Scheduling Rate-synchronous Lustre II: now we've really got the bools

Timothy Bourke[1] and Loïc Sylvestre[2]

[1]Inria Paris, École normale supérieure, PSL University (PARKAS team)
[1]Université de Toulouse - IRIT (TRACES team)

November 28 2025 - SYNCHRON '25 à Aussois

Motivation

- Current approach: generate ILP constraints
  - dependency constraints: integer phases + inequalities
  - resource balancing: boolean phase weights + *pseudo-boolean* equations
  - end-to-end latency: integer phases + encoding
- Observation: the problem is more combinatorial than linear, especially with multiple threads

- Question 1: Why not try a contraint solver like cp-sat?
- Question 2: What about a pure SAT encoding?

- Work-in-Progress: very preliminary results

CP-sat (1/2)

- a constraint solver based on SAT and constraint programming
- supporting ILP constraints via automatic encoding in SAT internally
- global constraints (e.g., `all-different`, `atmost-one`, `exactly-one`)
- developed by Google, but open source
- API for C++, python, Java
- automatic parallelization
- Gold medal of the `MiniZinc` competition, each year since 2013

CP-sat (2/2)

```
import sys
from ortools.sat.python import cp_model

model = cp_model.CpModel()

a = model.new_int_var(1, 100, 'a')
b = model.new_int_var(1, 100, 'b')
c = model.new_int_var(1, 100, 'c')
d = model.new_int_var(1, 100, 'd')
objective = model.new_int_var(0, 100, 'objective')

model.AddAllDifferent([a, b, c, d])
model.add(a + b + c + d  == objective)

model.minimize(objective)
solver = cp_model.CpSolver() #   => {a = 4, b = 3, c = 2, d = 1}
```

Constraint generation for All-in-Lustre (1/3)

Defining the problem

```
import sys
from ortools.sat.python import cp_model

...
model = cp_model.CpModel()

rsum_pw_0_ops = model.new_int_var(0,1920,'rsum_pw_0_ops')
rsum_pw_1_ops = model.new_int_var(0,1920,'rsum_pw_1_ops')
...
pw_0_vz_control = model.new_bool_var('pw_0_vz_control')
pw_1_vz_control = model.new_bool_var('pw_1_vz_control')
...
model.add(-1 * p_vz_filter + (0) + (pw_1_vz_filter)
                    + (2 * pw_2_vz_filter) + (3 * pw_3_vz_filter) == 0)
...
```

Constraint generation for All-in-Lustre (2/3)

Solving the problem

```
__objective__ = model.new_int_var(0,max_int,'__objective__')

model.add(__objective__ == rmax_cycle_ops)
model.minimize(__objective__)

solver = cp_model.CpSolver()
solver.parameters.relative_gap_limit = mip_gap
solver.parameters.num_workers = num_workers
solver.parameters.log_search_progress = True
...

status = solver.solve(model)
```

Constraint generation for All-in-Lustre (3/3)

Printing the schedule (to use it then in the compiler)

```
<variables>
  <variable name="rsum.pw.0.ops" value="82"/>
  <variable name="rsum.pw.1.ops" value="1272"/>
  <variable name="rsum.pw.2.ops" value="558"/>
   ...
</variables>
<objectiveValues>
  <objective name="__objective__" value="1272" />
</objectiveValues>
```

A first optimization

Replacing some pseudo-boolean constraints by global constraints

This constraint:

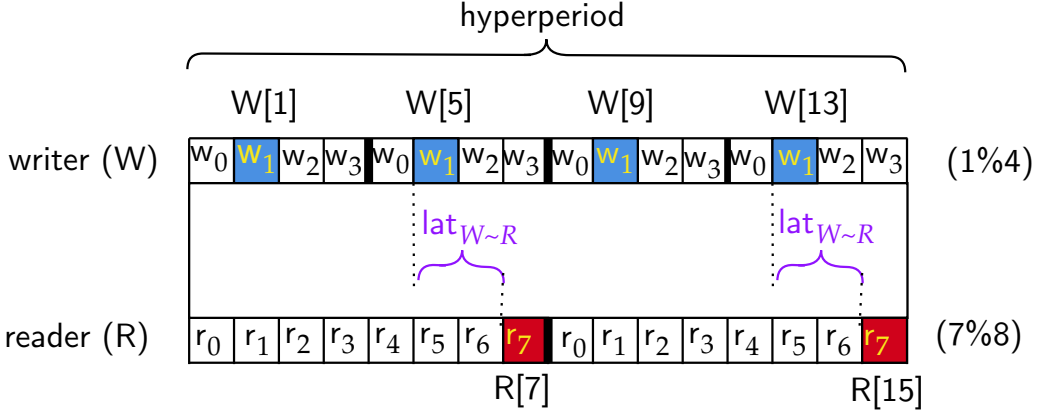
```
model.add(pw_3_vz_filter + pw_2_vz_filter + pw_1_vz_filter
    + pw_0_vz_filter == 1)
```

becomes:

```
model.add_exactly_one([pw_3_vz_filter, pw_2_vz_filter, pw_1_vz_filter
                           , pw_0_vz_filter])
```

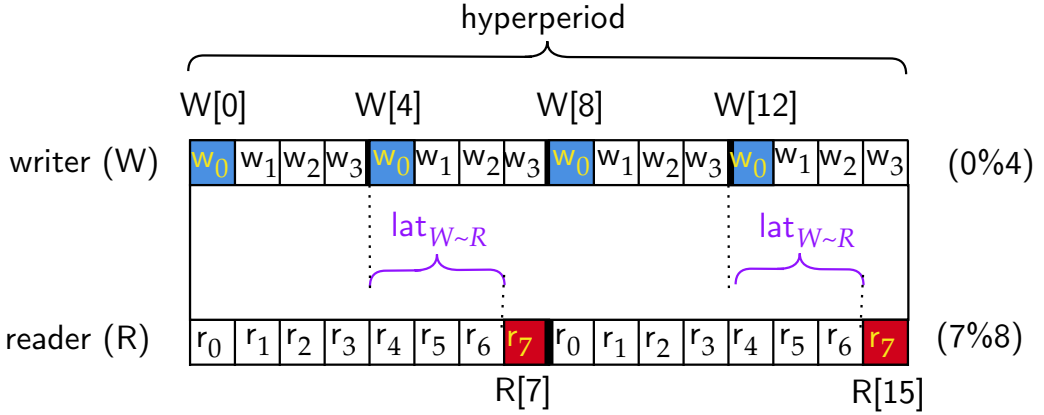
Measured speedup ratio: 11%
⇒ encouraging SAT encoding for latency chains and causality

Latency from a writer to a reader



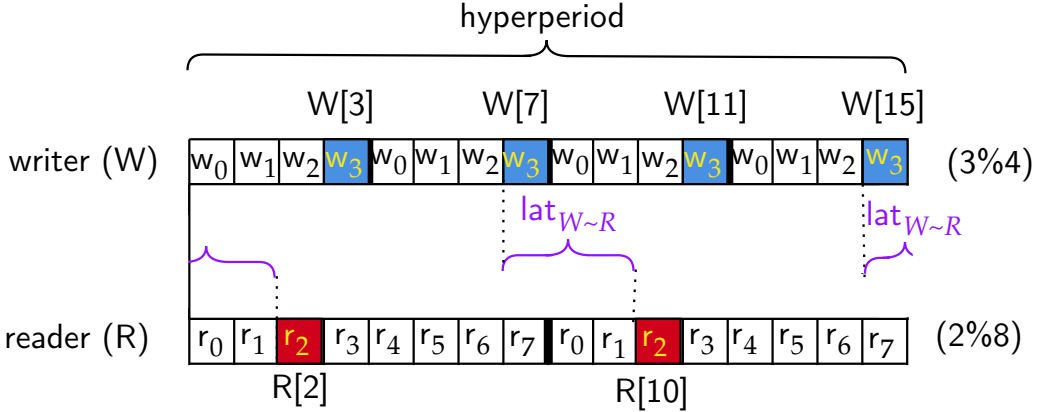
notation: W[i] := W_k **where** $k := i \,\%\, period(W)$

Latency from a writer to a reader



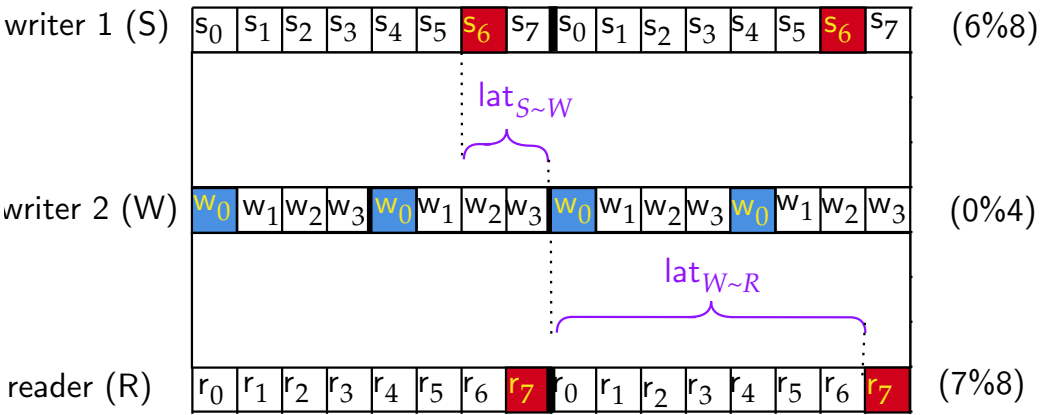
notation: W[i] := W_k **where** $k := i \,\%\, period(W)$

Latency from a writer to a reader

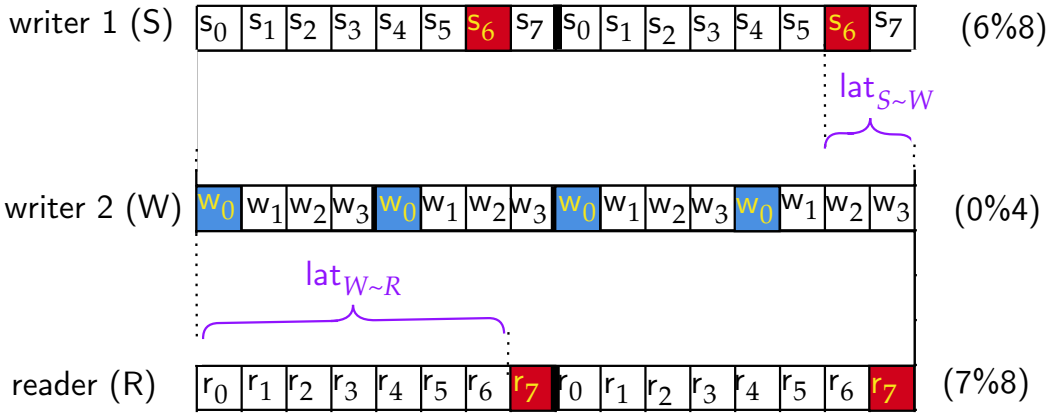


notation: W[i] := W_k **where** $k := i \,\%\, period(\mathrm{W})$

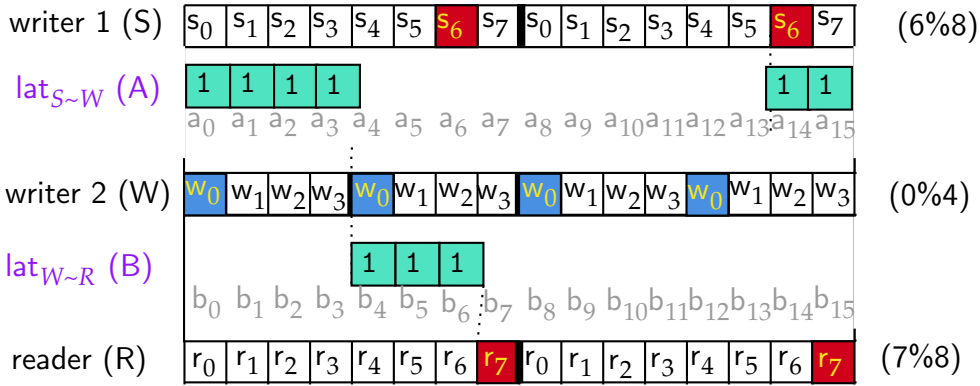
Latency chain



Latency chain



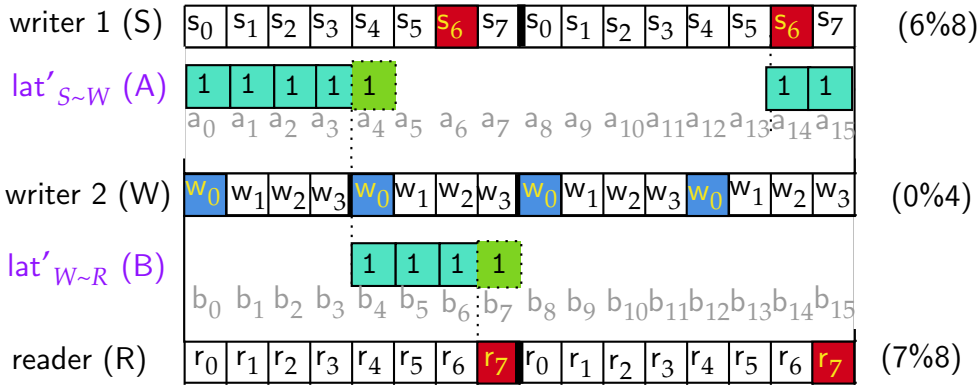
Encoding end-to-end latency: counting the 1s



$$\text{end-to-end-latency}(X_1, X_2, \ldots X_n) = \sum_{i=1}^{n-1} \text{lat}_{X_i \sim X_{i+1}} < c$$

for instance: end-to-end-latency(S,W,R)

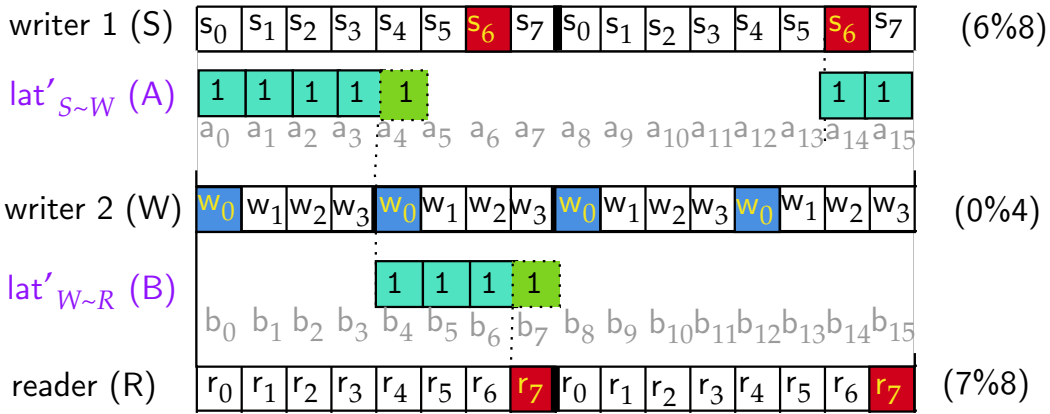
Encoding end-to-end latency: counting the 1s



$$\text{end-to-end-latency}(X_1, X_2, \ldots X_n) = \left(\sum_{i=1}^{n-1} \text{lat}'_{X_i \sim X_{i+1}}\right) - (n-1) < c$$

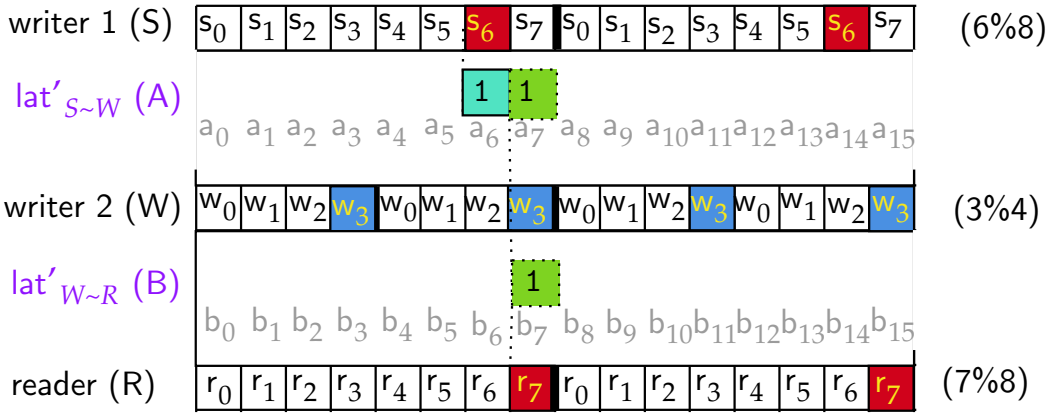
for instance: end-to-end-latency(S,W,R)

latency from a writer to a reader: **forward case**



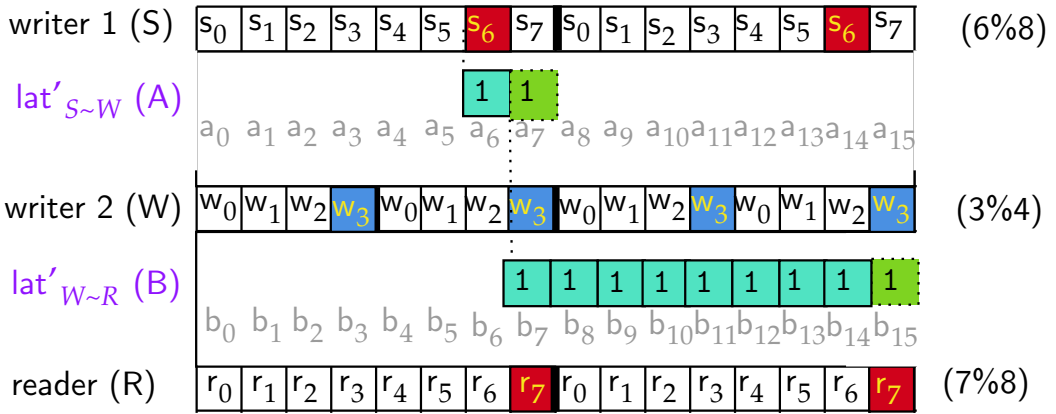
forward case: B[i] ⇔ (W[i] & A[i]) or (B[i-1] & not(R[i-1]))

latency from a writer to a reader: **forward case**



forward case: B[i] ⇔ (W[i] & A[i]) or (B[i-1] & not(R[i-1]))

latency from a writer to a reader: **backward case**



backward case:

B[i] ⇔ (W[i] & A[i]) or (B[i-1] & not(R[i-1]))
or (W[i-1] & R[i-1] & not(A[i]))

How to compute the initial previous latency?

- add an initial latency variable (P) selecting one of the instances of the first writer
- In the previous example, constraints are:
  - $P[0] + P[8] = S[0]$
  - $P[1] + P[9] = S[1]$
  - $P[2] + P[10] = S[2]$
  - ...
  - $P[6] + P[14] = S[6]$
  - $P[7] + P[15] = S[7]$

For example, if the phase of the first writer (S) is 6, then only bits of S that is 1 is `S[6]`, hence either `P[7]` or `P[14]` is 1 (the solver will choose to minimize the objective); other bits of `P` are 0s.

Generating the latency constraints for cp-sat (with Python input)

Need to encode the previous formula in Conjunctive normal form (CNF)

```
def latency_constraint_forward(bi, ai, aim1, wi, wim1, bim1, rim1):
  model.add_bool_or([bi, ~ai, ~wi])
  model.add_bool_or([bi, ~bim1, rim1])
  model.add_bool_or([~bi, ai, bim1])
  model.add_bool_or([~bi, ai, ~rim1])
  model.add_bool_or([~bi, bim1, wi])
  model.add_bool_or([~bi, wi, ~rim1])

...
latency_constraint_forward(dynamics_h_filter_lat_49_v0_50,lat_init_58_v0...)
latency_constraint_forward(dynamics_h_filter_lat_49_v1_51,lat_init_58_v1...)
latency_constraint_forward(dynamics_h_filter_lat_49_v2_52,lat_init_58_v2...)
latency_constraint_forward(dynamics_h_filter_lat_49_v3_53,lat_init_58_v3...)
...
```

Preliminary results (subject to changes)

- use `cpsat` backend: significant speedups
- use global constraints in `cpsat` (`atmost1`, `exactly1`): limited speedup ($\simeq 15\%$)
- sat encoding of latency: slower than the encoding of the ILP version by `cp-sat`
- sat encoding of causality (work in progress)
- resource variable (e.g., WCET): the number of bits needed is sometimes high (e.g., 28 bits) for bitblasting
- other backends
  - Z3 – but no automatic parallelization nor mip gap for minimization
  - using MiniSAT+ (Eén& Sörensson) for translating pseudo-boolean constraints to pure boolean ones – but seems less efficient than `cp-sat` automatic encoding
    (e.g., translating time is longer than solving the original problem)

Future Work

- finish encoding and fix bugs
- simplifying compiler structure to combine the different backends
- systematic benchmarking and understand solver behaviors
- compare the approach with other approaches, such as list scheduling