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

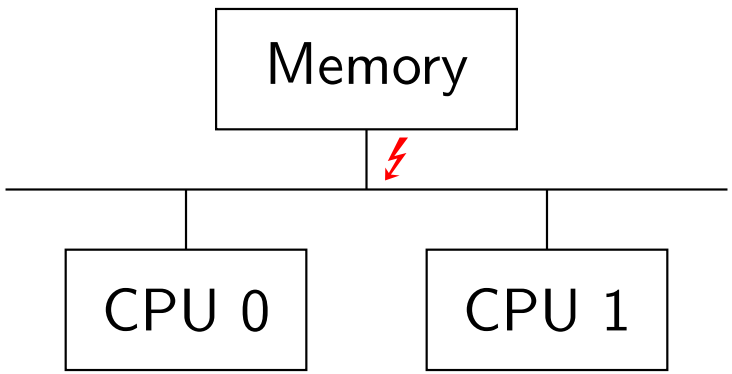
Frédéric Fort[1], **Julien Forget** [1], Claire Pagetti[2]

[1]Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, Lille, France
firstname.lastname@univ-lille.fr

[2]Onera, Toulouse, France
claire.pagetti@onera.fr

Outline

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;

$\Rightarrow$ 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.

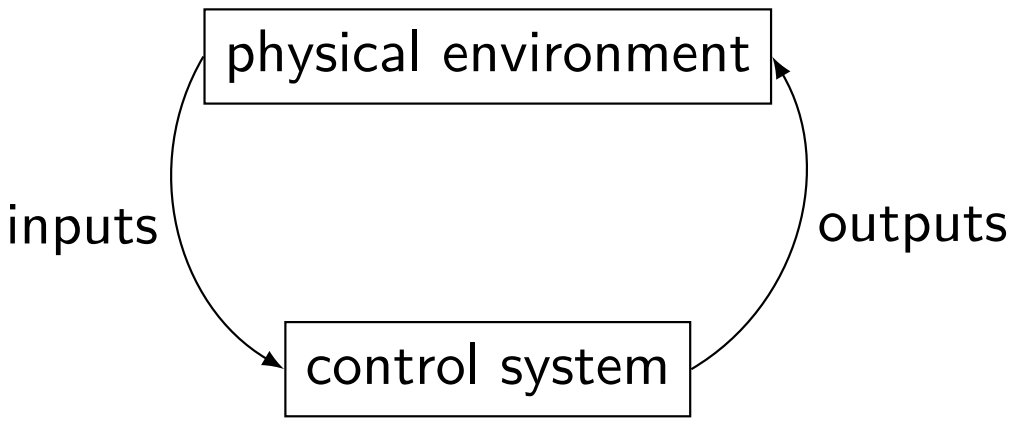
- Input: synchronous data-flow + real-time constraints;
- Output: C code, multi-task;
- Automated code generation.

Benefits:

1. Programmer abstracts from low-level details:
   - Task synchronizations;
   - Memory transfers;
2. Easy PREM vs non-PREM comparison.

Outline

Synchronous reactive programming



- Control a device in its physical environment;
- Acquire inputs - Compute - Produce Outputs $\Rightarrow$ loop.

The synchronous data-flow model

- Program behaviour = succession of **instants** (reactions);
- **Synchronous hypothesis**: computations complete before the next instant;
  - $\Rightarrow$ We can ignore the duration of an instant;
  - $\Rightarrow$ 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

Example

```
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
```

i	1	3	5	7	9
j	2	2	4	4	3
c	T	T	F	T	F
o	3	5	9	11	12
p	0	1	2	3	4
q	1	3		7	

Example: a simple Lustre program

Example

```
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
```

i	1	3	5	7	9
j	2	2	4	4	3
c	T	T	F	T	F
o	3	5	9	11	12
p	0	1	2	3	4
q	1	3		7	

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 implicitely shared state;
- All inputs must be available before task execution starts;
- All outputs are produced at task completion;

$\Rightarrow$ Synchronous model: a natural fit for the AER model.

Synchronous model vs AER model

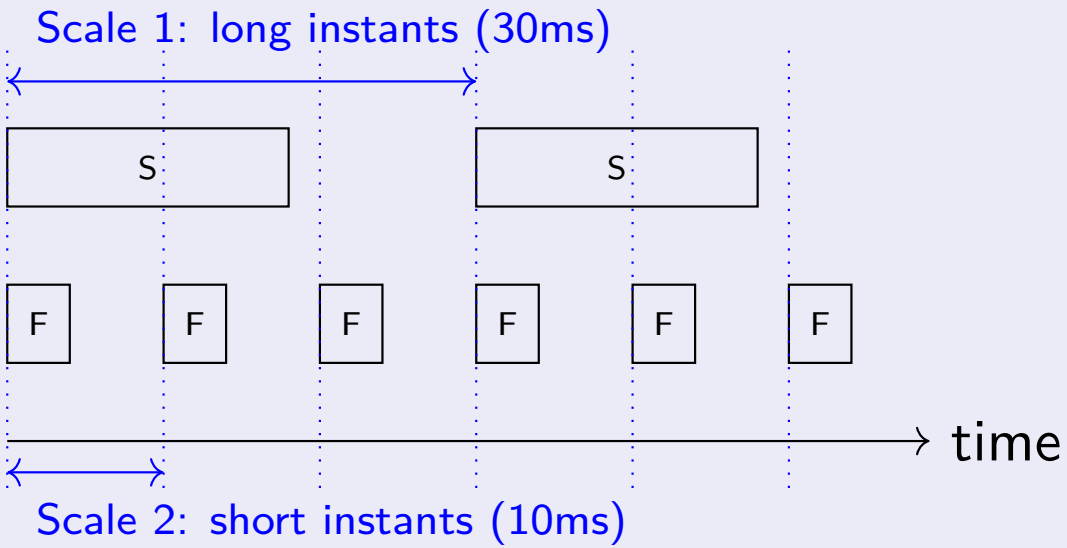
- In the synchronous model, data-dependencies are **explicit**:
  - Tasks have no side-effects;
  - No implicitely shared state;
- All inputs must be available before task execution starts;
- All outputs are produced at task completion;

$\Rightarrow$ 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



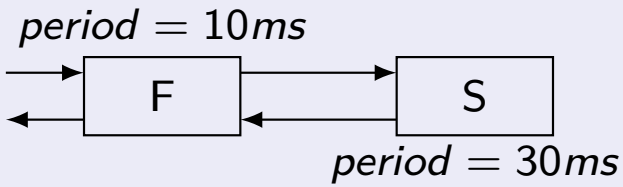
- Different logical time scales $\Rightarrow$ different durations for instants;
- Real-time serves as a reference between different logical time scales.

Relaxed synchronous hypothesis

Computations complete before next activation (as good ol' Liu&Layland).

Simple multi-rate example

Multi-rate system



Multi-rate communications: rate transition operators

Example

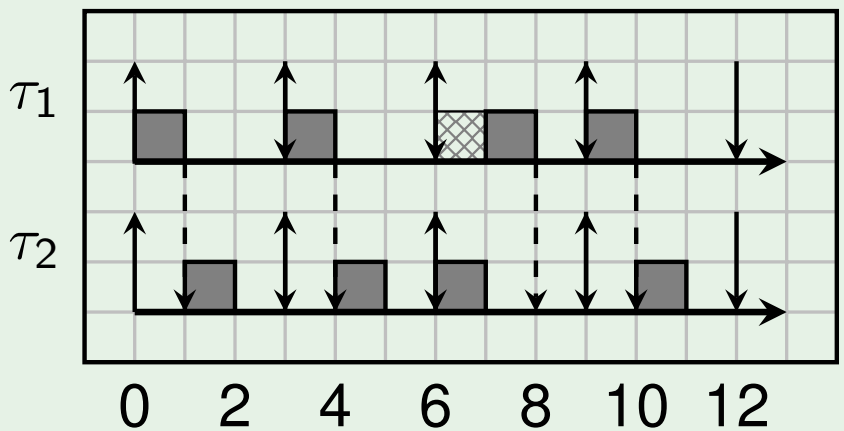
```
node sampling(i: rate (10, 0)) returns (o)
  var vf, vs;
let
  (o, vf)=F(i, (0 fby vs)*^3);
  vs=S(vf/^3);
tel
```

date	0	10	20	30	40	50	60	70	80	...
vf	vf_0	vf_1	vf_2	vf_3	vf_4	vf_5	vf_6	vf_7	vf_8	...
vf/^3	vf_0			vf_3			vf_6			...
vs	vs_0			vs_1			vs_2			...
0 fby vs	0			vs_0			vs_1			...
(0 fby vs)*^3	0	0	0	vs_0	vs_0	vs_0	vs_1	vs_1	vs_1	...

Communication semantics: latest-value

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

Example

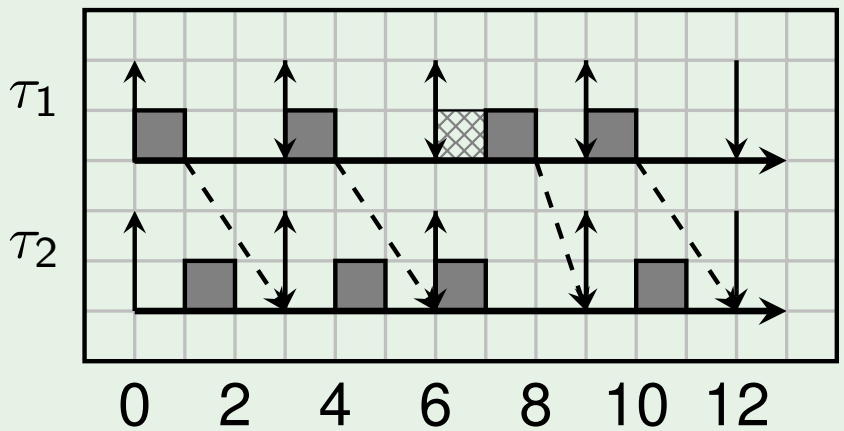


- Advantage: easy to implement;
- Inconvenient: functional behaviour depends on schedule.

Communication semantics: Logical Execution Time (LET)

- Output data is available at job deadline.

Example

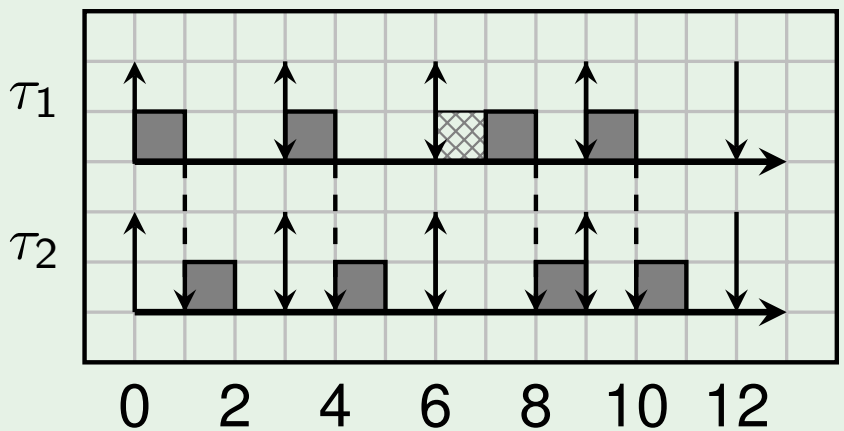


- Advantage: no synchronizations needed, deterministic;
- Inconvenient: potentially huge end-to-end latencies.

Communication semantics: causal communications

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

Example



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

Prelude relies on causal communications.

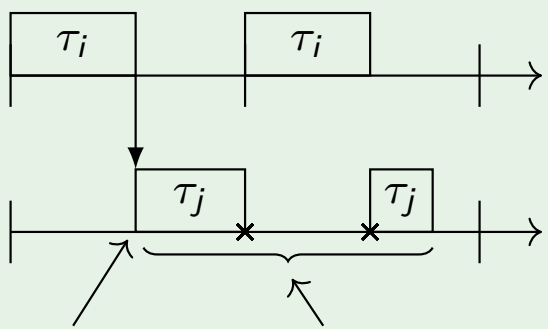
Causal communications: induced constraints

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

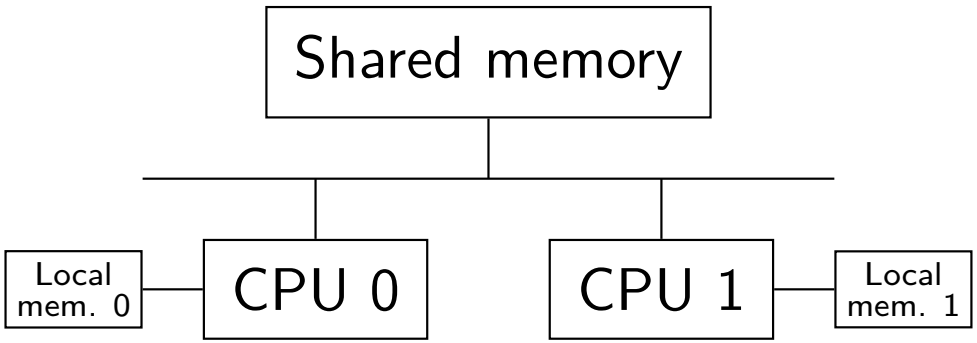
$(T_j = 2T_i)$



(1): τ_j^0 after τ_i^0 (2) keep τ_i^0 available

Outline

Target hardware model



Distributed memory architecture

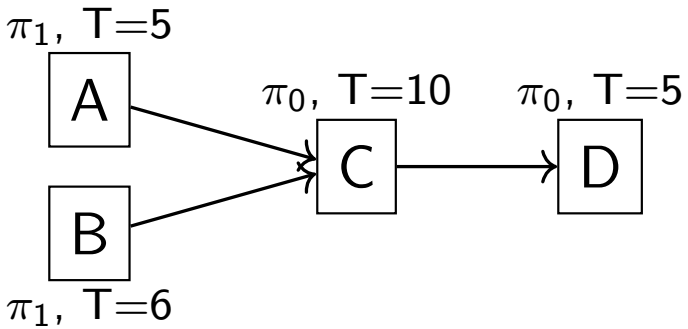
Local memory ($\mathcal{M}_i$)

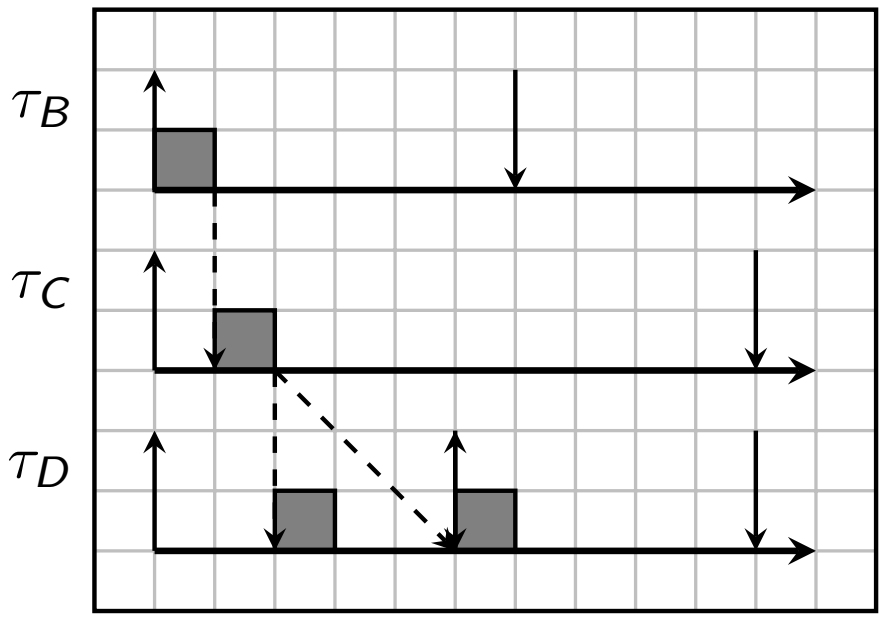
- Contention-free;
- Private to a CPU;
- Implemented with:
  - Cache;
  - **Scratchpad**
  - ...

Shared memory ($\mathcal{M}_G$)

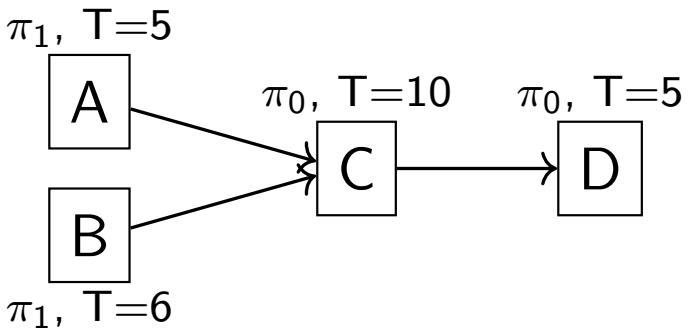
- Subject to contentions;
- Inter-core communication.

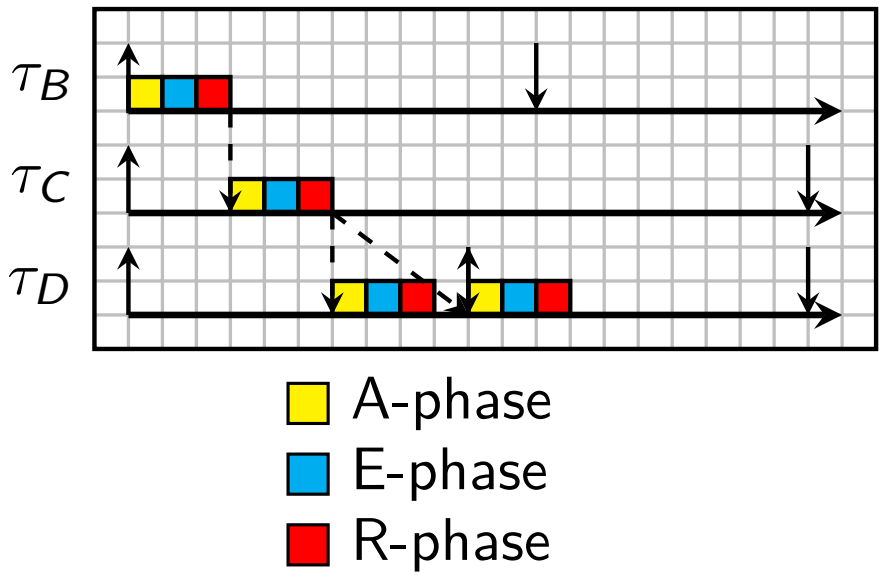
Multi-phase tasks



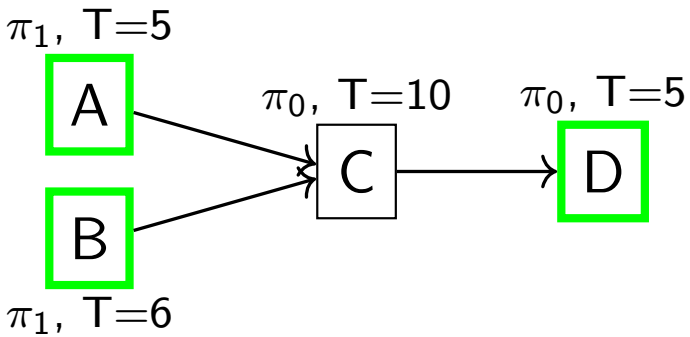


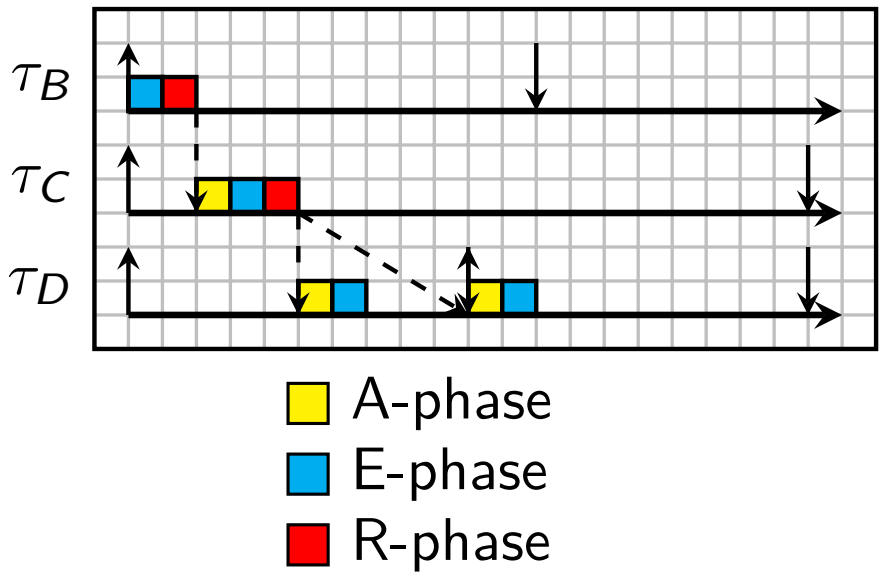
Multi-phase tasks





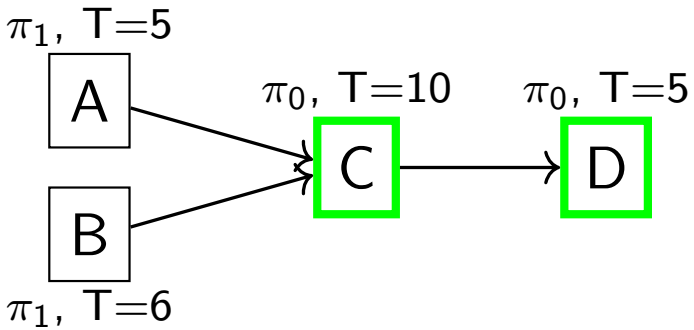
Multi-phase tasks

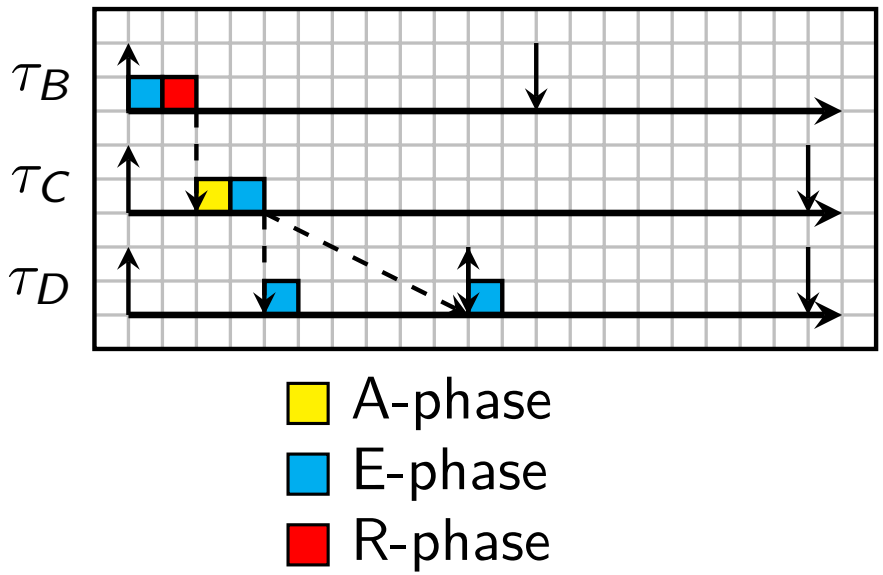




Simplification 1: Sensors/Actuators have no A-/R-phase

Multi-phase tasks

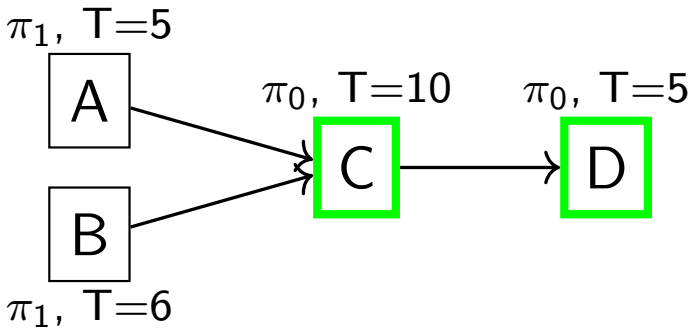


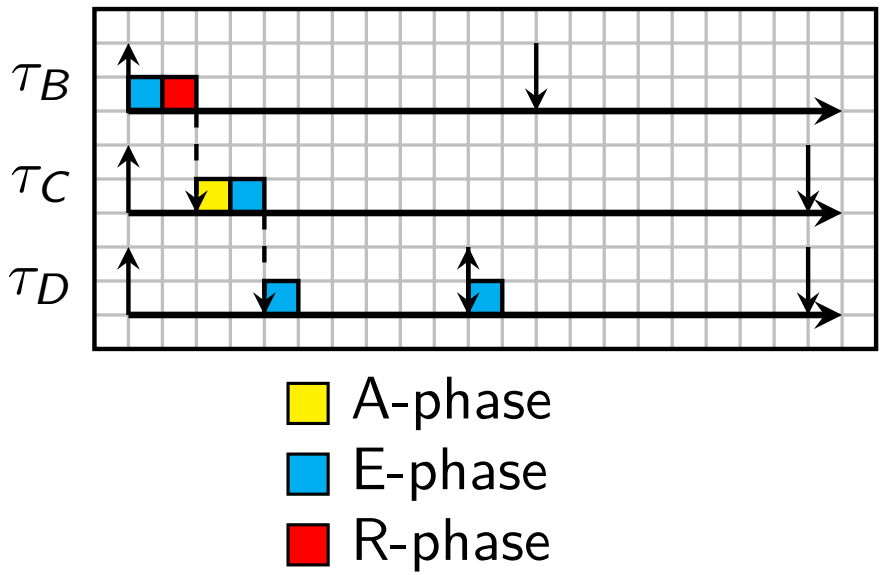


Simplification 1: Sensors/Actuators have no A-/R-phase

Simplification 2: E-phases handle **colocated** communications

Multi-phase tasks





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

Simplification 2: E-phases handle **colocated** communications

Simplification 3: Remove redundant data-dependencies

Code generation: non-PREM

```
1   void C()
2   {
3     int a_loc=A_C_buf;
4     int b_loc;
5
6     if(must_change_B_C())
7       b_loc=B_C_buff[next_cell()];
8
9     C_D_buf = C(a_loc, b_loc);
10  }
```

```
1   void A()
2   {
3     int a_loc = A();
4
5     if (must_write_A_C())
6       A_C_buff=a_loc;
7   }
```

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

Code generation: PREM

```
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  }
```

```
1   void A_E()
2   {
3     a_out = A();
4   }
5
6   void A_R()
7   {
8     if (must_write_A_C())
9       write_val(A_C_buff, a_loc);
10
11    if (must_post_A_C())
12      post_sem(sem_A_C);
13  }
```

Code generation: PREM

```
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  }
```

```
1   void A_E()
2   {
3     a_out = A();
4   }
5
6   void A_R()
7   {
8     if (must_write_A_C())
9       write_val(A_C_buff, a_loc);
10
11    if (must_post_A_C())
12      post_sem(sem_A_C);
13  }
```

- `X_X_buff` located in $\mathcal{M}_G$;
- `x_loc` located in $\mathcal{M}_i$;
- `read_val`/`write_val`: do $\mathcal{M}_G \leftrightarrow \mathcal{M}_i$ transfer;

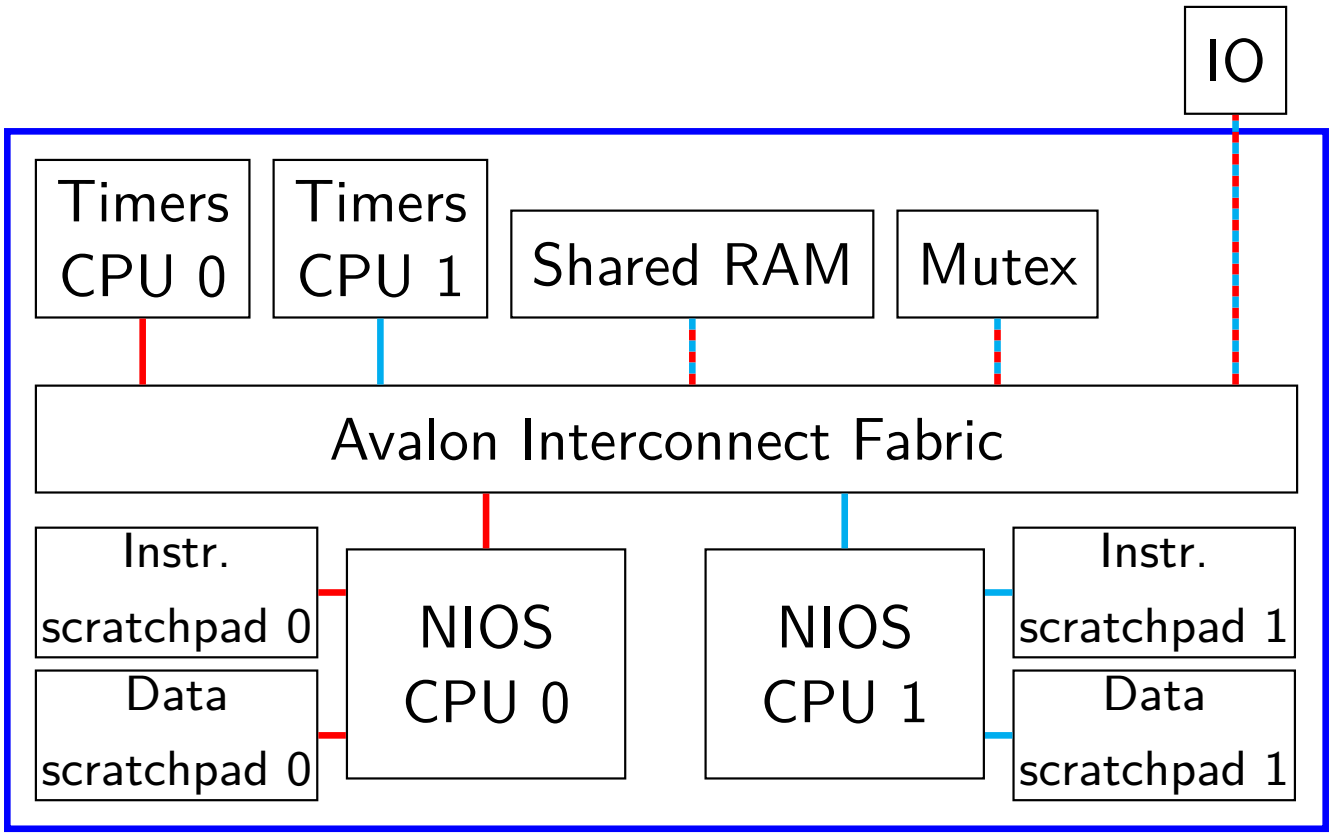
Code generation: PREM

```
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  }
```

```
1   void A_E()
2   {
3     a_out = A();
4   }
5
6   void A_R()
7   {
8     if (must_write_A_C())
9       write_val(A_C_buff, a_loc);
10
11    if (must_post_A_C())
12      post_sem(sem_A_C);
13  }
```

- `sem_X_Y`: binary semaphore for synchronization;

Experimental setup: hardware

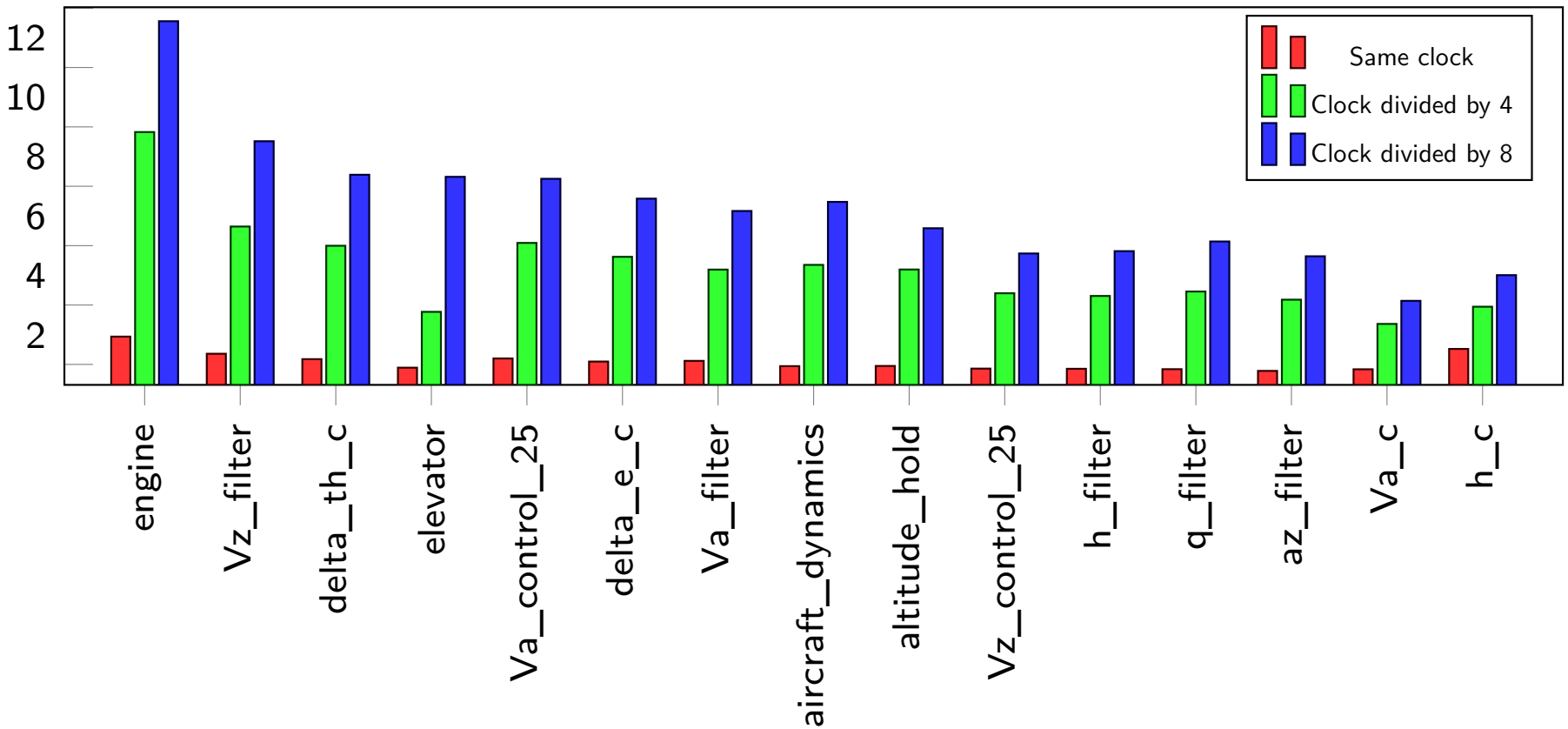


- On FPGA;
- Can switch between scratchpad and cache memories.

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)



Outline

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.
https://www.cristal.univ-lille.fr/~forget/prelude.html.

F. Fort and J. Forget.
Code generation for multi-phase tasks on a multi-core distributed memory platform.
In *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'19)*, pages 1–6. IEEE, 2019.

C. Pagetti, J. Forget, H. Falk, D. Oehlert, and A. Luppold.
Automated generation of time-predictable executables on multi-core.
In *RTNS 2018*, Proceedings of the 26th International Conference on Real-Time Networks and Systems, POITIERS, France, Oct. 2018.