Code generation for multi-phase tasks on a multi-core distributed memory platform

Frédéric Fort\textsuperscript{1}, Julien Forget\textsuperscript{1}, Claire Pagetti\textsuperscript{2}

\textsuperscript{1}Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, Lille, France
first\textsuperscript{1}name.last\textsuperscript{1}name@univ-lille.fr

\textsuperscript{2}Onera, Toulouse, France
claire.pagetti@onera.fr
Outline

1. Introduction
2. A question of semantics: synchronous real-time with Prelude
3. Code generation for distributed memory
4. Conclusion
Problem: memory bottleneck and its impact on WCET

- Shared memory on multicore $\Rightarrow$ bus contentions;
- Bus contentions $\Rightarrow$ delays, hard to predict;
$\Rightarrow$ Overly pessimistic WCET.
Solution: multi-phase tasks

**PRedictable Execution Model (PREM)**

*Main idea:* decouple computation from communication.

- Tasks are split into several phases;
- Computation phases do not access shared memory;
- Communication phases contend for the bus;

⇒ No need to account for contentions in WCET of computation phases.

In this work we consider the AER 3-phase model:

- **Acquisition** of inputs;
- **Execution** (computation);
- **Restitution** of outputs.
Related works

Difficulty: writing PREM-compliant code is unintuitive.

In the litterature, we find works on:

- Schedulability analysis;
- OS/HW support;
- Source refactoring for legacy code;
- C compiler support.
Our contribution: AER code generation

Contribution

A compiler from synchronous code, in Prelude, to multi-task AER code.

- Input: synchronous data-flow + real-time constraints;
- Output: C code, multi-task;
- Automated code generation.

Benefits:

1. Programmer abstracts from low-level details:
   - Task synchronizations;
   - Memory transfers;

2. Easy PREM vs non-PREM comparison.
Outline

1. Introduction

2. A question of semantics: synchronous real-time with Prelude

3. Code generation for distributed memory

4. Conclusion
Synchronous real-time with Prelude

Synchronous reactive programming

- Control a device in its physical environment;
- Acquire inputs - Compute - Produce Outputs ⇒ loop.
The synchronous data-flow model

- Program behaviour = succession of **instants** (reactions);
- **Synchronous hypothesis**: computations complete before the next instant;
  - ⇒ We can ignore the duration of an instant;
  - ⇒ Behaviour described on a logical time scale;
- Expressions and variables = flows (infinite sequences);
- **Clock** of a flow = its logical time scale.
Example: a simple Lustre program

```
node simple(i, j: int; c: bool) returns (o, p: int; q: int when c)
let
  o = i + j;
  p = 0 fby (p + 1);
  q = i when c;
tel
```

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>o</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>p</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>q</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: a simple Lustre program

```lustre
def node simple(i, j: int; c: bool) returns (o, p: int; q: int when c)
  let
    o = i + j;
    p = 0 fby (p + 1);
    q = i when c;
  in
end
```

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>o</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>p</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>q</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What about real-time constraints?
Debunking the zero-time myth

Misleading claims:

- An instant takes zero time?
  - Only an idealized model, computation still does take time;
  - The synchronous hypothesis must be validated by a WCET analysis;
- Inputs, computations, outputs, within an instant are simultaneous?
  - From a logical time point-of-view, indeed;
  - However, execution order, within an instant, must respect data-dependencies: causality.
Debunking the zero-time myth

Misleading claims:

- An instant takes **zero time**?
  - Only an idealized model, computation still does take time;
  - The synchronous hypothesis must be **validated** by a WCET analysis;
- Inputs, computations, outputs, within an instant are **simultaneous**?
  - From a logical time point-of-view, indeed;
  - However, execution order, within an instant, must respect data-dependencies: **causality**.

Can we mix logical time with real-time?
Synchronous model vs AER model

- In the synchronous model, data-dependencies are **explicit**:
  - Tasks have no side-effects;
  - No implicitly shared state;
- All inputs must be available before task execution starts;
- All outputs are produced at task completion;
⇒ Synchronous model: a natural fit for the AER model.
Synchronous model vs AER model

- In the synchronous model, data-dependencies are **explicit**:
  - Tasks have no side-effects;
  - No implicitly shared state;
- All inputs must be available before task execution starts;
- All outputs are produced at task completion;
⇒ Synchronous model: a natural fit for the AER model.

Ok, but what about real-time constraints?
The multi-rate synchronous model in Prelude

Duration of instants

- **Scale 1**: long instants (30ms)
  - S
  - F
  - F
  - F
  - F
  - F

- **Scale 2**: short instants (10ms)
  - S
  - F
  - F

Different logical time scales ⇒ different durations for instants;
Real-time serves as a reference between different logical time scales.

Relaxed synchronous hypothesis

Computations complete before next activation (as good ol’ Liu&Layland).
Simple multi-rate example

Multi-rate system

$\text{period} = 10 ms$

$\text{period} = 30 ms$
Synchronous real-time with Prelude

Multi-rate communications: rate transition operators

Example

```plaintext
node sampling(i: rate (10, 0)) returns (o)
    var vf, vs;
let
    (o, vf)=F(i, (0 fby vs)*^3);
    vs=S(vf/^3);
tel
```

| date  | 0  | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | ...
|-------|----|----|----|----|----|----|----|----|----|-------
| vf    | vf₀| vf₁| vf₂| vf₃| vf₄| vf₅| vf₆| vf₇| vf₈| ...
| vf/^³ | vf₀|    |    |    |    |    |    |    |    | ...
| vs    | vs₀| vs₁|    |    |    |    |    |    |    | ...
| 0 fby vs | 0 |    | vs₀|    | vs₁|    |    |    |    | ...
| (0 fby vs)*^³ | 0 | 0  | 0  | vs₀| vs₀| vs₀| vs₁| vs₁| vs₁| ...
```
Communication semantics: latest-value

- Output data is available at job completion;
- No inter-task synchronizations.

Example

- Advantage: easy to implement;
- Inconvenient: functional behaviour depends on schedule.
Communication semantics: Logical Execution Time (LET)

- Output data is available at job deadline.

**Example**

- Advantage: no synchronizations needed, deterministic;
- Inconvenient: potentially huge end-to-end latencies.
Synchronous real-time with Prelude

Communication semantics: causal communications

- Output data is available at job completion
- Precedence constraints between dependent tasks.

Example

![Diagram illustrating causal communications]

- Advantage: deterministic, lower latencies;
- Inconvenient: harder schedulability analysis and implementation.

Prelude relies on causal communications.
Causal communications: induced constraints

Conditions to respect the causal semantics:

1. Consumer starts after producer ends $\Rightarrow$ precedence constraints;
2. Do not overwrite data before consumer ends $\Rightarrow$ buffer copies.

Example

$T_j = 2T_i$

(1): $\tau_j^0$ after $\tau_i^0$  
(2) keep $\tau_i^0$ available
Outline

1. Introduction
2. A question of semantics: synchronous real-time with Prelude
3. Code generation for distributed memory
4. Conclusion
Target hardware model

Distributed memory architecture

Local memory ($M_i$)
- Contention-free;
- Private to a CPU;
- Implemented with:
  - Cache;
  - Scratchpad;
  - ...

Shared memory ($M_G$)
- Subject to contentions;
- Inter-core communication.
Multi-phase tasks

\[ \pi_1, T=5 \]

\[ \tau_B \]

\[ \tau_C \]

\[ \tau_D \]

Simplification 1: Sensors/Actuators have no A-/R-phase
Simplification 2: E-phases handle colocated communications
Simplification 3: Remove redundant data-dependencies
Multi-phase tasks

\[ \pi_1, T=5 \]
\[ \pi_0, T=10 \]
\[ \pi_0, T=5 \]

Simplification 1: Sensors/Actuators have no A-/R-phase
Simplification 2: E-phases handle colocated communications
Simplification 3: Remove redundant data-dependencies

\[ \tau_B \]
\[ \tau_C \]
\[ \tau_D \]

- A-phase
- E-phase
- R-phase
Multi-phase tasks

Simplification 1: Sensors/Actuators have no A-/R-phase
Multi-phase tasks

Simplification 1: Sensors/Actuators have no A-/R-phase
Simplification 2: E-phases handle *colocated* communications
Multi-phase tasks

Simplification 1: Sensors/Actuators have no A-/R-phase
Simplification 2: E-phases handle collocated communications
Simplification 3: Remove redundant data-dependencies
Code generation: non-PREM

```c
void C()
{
    int a_loc=A_C_buf;
    int b_loc;
    if (must_change_B_C())
        b_loc=B_C_buff[next_cell()];
    C_D_buf = C(a_loc, b_loc);
}
```

```c
void A()
{
    int a_loc = A();
    if (must_write_A_C())
        A_C_buff=a_loc;
}
```

- **X_X_buff**: global shared variable;
- **x_loc**: local variable;
- **must_*_X_Y**: multi-periodic communication protocol.
Code generation: PREM

```c
void C_A()
{
    wait_sem(sem_A_C);
    if (must_wait_B_C())
        wait_sem(sem_B_C);
    a_loc = read_val(A_C_buff);
    if (must_change_B_C())
        b_loc = read_val(B_C_buff);
}

void C_E()
{
    c_out = C(a_loc, b_loc);
    C_D_buff = c_out;
    post_sem(sem_C_D);
}

void A_E()
{
    a_out = A();
}

void A_R()
{
    if (must_write_A_C())
        write_val(A_C_buff, a_loc);
    if (must_post_A_C())
        post_sem(sem_A_C);
}
```
Code generation: PREM

```c
1  void C_A()
2  {  
3      wait_sem(sem_A_C);
4
5      if (must_wait_B_C())
6          wait_sem(sem_B_C);
7
8      a_loc = read_val(A_C_buff);
9
10     if (must_change_B_C())
11        b_loc = read_val(B_C_buff);
12  }
13
14  void C_E()
15  {  
16      c_out = C(a_loc, b_loc);
17
18      C_D_buff = c_out;
19
20     post_sem(sem_C_D);
21  }
```

- `X_X_buff` located in $M_G$;
- `x_loc` located in $M_i$;
- `read_val/write_val`: do $M_G \leftrightarrow M_i$ transfer;
Code generation: PREM

```c
1 void C_A()
2 {
3     wait_sem(sem_A_C);
4     if (must_wait_B_C())
5         wait_sem(sem_B_C);
6     a_loc = read_val(A_C_buff);
7     if (must_change_B_C())
8         b_loc = read_val(B_C_buff);
9 }
10
11 void C_E()
12 {
13     c_out = C(a_loc, b_loc);
14     C_D_buff = c_out;
15     post_sem(sem_C_D);
16 }
```

- **sem_X_Y**: binary semaphore for synchronization;
On FPGA;

Can switch between scratchpad and cache memories.
Experimental setup: software

- Rosace case-study (longitudinal flight controller);
- Measured speedup between:
  - PREM + scratchpad private memory;
  - Non-PREM + cache private memory;
- Both codes generated by the Prelude compiler (two options);
- Shared RAM artificially slowed down.
Results: PREM speedup (vs non-PREM)

[Diagram showing speedup results for various tasks such as engine, Vz_filter, delta_th_c, elevator, Va_control_25, delta_e_c, Va_filter, aircraft_dynamics, altitude_hold, Vz_control_25, h_filter, q_filter, az_filter, Va_c, h_c. The x-axis represents the tasks, and the y-axis represents the speedup ratio. There are three categories on the y-axis: Same clock, Clock divided by 4, and Clock divided by 8.]
Outline

1. Introduction

2. A question of semantics: synchronous real-time with Prelude

3. Code generation for distributed memory

4. Conclusion
Conclusion

Summary

- The synchronous model is a natural fit for AER;
- The relaxed synchronous model is a natural fit for implicit-deadline tasks;
- Extending Prelude for AER code generation is...natural;
- Advantages:
  - Spares error-prone low-level concerns;
  - Enables easy PREM vs non-PREM comparison.

Semantics matters.
References

J. Forget.
Prelude: programming critical real-time systems.

F. Fort and J. Forget.
Code generation for multi-phase tasks on a multi-core distributed memory platform.

Automated generation of time-predictable executables on multi-core.