WIP: An attempt at multithreading via integer linear programming for rate-synchronous Lustre

Timothy Bourke

Inria Paris
École normale supérieure, PSL University

SYNCHRON 2024 · Bamberg, Germany

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 via sequences of cyclic tasks,
  write to actuators via the bus.

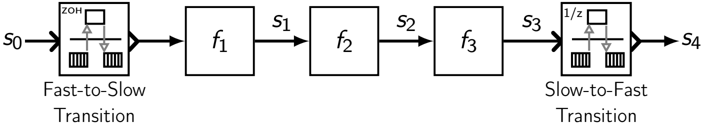
Context

- 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 via sequences of cyclic tasks,
  write to actuators via the bus.

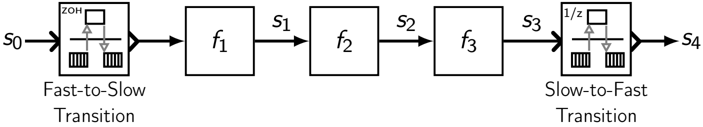
Airbus project "All-in-Lustre"

- *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.

A Simple Example

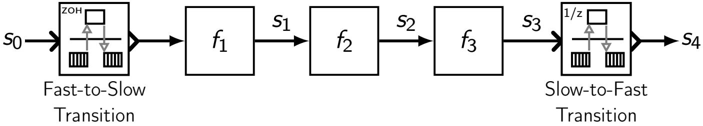


A Simple Example

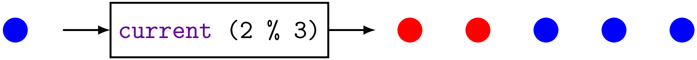



```
s1 = f1(s0 when (0 % 3));
s2 = f2(s1);
s3 = f3(s2);
s4 = current(s3, (2 % 3));
```

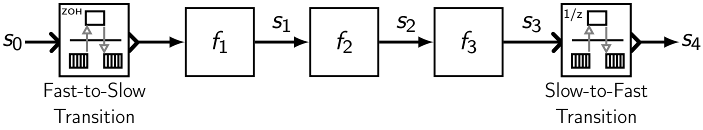
A Simple Example




```
s1 = f1(s0 when (0 % 3));
s2 = f2(s1);
s3 = f3(s2);
s4 = current(s3, (2 % 3));
```

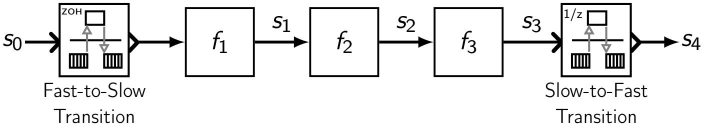


A Simple Example




```
node main(s0 : int) returns (s4 : int)
var s1, s2 : int :: 1/3;
    s3 : int :: 1/3 last = 0;
let
  s1 = f1(s0 when (0 % 3));
  s2 = f2(s1);
  s3 = f3(s2);
  s4 = current(s3, (2 % 3));

  latency_chain forward <= 1 (s1 -> s2 -> s3);

tel
```

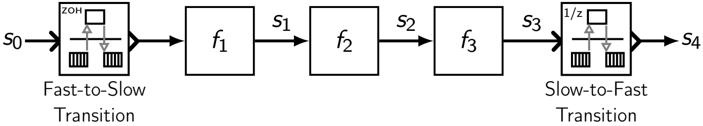
A Simple Example




```
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)
var s1, s2 : int :: 1/3;
    s3 : int :: 1/3 last = 0;
let
  s1 = f1(s0 when (0 % 3));
  s2 = f2(s1);
  s3 = f3(s2);
  s4 = current(s3, (2 % 3));

  latency_chain forward <= 1 (s1 -> s2 -> s3);

tel
```

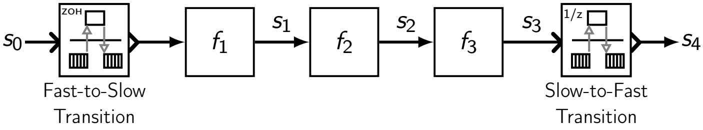
A Simple Example




```
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)
var s1, s2 : int :: 1/3;
    s3 : int :: 1/3 last = 0;
let
  s1 = f1(s0 when (0 % 3));
  s2 = f2(s1);
  s3 = f3(s2);
  s4 = current(s3, (2 % 3));

  latency_chain forward <= 1 (s1 -> s2 -> s3);
  resource balance cpu;
tel
```

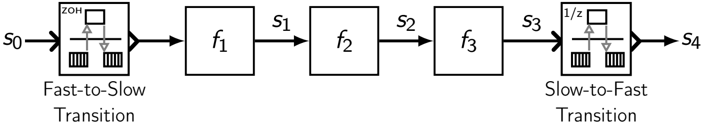
Fresh: non-deterministic sample choices




```
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)
var s1, s2 : int :: 1/3;
    s3 : int :: 1/3 last = 0;
let
  s1 = f1(s0 when (0 % 3));
  s2 = f2(s1);
  s3 = f3(s2);
  s4 = current(s3, (2 % 3));

  latency_chain forward <= 1 (s1 -> s2 -> s3);
  resource balance cpu;
tel
```

Fresh: non-deterministic sample choices




```
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)
var s1, s2 : int :: 1/3;
    s3 : int :: 1/3 last = 0;
let
  s1 = f1(s0 when (? % 3));
  s2 = f2(s1);
  s3 = f3(s2);
  s4 = current(s3, (? % 3));

  latency_chain forward <= 1 (s1 -> s2 -> s3);
  resource balance cpu;
tel
```

Aside: fby or last

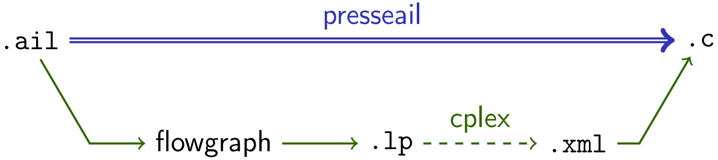
```
x = c fby e;
P
```

⇓

```
var nx : T last = c

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

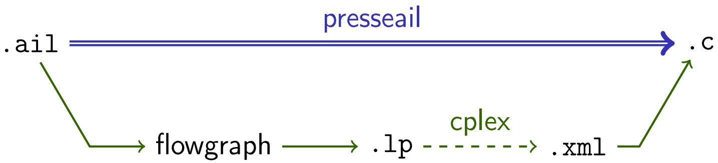
- `c fby e` initialized unit delay / register / `delay c e`
- `last x` previous value of initialized variable
  [Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]
- Here: easier to work with `last x`

Overview: compilation using Integer Linear Programming (ILP)



Overview: compilation using Integer Linear Programming (ILP)

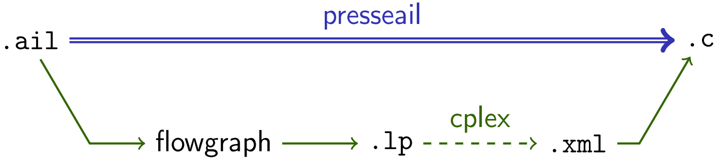
- Like Prelude [Forget, Boniol, Lesens, and Pagetti (2010): A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems]
  But, no WCET, no deadlines, no real-time tasks
- Rates expressed as $1/n$ of the base clock



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

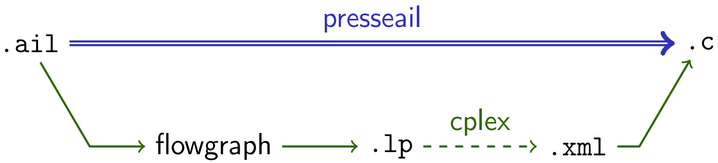
- One or more `step` functions
- Called cyclically at the base rate



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

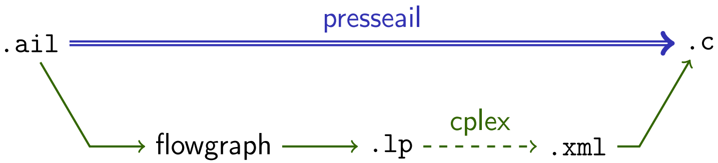


- Vertex = equation
- Arc from producer to consumer
- Independent of source language

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

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`

Multi-core execution: main approach

`presseail` $\Longrightarrow$ Heptagon $\Longrightarrow$ Lopht

presseail

- Read and analyze `.ail`
- Generate ILP: equation $\mapsto$ phase
- ~~Generate sequential code~~
- Hyperperiod expansion to Heptagon with annotations for semi-linear updates

Lopht

- Dumitru's *Logical to Physical Time compiler*
- Parallelize Heptagon output in each phase of the hypercycle
- Add inter-thread synchronization between writers and readers

Multi-core execution: main approach

`presseail` $\Longrightarrow$ Heptagon $\Longrightarrow$ Lopht

presseail

- Read and analyze `.ail`
- Generate ILP: equation $\mapsto$ phase
- ~~Generate sequential code~~
- Hyperperiod expansion to Heptagon with annotations for semi-linear updates

Lopht

- Dumitru's *Logical to Physical Time compiler*
- Parallelize Heptagon output in each phase of the hypercycle
- Add inter-thread synchronization between writers and readers

In short: fix the phases, then parallelize

Multi-core execution: experimental alternative

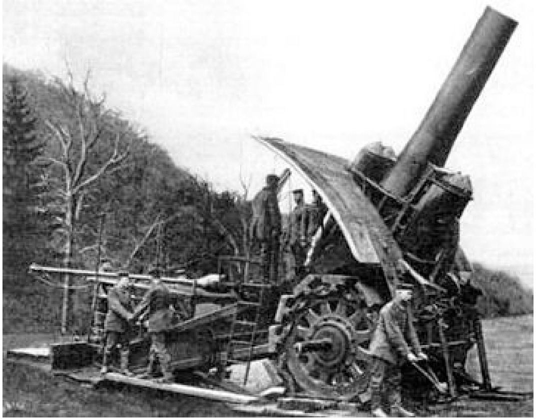
Why not?...

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

Multi-core execution: experimental alternative

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"

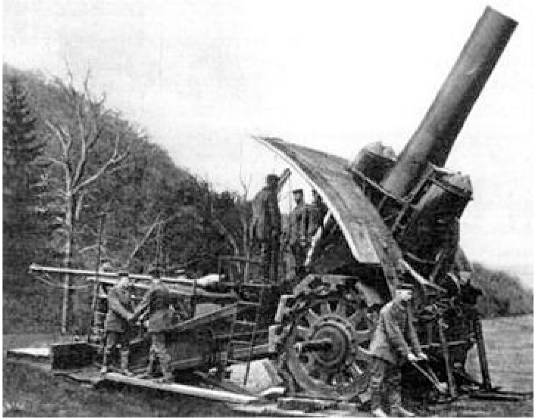
Multi-core execution: experimental alternative

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)

Threads and phases: same period

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

Threads and phases: same period

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

- Encode in ILP, start with `r = f(w)`, $\mathrm{period}(r) = \mathrm{period}(w)$

Threads and phases: same period

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

- Encode in ILP, start with `r = f(w)`, $\mathrm{period}(r) = \mathrm{period}(w)$
- New variable $b = \begin{cases} 0 & t_r = t_w \\ 1 & \textit{unconstrained} \end{cases}$ $\quad b \cdot (1 - N_t) \leq t_r - t_w \leq b \cdot (N_t - 1)$

Threads and phases: same period

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

- Encode in ILP, start with `r = f(w)`, $\mathrm{period}(r) = \mathrm{period}(w)$
- New variable $b = \begin{cases} 0 & t_r = t_w \\ 1 & \textit{unconstrained} \end{cases}$ $\quad b \cdot (1 - N_t) \leq t_r - t_w \leq b \cdot (N_t - 1)$
- Constraints on phases:
  $b - \mathrm{period}(w) \cdot (1 - y) \leq p_r - p_w \leq \mathrm{period}(r) \cdot y - b$

Threads and phases: same period

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

- Encode in ILP, start with `r = f(w)`, $\mathrm{period}(r) = \mathrm{period}(w)$
- New variable $b = \begin{cases} 0 & t_r = t_w \\ 1 & \textit{unconstrained} \end{cases}$ $\qquad b \cdot (1 - N_t) \leq t_r - t_w \leq b \cdot (N_t - 1)$
- Constraints on phases:
  $b - \mathrm{period}(w) \cdot (1 - y) \leq p_r - p_w \leq \mathrm{period}(r) \cdot y - b$
- Another new variable y to encode $|p_r - p_w|$.

Threads and phases: same period

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

- Encode in ILP, start with `r = f(w)`, $\mathrm{period}(r) = \mathrm{period}(w)$
- New variable $b = \begin{cases} 0 & t_r = t_w \\ 1 & \textit{unconstrained} \end{cases}$ $\quad b \cdot (1 - N_t) \leq t_r - t_w \leq b \cdot (N_t - 1)$
- Constraints on phases:
  $b - \mathrm{period}(w) \cdot (1 - y) \leq p_r - p_w \leq \mathrm{period}(r) \cdot y - b$
- Another new variable y to encode $|p_r - p_w|$.

  if $y = 0$ $(p_r \leq p_w)$ then $\quad b - \mathrm{period}(w) \leq p_r - p_w \leq -b$

Threads and phases: same period

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

- Encode in ILP, start with `r = f(w)`, $\mathrm{period}(r) = \mathrm{period}(w)$
- New variable $b = \begin{cases} 0 & t_r = t_w \\ 1 & \textit{unconstrained} \end{cases}$ $\qquad b \cdot (1 - N_t) \leq t_r - t_w \leq b \cdot (N_t - 1)$
- Constraints on phases:
  $b - \mathrm{period}(w) \cdot (1 - y) \leq p_r - p_w \leq \mathrm{period}(r) \cdot y - b$
- Another new variable y to encode $|p_r - p_w|$.

if $y = 0$ $(p_r \leq p_w)$ then $\quad b - \mathrm{period}(w) \leq p_r - p_w \leq -b$

$b = 0$: $\quad -\mathrm{period}(w) \leq p_r - p_w \leq 0$ $\qquad$ *unconstrained*

Threads and phases: same period

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

- Encode in ILP, start with `r = f(w)`, $\mathrm{period}(r) = \mathrm{period}(w)$
- New variable $b = \begin{cases} 0 & t_r = t_w \\ 1 & \textit{unconstrained} \end{cases}$ $\qquad b \cdot (1 - N_t) \leq t_r - t_w \leq b \cdot (N_t - 1)$
- Constraints on phases:
  $b - \mathrm{period}(w) \cdot (1 - y) \leq p_r - p_w \leq \mathrm{period}(r) \cdot y - b$
- Another new variable y to encode $|p_r - p_w|$.

if $y = 0$ $(p_r \leq p_w)$	then	$b - \mathrm{period}(w) \leq p_r - p_w \leq -b$	
	$b = 0$:	$-\mathrm{period}(w) \leq p_r - p_w \leq 0$	*unconstrained*
	$b = 1$:	$1 - \mathrm{period}(w) \leq p_r - p_w \leq -1$	*unequal*

Threads and phases: same period

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

- Encode in ILP, start with `r = f(w)`, $\mathrm{period}(r) = \mathrm{period}(w)$
- New variable $b = \begin{cases} 0 & t_r = t_w \\ 1 & \textit{unconstrained} \end{cases}$ $\qquad b \cdot (1 - N_t) \leq t_r - t_w \leq b \cdot (N_t - 1)$
- Constraints on phases:
  $b - \mathrm{period}(w) \cdot (1 - y) \leq p_r - p_w \leq \mathrm{period}(r) \cdot y - b$
- Another new variable y to encode $|p_r - p_w|$.

if $y = 0$ ($p_r \leq p_w$)	then	$b - \mathrm{period}(w) \leq p_r - p_w \leq -b$	
	$b = 0$:	$-\mathrm{period}(w) \leq p_r - p_w \leq 0$	*unconstrained*
	$b = 1$:	$1 - \mathrm{period}(w) \leq p_r - p_w \leq -1$	*unequal*
if $y = 1$ ($p_r \geq p_w$)	then	$b \leq p_r - p_w \leq \mathrm{period}(r) - b$	

Threads and phases: same period

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

- Encode in ILP, start with `r = f(w)`, $\mathrm{period}(r) = \mathrm{period}(w)$
- New variable $b = \begin{cases} 0 & t_r = t_w \\ 1 & \textit{unconstrained} \end{cases}$ $\quad b \cdot (1 - N_t) \leq t_r - t_w \leq b \cdot (N_t - 1)$
- Constraints on phases:
  $b - \mathrm{period}(w) \cdot (1 - y) \leq p_r - p_w \leq \mathrm{period}(r) \cdot y - b$
- Another new variable y to encode $|p_r - p_w|$.

if $y = 0$ ($p_r \leq p_w$) then		$b - \mathrm{period}(w) \leq p_r - p_w \leq -b$	
	$b = 0$:	$-\mathrm{period}(w) \leq p_r - p_w \leq 0$	*unconstrained*
	$b = 1$:	$1 - \mathrm{period}(w) \leq p_r - p_w \leq -1$	*unequal*
if $y = 1$ ($p_r \geq p_w$) then		$b \leq p_r - p_w \leq \mathrm{period}(r) - b$	
	$b = 0$:	$0 \leq p_r - p_w \leq \mathrm{period}(r)$	*unconstrained*

Threads and phases: same period

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

- Encode in ILP, start with `r = f(w)`, $\mathrm{period}(r) = \mathrm{period}(w)$
- New variable $b = \begin{cases} 0 & t_r = t_w \\ 1 & \textit{unconstrained} \end{cases}$ $\quad b \cdot (1 - N_t) \leq t_r - t_w \leq b \cdot (N_t - 1)$
- Constraints on phases:
  $b - \mathrm{period}(w) \cdot (1 - y) \leq p_r - p_w \leq \mathrm{period}(r) \cdot y - b$
- Another new variable y to encode $|p_r - p_w|$.

if $y = 0$ $(p_r \leq p_w)$	then	$b - \mathrm{period}(w) \leq p_r - p_w \leq -b$	
	$b = 0$:	$-\mathrm{period}(w) \leq p_r - p_w \leq 0$	*unconstrained*
	$b = 1$:	$1 - \mathrm{period}(w) \leq p_r - p_w \leq -1$	*unequal*
if $y = 1$ $(p_r \geq p_w)$	then	$b \leq p_r - p_w \leq \mathrm{period}(r) - b$	
	$b = 0$:	$0 \leq p_r - p_w \leq \mathrm{period}(r)$	*unconstrained*
	$b = 1$:	$1 \leq p_r - p_w \leq \mathrm{period}(r) - 1$	*unequal*

Threads and phases: rate change

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

- What about `r = w when (s % n)` and `r = current(w, (s % n))`?

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

- What about `r = w when (s % n)` and `r = current(w, (s % n))`?
- Same idea:
  $b - \mathrm{period}(w) \cdot (1 - y) \leq p_r^* - p_w^* \leq \mathrm{period}(r) \cdot y - b$

Threads and phases: rate change

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

- What about `r = w when (s % n)` and `r = current(w, (s % n))`?
- Same idea:
  $b - \mathrm{period}(w) \cdot (1 - y) \leq p_r^* - p_w^* \leq \mathrm{period}(r) \cdot y - b$
- For `r = w when (s % n)`, $p_r^* = \delta$, where $p_r = k \cdot \mathrm{period}(w) + \delta$
  i.e., $\delta = p_r \mod \mathrm{period}(w)$, but it costs 2 new variables

Threads and phases: rate change

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

- What about `r = w when (s % n)` and `r = current(w, (s % n))`?
- Same idea:
  $b - \text{period}(w) \cdot (1 - y) \leq p_r^* - p_w^* \leq \text{period}(r) \cdot y - b$
- For `r = w when (s % n)`, $p_r^* = \delta$, where $p_r = k \cdot \text{period}(w) + \delta$
  i.e., $\delta = p_r \mod \text{period}(w)$, but it costs 2 new variables
- For `r = current(w, (s % n))`, $p_w^* = \delta$, where $p_w = k \cdot \text{period}(r) + \delta$
  i.e., $\delta = p_w \mod \text{period}(r)$, but it costs 2 new variables

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
...
```

Demos

- Pipelining
- Chain 1 and 2
- Chain 3 with `-relax-direct`
- Chain 4

- Industrial Case-study (with partitioning)

Inconclusion

- Works for small examples (modulo bugs)
- No results for industrial case study
  - Still debugging and tweaking
  - Expensive problem to solve, not very linear
- Solutions may be prevented by
  - The "same thread or different phase" discipline
  - Pre-solve graph partitioning
- Why not replace partitioning, and maybe solving, by heuristics?
- Can minimizing same-thread-same-phase communications help Lopht?