Dynamic Scheduling of Synchronous Programs in Lucid Synchrone

Adrien Guatto
Joint work with L. Mandel and M. Pouzet

PARKAS team, LIENS & INRIA

SYNCHRON 2011
What this is about

Alternative titles

- Modular code generation for **Lustre** / **Lucy-n** without static clock information
- Experiments with Latency-Insensitive Design in **Lucid Synchrone**
- One use of higher-order stream functions

Bottom line

A latency insensitive shallow embedding of **Lustre/Lucy-n** in **Lucid Synchrone**.
Introduction

Context

A latency-insensitive protocol

Prototype implementation in LUCID SYNCHRONE

Conclusion
**Original motivations**

**LUCY-N**
A variant of Lustre with:
- ultimately periodic sampling/merging conditions;
- a buffer operator.

**lucync**
The compiler’s role is to:
- infer clocks;
- compute buffer sizes;
- generate code.
Lustre 101

```lucid
let node f c = o where
    rec o = merge c m 42
    and m = 0 fby (m + 1)

f(true fby false fby true fby true fby false fby true...)
```

| time | $t_0$ | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | ...
|------|------|------|------|------|------|------|------|
| $c$  | true| false| true| true| false| true| ...
| $o$  | 0    | 42   | 1    | 2    | 42   | 3    |
| $m$  | 0    | .    | 1    | 2    | .    | 3    |

Clocks:

- $f :: \text{'a -> 'a}$
- $m :: \text{'a on c}$

In the generated code, state changes for $m$ must occur exactly when $c$ is true.
**Lucy-n (101)**

```plaintext
let node f x = o where
  rec o = buffer v1 + v2
  and v1 = x when (10)
  and v2 = x when (01)
```

<table>
<thead>
<tr>
<th>time</th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>x</td>
<td>x0</td>
<td>x1</td>
<td>x2</td>
<td>x3</td>
<td>x4</td>
<td>x5</td>
</tr>
<tr>
<td>v1</td>
<td>x0</td>
<td>x2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v2</td>
<td></td>
<td></td>
<td></td>
<td>x3</td>
<td>x5</td>
<td></td>
</tr>
<tr>
<td>buffer v1</td>
<td>x0</td>
<td>x2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>x0 + x1</td>
<td>x3 + x2</td>
<td>x5 + x4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Clocks:**

- v1 :: 'a on (10)
- v2 :: 'a on (01)
- o :: 'a on (01)
Traditional modular code generation for Lustre

```
let node f c = o where
  rec o = merge c m 42
  and m = 0 fby (m + 1)

m :: 'a on c
o :: 'a
```

class f:

```java
mem m = 0;
method step(in c, out o):
  if (c):
    o := m;
    m := m + 1;
  else:
    o := 42;
```

- Compiling means translating equations with (implicit) activation rhythms to guarded affectations.
- Code generation translates clock types to conditional statements.
Modular code generation for **Lucy-n**

```ocaml
let node f (x, y) = x when (1001) + y when (0110)
```

```ocaml
val f :: forall 'a.
  'a on (011110) * 'a on (110011) -> 'a on (010010)
```

**Clocking Lucy-n**

- Clock types feature ultimately periodic binary words rather than names.
- Clocking a program amounts to solving some cyclic scheduling problem.
- Clocks are *schedules*, and thus `lucync` has to invent clocks that are not present in the source program.
- This may pose a practical problem for code generation with the previous method.
Circumventing the clock generation problem

```plaintext
let node g () = (o1, o2) where
  rec n = 0 fby (n + 1)
  and o1 = buffer (n when (00101)) + 1 when (10)
  and o2 = buffer (n when (01)) + 2 when (01)

  n :: α on (1101011001100110011010100110011001100)
```

Ideas

- Have the clocking pass generate simpler clocks;
- generate more efficient code for the given clocks:
  - try some compression methods on words;
  - decompose words into simpler ones thanks to algebraic properties;
- discard the static clock information and compute the activation rhythms on line ("clocking" at run-time).
Where are clocks needed in **Lucy-N**?

- `fby`;
- `node application`;
- `buffer`.

Designing a protocol to compute clocks *on-line* means adding control signals and logic to the source program.

- Which control signals?
- What control logic?
Understanding control signals through buffers

Which control signals for the buffer?

- **Req:** “I want to read in the buffer” bit.
- **Ok:** “I want to write in the buffer” bit.
- For modularity reasons, we add these signals everywhere.
What’s in an interface for source-level values of type $\alpha$?

- $req$, boolean: G tells F “Give me data!”;
- $data$, of type $\alpha$: F sends G data of type $\alpha$;
- $ok$, boolean: F tells G “I’m giving you valid data”. 

The protocol
Behaviors for various constructs

▸ constants $c$:
  
  \[
  \text{ok} = \text{req}, \text{data} = c;
  \]

▸ synchronous operators $(+, \ldots)$:
  force synchronization of operands;

▸ merge of $e_1$ and $e_2$:
  set either $\text{req}_1$ or $\text{req}_2$ according to condition;

▸ when:
  set $\text{ok}$ according to the sampling condition;

▸ buffer:
  eager, always ask the producer for data when non-empty;

▸ fby:
  initialized buffer of size one.
Local synchronization

\[ x + (y \text{ when } (001)) \]
Behaviors for various constructs

- **constants c:**
  \[
  ok = req, data = c;
  \]

- **synchronous operators (+, ...):**
  force synchronization of operands;

- **merge of \( e_1 \) and \( e_2 \):**
  set either \( req_1 \) or \( req_2 \) according to condition;

- **when:**
  set \( ok \) according to the sampling condition;

- **buffer:**
  eager, always ask the producer for data when non-empty;

- **fby:**
  initialized buffer of size one.
Lazy sampling

merge (10) x (y when (01))
Eager sampling

\[ \text{merge} \ (10) \ x \ (y \ \text{when} \ (01)) \]
Behaviors for various constructs

- **constants c:**
  \[ ok = req, \ data = c; \]

- **synchronous operators (+, \ldots):**
  force synchronization of operands;

- **merge of \( e_1 \) and \( e_2 \):**
  set either \( req_1 \) or \( req_2 \) according to condition;

- **when:**
  set \( ok \) according to the sampling condition;

- **buffer:**
  eager, always ask the producer for data when non-empty;

- **fby:**
  initialized buffer of size one.
Some remarks

▶ Invariant: it is impossible to receive data that was not asked for: \( \neg \text{req} \Rightarrow \neg \text{ok} \).

▶ Each construct is naturally delay insensitive, in the sense that the functional behavior of the program do not change if it receives spurious 0 on its control wires.

▶ Multiple reads are no longer free, since we have to somehow merge the two \text{req} wires!
Expressing the translation from the typing point of view?

\[[\alpha] = \text{bool} \Rightarrow \alpha \times \text{bool}\]

In Lucid Synchrone, we can use higher-order stream functions:

\text{my\_plus} : (\text{bool} => \text{int} \times \text{bool}) \times (\text{bool} => \text{int} \times \text{bool})
\rightarrow (\text{bool} => \text{int} \times \text{bool})
Some code

```ocaml
let node my_const c req = (c, req)

val my_const :: 'a -> (bool => 'a * bool)

let node my_when s e req = (o, ok) where
  rec req_in = req || not b
  and (o, ok) = run e req_in
  and ok = req && b && ok_in
  and b = bit_of s
  and w =
    s fby (if shift then shift_sampler s else s)

val my_when :
  sampler -> (bool => 'a * bool) -> (bool => 'a * bool)
```

Adrien Guatto – Dynamic Scheduling of Synchronous Programs in Lucid Synchrone
Synchronization

```ml
let node my_synchro e1 e2 (clock req) = (o, ok) where
  rec req1 = req && empty1 and req2 = ...

  and (v1, ok1) = run e1 req1 and (v2, ok2) = ...

  and ok1' = ok1 || not empty1 and ok2' = ...
  and v1' = if empty1 then v1 else b1 and v2' = ...

  and ok = ok1' && ok2'
  and o = (v1', v2')

  and b1 = v1 fby v1' and b2 = ...

  and empty1 = true fby (ok || (not ok1 && empty1))
  and empty2 = ...

val my_synchro :
  (bool => 'a * bool) * (bool => 'b * bool) -> (bool => ('a * 'b) * bool)
```

Adrien Guatto – Dynamic Scheduling of Synchronous Programs in Lucid Synchrone
Remarks and perspectives

Related work

- Latency-Insensitive Design (Carloni et al.), and in particular...
- Synchronous ELastic Flow (Kishinevsky et al.).

Remarks

- using statically scheduled code inside a dynamically scheduled context is easy;
- ignoring control-flow issues, a SELF-like protocol may be preferable.
- we do not target hardware implementation (combinatorial pathes everywhere!);
- we have experimented with a truly asynchronous implementation of the protocol in ERLANG.
Conclusion and future work

What we did present
A dynamic scheduling protocol for Lucy-n (or Lustre) akin to Latency-Insensitive Design.

TODO list

- Conjecture: well-clocked programs are live.
- Explore macro-expansion to imperative code or continuation-based functional code, and compare with the current static code generator.
- Does the Erlang experiment has anything to do with asynchronous circuits?