_{\ell,j}(1 - y_{j,\ell}) \right)$$

subject to

$$\begin{aligned} y_{i,j} + y_{j,k} - y_{i,k} &\leq 1, \quad 1 \leq i < j < k \leq n \\ -y_{i,j} - y_{j,k} + y_{i,k} &\leq 0, \quad 1 \leq i < j < k \leq n \\ y_{i,j} &= \{0,1\}, \quad 1 \leq i < j \leq n. \end{aligned}$$

($c_{i,j} = 0$ if $(i,j) \notin E$ $\quad$ $y_{i,j} = 0$: i precedes j in ordering)

- Encode linear ordering as a graph:



- $O(n^2)$ binaries and $O(n^3)$ constraints...
- branch-and-cut:

[Grötschel, Jünger, and Reinelt (1984): A Cutting Plane Algorithm for the Linear Ordering Problem]

Cycles: excluding via ILP Encodings

Classic encodings: see, e.g., [Baharev, Schichl, Neumaier, and Achterberg (2021): An Exact Method for the Minimum Feedback Arc Set Problem] §3

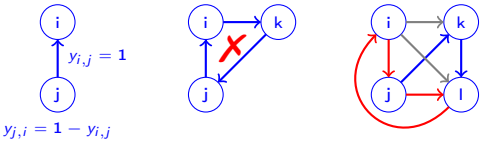
$$\min_y \sum_{j=1}^{n} \left(\sum_{k=1}^{j-1} c_{k,j} y_{k,j} + \sum_{\ell=j+1}^{n} c_{\ell,j}(1 - y_{j,\ell}) \right)$$

subject to

$$\begin{aligned} y_{i,j} + y_{j,k} - y_{i,k} &\leq 1, \quad 1 \leq i < j < k \leq n \\ -y_{i,j} - y_{j,k} + y_{i,k} &\leq 0, \quad 1 \leq i < j < k \leq n \\ y_{i,j} &= \{0,1\}, \quad 1 \leq i < j \leq n. \end{aligned}$$

($c_{i,j} = 0$ if $(i,j) \notin E$ $\quad$ $y_{i,j} = 0$: i precedes j in ordering)

- Encode linear ordering as a graph:



- $O(n^2)$ binaries and $O(n^3)$ constraints...
- branch-and-cut:

[Grötschel, Jünger, and Reinelt (1984): A Cutting Plane Algorithm for the Linear Ordering Problem]

Cycles: excluding via ILP Encodings

Classic encodings: see, e.g., [Baharev, Schichl, Neumaier, and Achterberg (2021): An Exact Method for the Minimum Feedback Arc Set Problem] §3

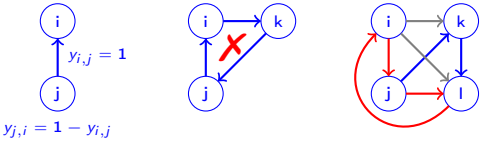
$$\min_y \sum_{j=1}^{n} \left(\sum_{k=1}^{j-1} c_{k,j} y_{k,j} + \sum_{\ell=j+1}^{n} c_{\ell,j}(1 - y_{j,\ell}) \right)$$

subject to

$$\begin{aligned} y_{i,j} + y_{j,k} - y_{i,k} &\leq 1, \quad 1 \leq i < j < k \leq n \\ -y_{i,j} - y_{j,k} + y_{i,k} &\leq 0, \quad 1 \leq i < j < k \leq n \\ y_{i,j} &= \{0,1\}, \quad 1 \leq i < j \leq n. \end{aligned}$$

($c_{i,j} = 0$ if $(i,j) \notin E$ $\quad$ $y_{i,j} = 0$: i precedes j in ordering)

- Encode linear ordering as a graph:



- $O(n^2)$ binaries and $O(n^3)$ constraints...
- branch-and-cut:

[Grötschel, Jünger, and Reinelt (1984): A Cutting Plane Algorithm for the Linear Ordering Problem]

Cycles: excluding via ILP Encodings

Classic encodings: see, e.g., [Baharev, Schichl, Neumaier, and Achterberg (2021): An Exact Method for the Minimum Feedback Arc Set Problem] §3

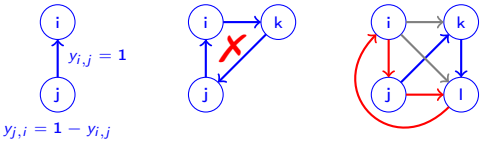
$$\min_y \sum_{j=1}^{n} \left(\sum_{k=1}^{j-1} c_{k,j} y_{k,j} + \sum_{\ell=j+1}^{n} c_{\ell,j}(1 - y_{j,\ell}) \right)$$

subject to

$$\begin{aligned} y_{i,j} + y_{j,k} - y_{i,k} &\le 1, \quad 1 \le i < j < k \le n \\ -y_{i,j} - y_{j,k} + y_{i,k} &\le 0, \quad 1 \le i < j < k \le n \\ y_{i,j} &= \{0,1\}, \quad 1 \le i < j \le n. \end{aligned}$$

($c_{i,j} = 0$ if $(i,j) \notin E$ $\quad$ $y_{i,j} = 0$: i precedes j in ordering)

- Encode linear ordering as a graph:



- $O(n^2)$ binaries and $O(n^3)$ constraints...
- branch-and-cut:

[Grötschel, Jünger, and Reinelt (1984): A Cutting Plane Algorithm for the Linear Ordering Problem]

$$\min_y \sum_{j=1}^{m} w_j y_j$$

$$\text{s.t.} \sum_{j=1}^{m} a_{ij} y_j \ge 1 \qquad \text{for each } i = 1, 2, \ldots, \ell$$

$$y_j \text{ is binary}$$

- $y_j = 1$: arc j is in the feedback arc set
- $a_{ij} = 1$: arc j participates in cycle i
- l elementary cycles, worst case $\approx O(2^m)$...
- branch-and-cut:

[Grötschel, Jünger, and Reinelt (2022): Comments on "An Exact Method for the Minimum Feedback Arc Set Problem"]

- cross fingers and enumerate?

[Johnson (1975): Finding All the Elementary Circuits of a Directed Graph]

Cycles: excluding via ILP Encodings

Classic encodings: see, e.g., [Baharev, Schichl, Neumaier, and Achterberg (2021): An Exact Method for the Minimum Feedback Arc Set Problem] §3

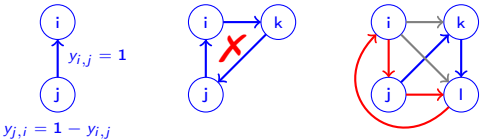
$$\min_y \sum_{j=1}^{n} \left(\sum_{k=1}^{j-1} c_{k,j} y_{k,j} + \sum_{\ell=j+1}^{n} c_{\ell,j}(1 - y_{j,\ell}) \right)$$

subject to

$$\begin{aligned} y_{i,j} + y_{j,k} - y_{i,k} &\le 1, \quad 1 \le i < j < k \le n \\ -y_{i,j} - y_{j,k} + y_{i,k} &\le 0, \quad 1 \le i < j < k \le n \\ y_{i,j} &= \{0,1\}, \quad 1 \le i < j \le n. \end{aligned}$$

($c_{i,j} = 0$ if $(i,j) \notin E$ $\quad$ $y_{i,j} = 0$: i precedes j in ordering)

- Encode linear ordering as a graph:



- $O(n^2)$ binaries and $O(n^3)$ constraints...
- branch-and-cut:

[Grötschel, Jünger, and Reinelt (1984): A Cutting Plane Algorithm for the Linear Ordering Problem]

$$\min_y \sum_{j=1}^{m} w_j y_j$$

$$\text{s.t.} \sum_{j=1}^{m} a_{ij} y_j \ge 1 \qquad \text{for each } i = 1, 2, \ldots, \ell$$

$$y_j \text{ is binary}$$

- $y_j = 1$: arc j is in the feedback arc set
- $a_{ij} = 1$: arc j participates in cycle i
- l elementary cycles, worst case $\approx O(2^m)$...
- branch-and-cut:

[Grötschel, Jünger, and Reinelt (2022): Comments on "An Exact Method for the Minimum Feedback Arc Set Problem"]

- cross fingers and enumerate?

[Johnson (1975): Finding All the Elementary Circuits of a Directed Graph]

Cycles: excluding via ILP Encodings

Classic encodings: see, e.g., [Baharev, Schichl, Neumaier, and Achterberg (2021): An Exact Method for the Minimum Feedback Arc Set Problem] §3

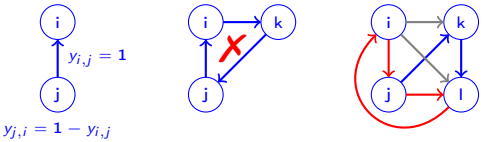
$$\min_y \sum_{j=1}^{n} \left(\sum_{k=1}^{j-1} c_{k,j} y_{k,j} + \sum_{\ell=j+1}^{n} c_{\ell,j}(1 - y_{j,\ell}) \right)$$

subject to

$$\begin{aligned} y_{i,j} + y_{j,k} - y_{i,k} &\le 1, \quad 1 \le i < j < k \le n \\ -y_{i,j} - y_{j,k} + y_{i,k} &\le 0, \quad 1 \le i < j < k \le n \\ y_{i,j} &= \{0,1\}, \quad 1 \le i < j \le n. \end{aligned}$$

($c_{i,j} = 0$ if $(i,j) \notin E$ $\quad$ $y_{i,j} = 0$: i precedes j in ordering)

- Encode linear ordering as a graph:



- $O(n^2)$ binaries and $O(n^3)$ constraints...
- branch-and-cut:

[Grötschel, Jünger, and Reinelt (1984): A Cutting Plane Algorithm for the Linear Ordering Problem]

$$\min_y \sum_{j=1}^{m} w_j y_j$$

$$\text{s.t.} \sum_{j=1}^{m} a_{ij} y_j \ge 1 \qquad \text{for each } i = 1, 2, \ldots, \ell$$

y_j is binary

- $y_j = 1$: arc j is in the feedback arc set
- $a_{ij} = 1$: arc j participates in cycle i
- l elementary cycles, worst case $\approx O(2^m)$...
- branch-and-cut:

[Grötschel, Jünger, and Reinelt (2022): Comments on "An Exact Method for the Minimum Feedback Arc Set Problem"]

- cross fingers and enumerate?

[Johnson (1975): Finding All the Elementary Circuits of a Directed Graph]

Cycles: excluding via ILP Encodings

Classic encodings: see, e.g., [Baharev, Schichl, Neumaier, and Achterberg (2021): An Exact Method for the Minimum Feedback Arc Set Problem] §3

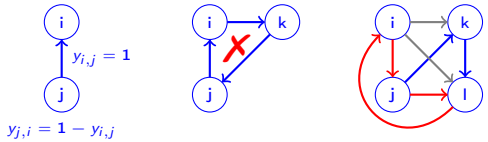
$$\min_y \sum_{j=1}^{n} \left(\sum_{k=1}^{j-1} c_{k,j} y_{k,j} + \sum_{\ell=j+1}^{n} c_{\ell,j}(1 - y_{j,\ell}) \right)$$

subject to

$$\begin{aligned} y_{i,j} + y_{j,k} - y_{i,k} &\le 1, \quad 1 \le i < j < k \le n \\ -y_{i,j} - y_{j,k} + y_{i,k} &\le 0, \quad 1 \le i < j < k \le n \\ y_{i,j} &= \{0,1\}, \quad 1 \le i < j \le n. \end{aligned}$$

($c_{i,j} = 0$ if $(i,j) \notin E$ $\quad$ $y_{i,j} = 0$: i precedes j in ordering)

- Encode linear ordering as a graph:



- $O(n^2)$ binaries and $O(n^3)$ constraints...
- branch-and-cut:

[Grötschel, Jünger, and Reinelt (1984): A Cutting Plane Algorithm for the Linear Ordering Problem]

$$\min_y \sum_{j=1}^{m} w_j y_j$$

$$\text{s.t.} \sum_{j=1}^{m} a_{ij} y_j \ge 1 \qquad \text{for each } i = 1, 2, \ldots, \ell$$

$$y_j \text{ is binary}$$

- $y_j = 1$: arc j is in the feedback arc set
- $a_{ij} = 1$: arc j participates in cycle i
- l elementary cycles, worst case $\approx O(2^m)$...
- branch-and-cut:

[Grötschel, Jünger, and Reinelt (2022): Comments on "An Exact Method for the Minimum Feedback Arc Set Problem"]

- cross fingers and enumerate?

[Johnson (1975): Finding All the Elementary Circuits of a Directed Graph]

Cycles: excluding via ILP Encodings

Classic encodings: see, e.g., [Baharev, Schichl, Neumaier, and Achterberg (2021): An Exact Method for the Minimum Feedback Arc Set Problem] §3

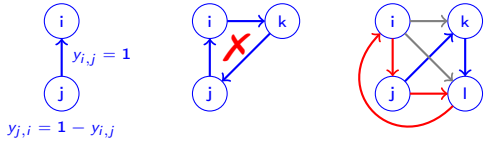
$$\min_y \sum_{j=1}^{n} \left(\sum_{k=1}^{j-1} c_{k,j} y_{k,j} + \sum_{\ell=j+1}^{n} c_{\ell,j}(1 - y_{j,\ell}) \right)$$

subject to

$$\begin{aligned} y_{i,j} + y_{j,k} - y_{i,k} &\le 1, & 1 \le i < j < k \le n \\ -y_{i,j} - y_{j,k} + y_{i,k} &\le 0, & 1 \le i < j < k \le n \\ y_{i,j} &= \{0,1\}, & 1 \le i < j \le n. \end{aligned}$$

($c_{i,j} = 0$ if $(i,j) \notin E$ $\quad$ $y_{i,j} = 0$: i precedes j in ordering)

- Encode linear ordering as a graph:



- $O(n^2)$ binaries and $O(n^3)$ constraints...
- branch-and-cut:

[Grötschel, Jünger, and Reinelt (1984): A Cutting Plane Algorithm for the Linear Ordering Problem]

$$\min_y \sum_{j=1}^{m} w_j y_j$$

$$\text{s.t.} \sum_{j=1}^{m} a_{ij} y_j \ge 1 \qquad \text{for each } i = 1, 2, \ldots, \ell$$

$$y_j \text{ is binary}$$

- $y_j = 1$: arc j is in the feedback arc set
- $a_{ij} = 1$: arc j participates in cycle i
- l elementary cycles, worst case $\approx O(2^m)$...
- branch-and-cut:

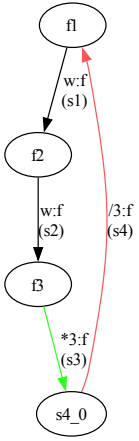
[Grötschel, Jünger, and Reinelt (2022): Comments on "An Exact Method for the Minimum Feedback Arc Set Problem"]

- cross fingers and enumerate?

[Johnson (1975): Finding All the Elementary Circuits of a Directed Graph]

Cycles: `--fast-first`

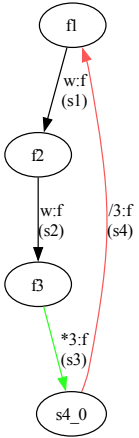
```
s1 = f1(s0 when (? % 3), s4 when (? % 3));
s2 = f2(s1);
s3 = f3(s2);
s4 = current(s3, (? % 3));
```



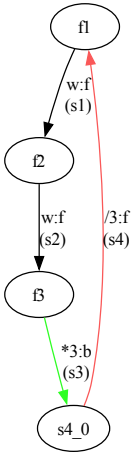
- Every flow has forward concomitance (`:f`)
- If all equations are scheduled in the same phase, then code generation fails because it cannot break the cycle.

Cycles: `--fast-first`

```
s1 = f1(s0 when (? % 3), s4 when (? % 3));
s2 = f2(s1);
s3 = f3(s2);
s4 = current(s3, (? % 3));
```



- Every flow has forward concomitance (`:f`)
- If all equations are scheduled in the same phase, then code generation fails because it cannot break the cycle.

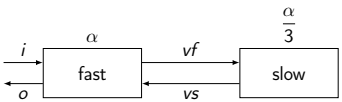


- With `--fast-first`, the flow between the last two equations, has backward concomitance (`:b`).
- Even if all equations are scheduled in the same phase, code generation succeeds because there is no cycle.

Plan

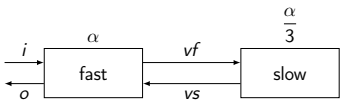
Causality, Scheduling, and Semantics

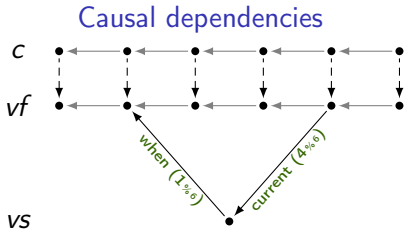
```
c = 0 fby (c + 1);
vf = current(vs, (4 % 6)) + c;
vs = vf when (1 % 6) + 5;
```



Causality, Scheduling, and Semantics

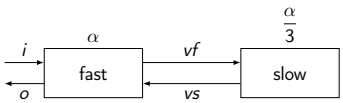
```
c = 0 fby (c + 1);
vf = current(vs, (4 % 6)) + c;
vs = vf when (1 % 6) + 5;
```



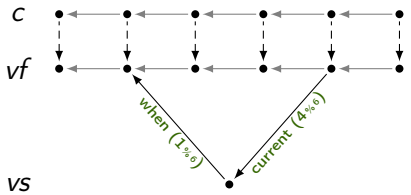


Causality, Scheduling, and Semantics

```
c = 0 fby (c + 1);
vf = current(vs, (4 % 6)) + c;
vs = vf when (1 % 6) + 5;
```



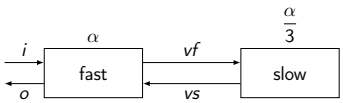
Causal dependencies



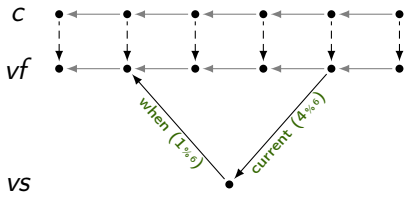
vf	0	1	2	3	10	11	12	13	14	15	28	29	···
c	0	1	2	3	4	5	6	7	8	9	10	11	···
vs	6						18						···

Causality, Scheduling, and Semantics

```
c = 0 fby (c + 1);
vf = current(vs, (4 % 6)) + c;
vs = vf when (1 % 6) + 5;
```



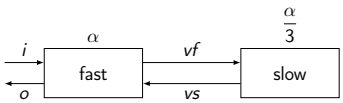
Causal dependencies



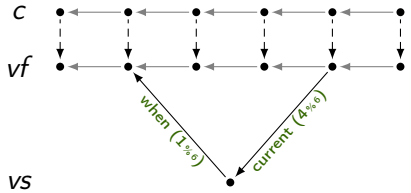
vf	0	1	2	3	10	11	12	13	14	15	28	29	···
c	0	1	2	3	4	5	6	7	8	9	10	11	···
vs	6						18						···

Causality, Scheduling, and Semantics

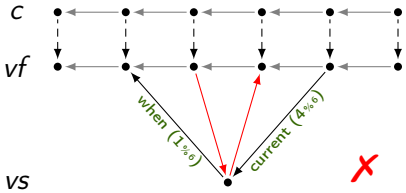
```
c = 0 fby (c + 1);
vf = current(vs, (4 % 6)) + c;
vs = vf when (1 % 6) + 5;
```



Causal dependencies



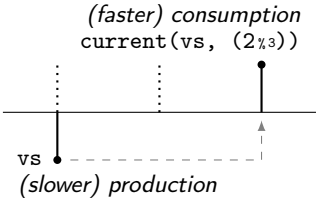
Scheduling dependencies



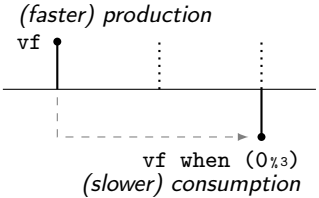
vf	0	1	2	3	10	11	12	13	14	15	28	29	···
c	0	1	2	3	4	5	6	7	8	9	10	11	···
vs	6						18						···

Adding equations to relax constraints/add buffering

Hold slow around fast reads

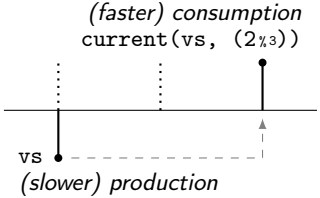


Hold fast around fast writes

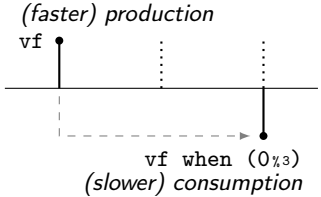


Adding equations to relax constraints/add buffering

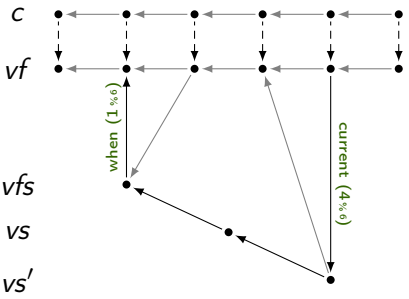
Hold slow around fast reads



Hold fast around fast writes




```
c = 0 fby (c + 1);
vf = current(vs', (4 % 6)) + c;
vfs = vf when (1 % 6)
vs = vfs + 5;
vs' = vs
```



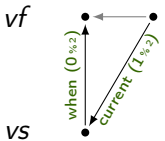
Causality: 1

- Which programs are valid? I.e., which have a semantics?
- Consider causality across the least common multiple of periods.
- Implicit dependencies to past elements on same flow.

Causality: 1

- Which programs are valid? I.e., which have a semantics?
- Consider causality across the least common multiple of periods.
- Implicit dependencies to past elements on same flow.

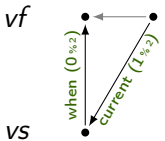
```
vf = current(vs, (1%2));
vs = vf when (0%2);
```



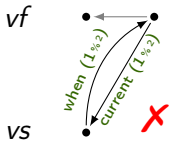
Causality: 1

- Which programs are valid? I.e., which have a semantics?
- Consider causality across the least common multiple of periods.
- Implicit dependencies to past elements on same flow.

```
vf = current(vs, (1%2));
vs = vf when (0%2);
```



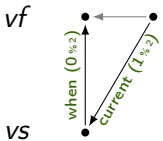

```
vf = current(vs, (1%2));
vs = vf when (1%2);
```



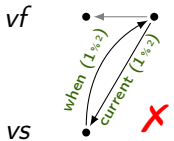
Causality: 1

- Which programs are valid? I.e., which have a semantics?
- Consider causality across the least common multiple of periods.
- Implicit dependencies to past elements on same flow.

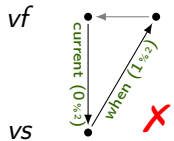
```
vf = current(vs, (1%2));
vs = vf when (0%2);
```




```
vf = current(vs, (1%2));
vs = vf when (1%2);
```



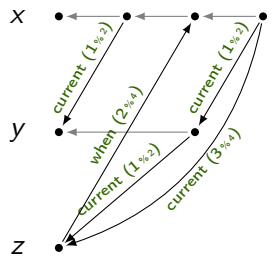

```
vf = current(vs, (0%2));
vs = vf when (1%2);
```



Causality: 2

Causality relations between `x` $:: \alpha$, `y` $:: \frac{\alpha}{2}$, and `z` $:: \frac{\alpha}{4}$.

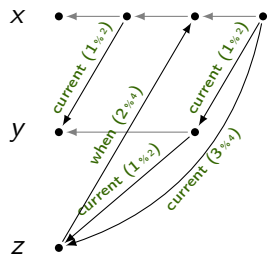
```
x = current(y, (1 % 2))
     + current(z, (3 % 4)) + 2;

y = current(z, (1 % 2)) + 20;

z = x when (2 % 4) + 200;
```



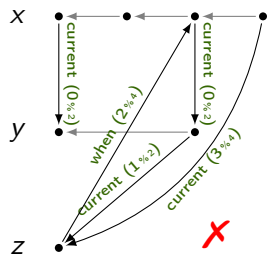
Causality: 2

Causality relations between `x :: α`, `y :: α/2`, and `z :: α/4`.

```
x = current(y, (1 % 2))
     + current(z, (3 % 4)) + 2;

y = current(z, (1 % 2)) + 20;

z = x when (2 % 4) + 200;
```

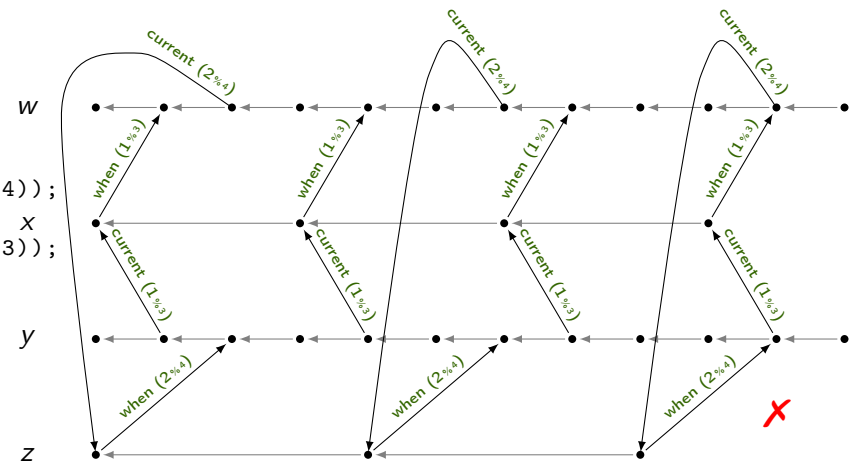



```
x = current(y, (0 % 2))
     + current(z, (3 % 4)) + 2;

y = current(z, (1 % 2)) + 20;

z = x when (2 % 4) + 200;
```



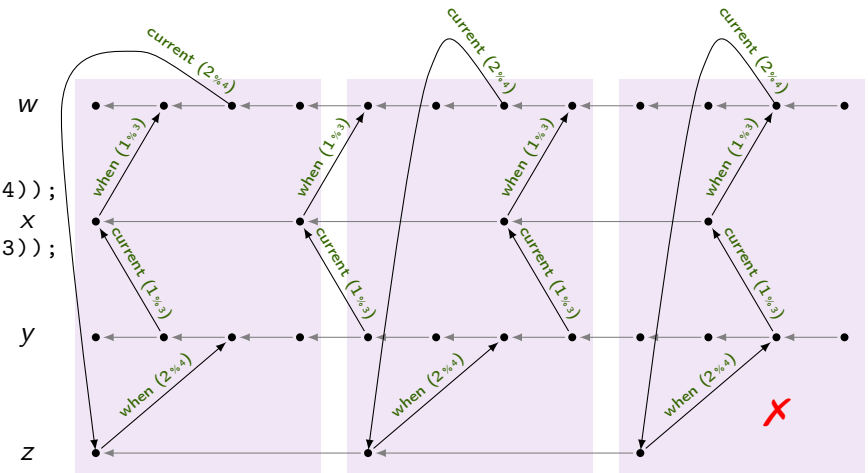
Causality: non-harmonic rates with ‘shifting’

```
w = current(z, (2 % 4));
x = w when (1 % 3);
y = current(x, (1 % 3));
z = y when (2 % 4);
```



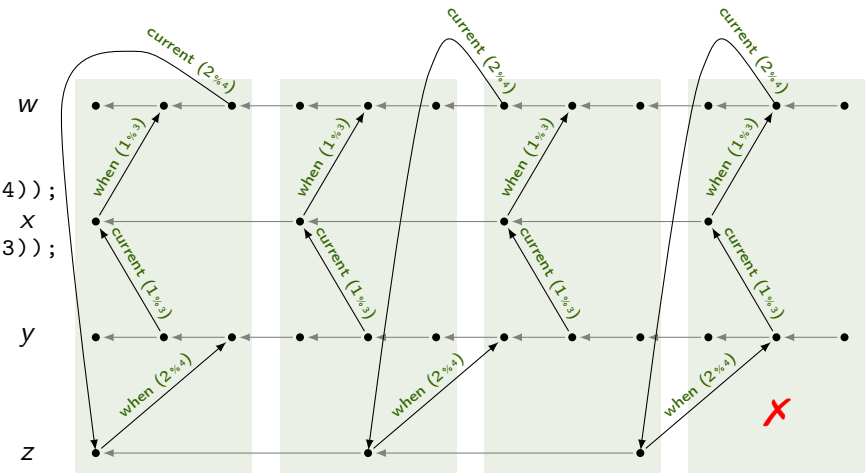
Causality: non-harmonic rates with 'shifting'

```
w = current(z, (2 % 4));
x = w when (1 % 3);
y = current(x, (1 % 3));
z = y when (2 % 4);
```



Causality: non-harmonic rates with ‘shifting’

```
w = current(z, (2 % 4));
x = w when (1 % 3);
y = current(x, (1 % 3));
z = y when (2 % 4);
```



Plan

Multi-threaded code generation: ongoing experiments

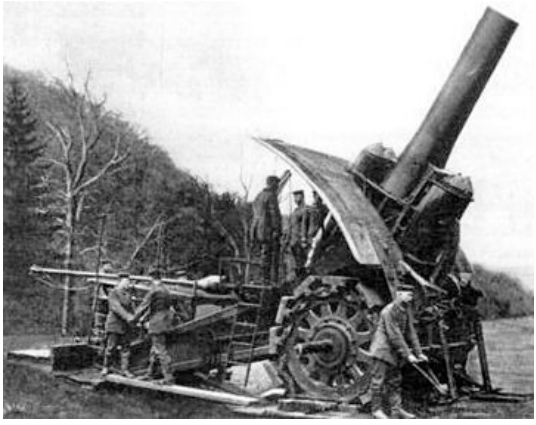
Why not?...

1. Generate ILP: equation $\mapsto$ thread & phase
2. Forbid inter-thread communication within a cycle

Multi-threaded code generation: ongoing experiments

Why not?...

1. Generate ILP: equation $\mapsto$ thread & phase
2. Forbid inter-thread communication within a cycle



"42-cm M-Gerät 14 Kurze Marinekanone L/12"

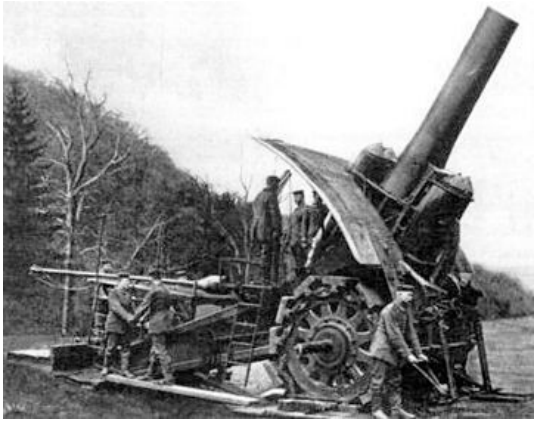
Multi-threaded code generation: ongoing experiments

Why not?...

1. Generate ILP: equation $\mapsto$ thread & phase
2. Forbid inter-thread communication within a cycle

...because

- the number of constraints explodes and the ILP solver may not be able to find a solution
- delayed communications may accumulate and increase end-to-end latency



"42-cm M-Gerät 14 Kurze Marinekanone L/12"

Threads and phases

Source program: `w = e;  r = f(w);`

$(t_w \neq t_r) \wedge (p_w = p_r)$: extra synchronization required

thread 1	thread 2
...	...
if (c % 2 == 0) { w = e; sem_post(wok); }	if (c % 2 == 0) { sem_wait(wok); r = f(w); }
...	...

Threads and phases

Source program: `w = e;  r = f(w);`

$(t_w \neq t_r) \wedge (p_w = p_r)$: extra synchronization required

thread 1	thread 2
...	...
if (c % 2 == 0) { w = e; sem_post(wok); }	if (c % 2 == 0) { sem_wait(wok); r = f(w); }
...	...

$(p_w \neq p_r)$: no race condition

thread 1	thread 2
...	...
if (c % 2 == 0) { w = e; }	if (c % 2 == 1) { r = f(w); }
...	...

Threads and phases

Source program: `w = e;  r = f(w);`

$(t_w \neq t_r) \wedge (p_w = p_r)$: extra synchronization required

thread 1	thread 2
...	...
if (c % 2 == 0) { w = e; sem_post(wok); }	if (c % 2 == 0) { sem_wait(wok); r = f(w); }
...	...

$(p_w \neq p_r)$: no race condition

thread 1	thread 2
...	...
if (c % 2 == 0) { w = e; }	if (c % 2 == 1) { r = f(w); }
...	...

for now, require: $t_w = t_r \vee p_w \neq p_r$ (may not be possible)

Constraints: same thread or different phase

require: $t_w = t_r \vee p_w \neq p_r$

- Try a completely boolean encoding using phase/thread weights?
- `w :: 1/4`, `r :: 1/12`, `--nthreads 2`

$$
\begin{aligned}
\texttt{th:0:ph:0:}w + \texttt{th:1:ph:0:}r + \texttt{th:1:ph:4:}r + \texttt{th:1:ph:8:}r &\leq 1 \\
\texttt{th:0:ph:1:}w + \texttt{th:1:ph:1:}r + \texttt{th:1:ph:5:}r + \texttt{th:1:ph:9:}r &\leq 1 \\
\texttt{th:0:ph:2:}w + \texttt{th:1:ph:2:}r + \texttt{th:1:ph:6:}r + \texttt{th:1:ph:10:}r &\leq 1 \\
&\vdots \\
\texttt{th:1:ph:0:}w + \texttt{th:0:ph:0:}r + \texttt{th:0:ph:4:}r + \texttt{th:0:ph:8:}r &\leq 1 \\
\texttt{th:1:ph:1:}w + \texttt{th:0:ph:1:}r + \texttt{th:0:ph:5:}r + \texttt{th:0:ph:9:}r &\leq 1 \\
\texttt{th:1:ph:2:}w + \texttt{th:0:ph:2:}r + \texttt{th:0:ph:6:}r + \texttt{th:0:ph:10:}r &\leq 1
\end{aligned}
$$

- Not very linear...

Resource Constraints

```
resource cpu : int

node f1(x : int) returns (y : int) requires (cpu = 5);
node f2(x : int) returns (y : int) requires (cpu = 2);
node f3(x : int) returns (y : int) requires (cpu = 2);
```

```
node main(s0 : int) returns (s4 : int)
let
  s1 = f1(s0 when (0 % 3));
  s2 = f2(s1);
  s3 = f3(s2);
  s4 = current(s3, (2 % 3));

  resource balance cpu;
tel
```

Existing encoding: per cycle

```
pw.def0.f1: pw.ph.0.f1 + pw.ph.1.f1 + pw.ph.2.f1 = 1
pw.def1.f1: -1 p.f1 + 2 pw.ph.2.f1 + pw.ph.1.f1  = 0
...
rsum.ph.0.cpu: rsum.ph.0.cpu - 2 pw.ph.0.f3 - 2 pw.ph.0.f2 - 5 pw.ph.0.f1 = 0
rsum.ph.1.cpu: rsum.ph.1.cpu - 2 pw.ph.1.f3 - 2 pw.ph.1.f2 - 5 pw.ph.1.f1 = 0
rsum.ph.2.cpu: rsum.ph.2.cpu - 2 pw.ph.2.f3 - 2 pw.ph.2.f2 - 5 pw.ph.2.f1 = 0
```

Resource Constraints

New possibility: `per thread per cycle`

```
...
tw.def1.thread.0: tw.1.thread.0 - thread.0 = 0
tw.def0.thread.0: tw.0.thread.0 + tw.1.thread.0 = 1
...
pw.def0.f1: pw.th.0.ph.0.f1 + pw.th.0.ph.1.f1 + pw.th.0.ph.2.f1
            + pw.th.1.ph.0.f1 + pw.th.1.ph.1.f1 + pw.th.1.ph.2.f1 = 1
pw.def1.f1: -1 p.f1 + 5 pw.th.1.ph.2.f1 + 4 pw.th.1.ph.1.f1
            + 3 pw.th.1.ph.0.f1 + 2 pw.th.0.ph.2.f1 + pw.th.0.ph.1.f1
            - 3 thread.0  = 0
...
rsum.th.0.ph.0.cpu: rsum.th.0.ph.0.cpu - 2 pw.th.0.ph.0.f3
                    - 2 pw.th.0.ph.0.f2 - 5 pw.th.0.ph.0.f1 = 0
rsum.th.0.ph.1.cpu: rsum.th.0.ph.1.cpu - 2 pw.th.0.ph.1.f3
                    - 2 pw.th.0.ph.1.f2 - 5 pw.th.0.ph.1.f1 = 0
rsum.th.1.ph.0.cpu: rsum.th.1.ph.0.cpu - 2 pw.th.1.ph.0.f3
                    - 2 pw.th.1.ph.0.f2 - 5 pw.th.1.ph.0.f1 = 0
...
```

Preliminary Experiments on largest Airbus case-study

- Oct. 2024: It does not work
  - » cplex: runs for days until out-of-memory
  - » `set mip strategy file 3`: runs for longer until dying

Preliminary Experiments on largest Airbus case-study

- Oct. 2024: It does not work
  - » cplex: runs for days until out-of-memory
  - » `set mip strategy file 3`: runs for longer until dying
- Nov. 2024: It works a little bit
  - » Schedule on one thread and generate a *MIP start file*
  - » Scheduling on two cores then starts from a valid solution, which the optimizer can iteratively improve.
  - » Solution after ≈35 hours

Preliminary Experiments on largest Airbus case-study

- Oct. 2024: It does not work
  - » cplex: runs for days until out-of-memory
  - » `set mip strategy file 3`: runs for longer until dying
- Nov. 2024: It works a little bit
  - » Schedule on one thread and generate a *MIP start file*
  - » Scheduling on two cores then starts from a valid solution, which the optimizer can iteratively improve.
  - » Solution after ≈35 hours
- Mar. 2025: "Boolean" encoding for same-thread or different-phase
  - » cplex: 2 hours
  - » cp-sat (with L. Sylvestre): 20 minutes
- Ongoing: Pure SAT encoding (with L. Sylvestre)
  - » Translate pseudo-boolean constraints following Eén and Sörensson (suggested by K. Claessen and M. Sheeran)
  - » Re-encode end-to-end latency constraints

~~ thread 0 ~~

hyperphase	equations	DGO01_A	DGO02_A	DGO03_A	DGO04_A	DGO05_A	DGO06_A	DGO08_A	cpu_A
0	679	7	3	4	4	6	3	4	1315251 (l:+519 g:+11164)
1	289	5	2	12	5	2	5	0	1316339 (l:+1607 g:+12252)
2	393	8	1	4	4	6	0	0	1316014 (l:+1282 g:+11927)
3	272	5	2	12	5	2	0	0	1313297 (l:-1435 g:+9210)
4	378	8	0	4	4	6	0	0	1315481 (l:+749 g:+11394)
5	266	5	2	12	5	2	0	0	1311635 (l:-3097 g:+7548)
6	416	7	1	4	5	6	4	0	1315593 (l:+861 g:+11506)
7	272	5	2	12	5	2	0	0	1316115 (l:+1383 g:+12028)
8	443	7	3	4	4	6	3	0	1315095 (l:+363 g:+11008)
9	276	5	2	12	5	2	5	0	1316585 (l:+1853 g:+12498)
10	391	7	3	4	4	6	0	0	1315298 (l:+566 g:+11211)
11	270	5	2	12	5	2	0	0	1313081 (l:-1651 g:+8994)
12	377	7	0	4	4	6	0	1	1315012 (l:+280 g:+10925)
13	266	5	2	12	5	2	0	0	1316618 (l:+1886 g:+12531)
14	400	7	2	4	5	6	4	0	1312386 (l:-2346 g:+8299)
15	275	5	2	12	5	2	0	3	1315161 (l:+429 g:+11074)
16	440	7	5	4	4	6	3	4	1314586 (l:-146 g:+10499)
17	275	5	2	12	5	2	5	0	1316550 (l:+1818 g:+12463)
18	391	7	1	4	4	6	0	0	1314626 (l:-106 g:+10539)
19	268	5	2	12	5	2	0	0	1311494 (l:-3238 g:+7407)
20	372	7	0	4	4	6	0	0	1315077 (l:+345 g:+10990)
21	265	5	2	12	5	2	0	0	1315509 (l:+777 g:+11422)
22	401	7	1	4	5	6	4	6	1312925 (l:-1807 g:+8838)
23	274	5	2	12	5	2	0	0	1313841 (l:-891 g:+9754)

cpu_A

1316618 (avg +0%)

1311494 (avg -0%)

012345678901234567890123

~~ thread 1 ~~

hyperphase	equations	DGO01_A	DGO02_A	DGO03_A	DGO04_A	DGO05_A	DGO06_A	DGO08_A	cpu_A
0	835	1	10	9	9	7	1	5	1316510 (l:+23068 g:+12423)
1	651	5	10	1	8	11	0	0	1314275 (l:+20833 g:+10188)
2	441	1	3	4	3	5	6	7	1312479 (l:+19037 g:+8392)
3	416	3	11	0	8	11	0	0	1313198 (l:+19756 g:+9111)
4	392	1	3	4	1	4	0	1	1314469 (l:+21027 g:+10382)
5	353	4	11	1	8	10	0	0	1297942 (l:+4500 g:-6145)
6	329	1	5	1	0	2	2	8	1315944 (l:+22502 g:+11857)
7	272	3	11	1	5	1	0	0	1311549 (l:+18107 g:+7462)
8	488	1	5	6	8	5	2	2	1313956 (l:+20514 g:+9869)
9	378	2	9	1	5	11	0	6	1309484 (l:+16042 g:+5397)
10	338	1	1	4	3	5	6	3	1315046 (l:+21604 g:+10959)
11	336	2	11	1	8	11	0	0	1297891 (l:+4449 g:-6196)
12	362	1	1	4	1	4	0	6	1314561 (l:+21119 g:+10474)
13	320	5	2	1	8	8	1	0	1257019 (l:-36423 g:-47068)
14	325	2	3	1	0	1	2	6	1314612 (l:+21170 g:+10525)
15	222	3	9	0	5	1	0	0	1181634 (l:-111808 g:-122453)
16	483	1	6	4	8	5	2	3	1312282 (l:+18840 g:+8195)
17	329	2	10	1	5	11	0	2	1309521 (l:+16079 g:+5434)
18	332	1	1	4	3	5	6	0	1309347 (l:+15905 g:+5260)
19	285	2	7	0	8	11	0	0	1206311 (l:-87131 g:-97776)
20	354	1	1	4	1	4	1	0	1305983 (l:+12541 g:+1896)
21	290	4	2	1	8	8	0	0	1237447 (l:-55995 g:-66640)
22	320	1	3	1	1	1	2	0	1306864 (l:+13422 g:+2777)
23	230	3	9	0	5	1	0	3	1254287 (l:-39155 g:-49800)

cpu_A

1316510 (avg +2%)

1181634 (avg -9%)

012345678901234567890123

Graph clustering prior to ILP

- Cluster flowgraph using *metis*[1] from Uni. Minnesota.
- Assign all equations in the same cluster to a single thread.
- Many partitions: not very effective at reducing solve time
- Few partitions: seems to preclude finding a valid schedule
  - » E.g., with four partitions, we can rapidly test all combinations for feasability: 1100, 0110, 1010
- Cluster in a different way? Feedback from ILP solver?

[1] https://github.com/karypislab/metis

WiP: Inter-thread communications for fast equations

- Same thread or different phase: only possible if $1/n < 1$.
- Prevents splitting writer/reader pairs when either is on the base clock

WiP: Inter-thread communications for fast equations

- Same thread or different phase: only possible if $1/n < 1$.
- Prevents splitting writer/reader pairs when either is on the base clock
- Current work: add a synchronization barrier

thread 1	thread 2
w0 = e0;	...
...	
sync();	sync();
	r0 = f(w0);
if (c % 2 == 0) { w1 = e1; }	if (c % 2 == 1) { r1 = f(w1); }
...	...

WiP: Inter-thread communications for fast equations

- Same thread or different phase: only possible if $1/n < 1$.
- Prevents splitting writer/reader pairs when either is on the base clock
- Current work: add a synchronization barrier

thread 1	thread 2
w0 = e0;	...
...	
sync();	sync();
	r0 = f(w0);
if (c % 2 == 0) { w1 = e1; }	if (c % 2 == 1) { r1 = f(w1); }
...	...

- Add (yet more) 0-1 variables to indicate before/after barrier.
- Add constraints for "same thread or different phase/side"
- Need to balance resources used before the barrier to minimize waiting.

Plan

Conclusion

- Block diagram language with deterministic stream semantics
- Scheduled using constraint solvers (cplex or cp-sat)
  - » Resource constraints and balancing
  - » End-to-end latency constraints
  - » Schedules have differing resource use, but calculate the same input/output function
- Prototype compiler with basic code generation.
- Tested on Airbus example with 5000 nodes (compiles in approx. 45 minutes).

References I

- Baharev, A., H. Schichl, A. Neumaier, and T. Achterberg (Apr. 2021). "An Exact Method for the Minimum Feedback Arc Set Problem". In: *ACM J. Experimental Algorithmics* 26.1, article 4.
- Biernacki, D., J.-L. Colaço, G. Hamon, and M. Pouzet (June 2008). "Clock-directed modular code generation for synchronous data-flow languages". In: *Proc. 9th ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008)*. Tucson, AZ, USA: ACM Press, pp. 121–130.
- Cohen, A., M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet (Jan. 2006). "N-Synchronous Kahn networks: a relaxed model of synchrony for real-time systems". In: *Proc. 33rd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 2006)*. Charleston, SC, USA: ACM Press, pp. 180–193.
- Cohen, A., L. Mandel, F. Plateau, and M. Pouzet (Dec. 2008). "Abstraction of Clocks in Synchronous Data-flow Systems". In: *Proc. 6th Asian Symp. Programming Languages and Systems (APLAS 2008)*. Ed. by G. Ramalingam. Vol. 5356. LNCS. Bangalore, India, pp. 237–254.
- Curic, A. (Sept. 2005). "Implementing Lustre Programs on Distributed Platforms with Real-Time Constraints". PhD thesis. Grenoble, France: Université Joseph Fourier.

References II

- Feiertag, N., K. Richter, J. Nordlander, and J. Jonsson (Nov. 2008). "A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics". In: *Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS 2008, co-located with RTSS 2008)*. Barcelona, Spain.
- Forget, J., F. Boniol, D. Lesens, and C. Pagetti (Dec. 2008). "A Multi-Periodic Synchronous Data-Flow Language". In: *Proc. 11th IEEE High Assurance Systems Engineering Symposium (HASE 2008)*. Nanjing, China, pp. 251–260.
- — (Mar. 2010). "A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems". In: *Proc. 25th ACM Symp. Applied Computing (SAC'10)*. Ed. by S. Y. Shin, S. Ossowski, M. Schumacher, M. J. Palakal, and C.-C. Hung. Sierre, Switzerland, pp. 527–534.
- Girault, A., C. Prévot, S. Quinton, R. Henia, and N. Sordon (Nov. 2018). "Improving and Estimating the Precision of Bounds on the Worst-Case Latency of Task Chains". In: *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems* 37.11, pp. 2578–2589.
- Grötschel, M., M. Jünger, and G. Reinelt (Nov. 1984). "A Cutting Plane Algorithm for the Linear Ordering Problem". In: *Operations Research* 32.6, pp. 1195–1220.

References III

- Grötschel, M., M. Jünger, and G. Reinelt (July 2022). "Comments on "An Exact Method for the Minimum Feedback Arc Set Problem"". In: *ACM J. Experimental Algorithmics* 27.1, article 3.
- Iooss, G., M. Pouzet, A. Cohen, D. Potop-Butucaru, J. Souyris, V. Bregeon, and P. Baufreton (Mar. 2020). "1-Synchronous Programming of Large Scale, Multi-Periodic Real-Time Applications with Functional Degrees of Freedom". preprint.
- Johnson, D. B. (Mar. 1975). "Finding All the Elementary Circuits of a Directed Graph". In: *SIAM J. Computing* 4.1, pp. 77–84.
- Mandel, L., F. Plateau, and M. Pouzet (June 2010). "Lucy-n: a n-Synchronous extension of Lustre". In: *Proc. 10th Int. Conf. on Mathematics of Program Construction (MPC 2010)*. Ed. by C. Bolduc, J. Desharnais, and B. Ktari. Vol. 6120. LNCS. Québec City, Canada, pp. 288–309.
- Pagetti, C., J. Forget, F. Boniol, M. Cordovilla, and D. Lesens (Sept. 2011). "Multi-task implementation of multi-periodic synchronous programs". In: *Discrete Event Dynamic Systems* 21.3, pp. 307–338.

References IV

- Pagetti, C., D. Saussié, R. Gratia, E. Noulard, and P. Siron (Apr. 2014). "The ROSACE Case Study: From Simulink Specification to Multi/Many-Core Execution". In: *20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2014)*. Berlin, Germany, pp. 309–318.
- Pouzet, M. (Apr. 2006). *Lucid Synchrone, v. 3. Tutorial and reference manual*. Université Paris-Sud.
- Smarandache, I. M., T. Gautier, and P. Le Guernic (Sept. 1999). "Validation of Mixed Signal-Alpha Real-Time Systems through Affine Calculus on Clock Synchronisation Constraints". In: *Proc. World Congress on Formal Methods in the Development of Computing Systems (FM'99)*. Ed. by J. M. Wing, J. Woodcock, and J. Davies. Vol. 1709. LNCS. Toulouse, France, pp. 1364–1383.
- Wyss, R., F. Boniol, J. Forget, and C. Pagetti (Dec. 2012). "A Synchronous Language with Partial Delay Specification for Real-Time Systems Programming". In: *Proc. 10th Asian Symp. Programming Languages and Systems (APLAS 2012)*. Ed. by R. Jhala and A. Igarashi. Vol. 7705. LNCS. Kyoto, Japan, pp. 223–238.