WATERS Industrial Challenge 2017 with Prelude

Frédéric Boniol
ONERA, Toulouse, France
Email: frederic.boniol@onera.fr

Julien Forget
Univ. Lille, France
Email: julien.forget@univ-lille1.fr

Claire Pagetti
ONERA/ENSEEIHT/TUHH
Email: claire.pagetti@onera.fr

I. Problem description

In this section we describe our understanding of the AMALTHEA semantics [1] and of the questions raised by the WATERS challenge [2], [3].

A. Software model

The software model[1] is composed of:
1250 runnables. A *runnable* is a piece of code that reads a set of *labels* and writes another set of labels (these two sets are not necessarily disjoint). As an example, *Runnable_100ms_246* reads the labels {*Label_4530, Label_3583, Label_4338, Label_316, Label_2354, Label_4455, Label_5707, Label_7334, Label_7864*} and writes the labels {*Label_65, Label_2560, Label_9604, Label_9660*}.

A *best-case execution time* (BCET) and a *worst-case execution time* (WCET) are provided for each runnable. For instance, the execution time of *Runnable_100ms_246* belongs to $[8951, 32363]$ (in micro-seconds).

The activation of a runnable can be of two types:

- *Periodic*: for instance *Runnable_100ms_246* has a period of 100 ms;
- *Sporadic*: for instance the inter-arrival time of runnable *Runnable_sporadic_900us_1000us_4* is $900\mu s$ at minimum and $1000\mu s$ at maximum.

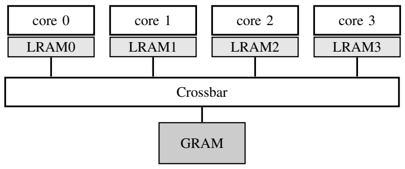
10000 labels. These serve as the communication variables between runnables. A label can be read by several runnables but can only be written by at most one, which we will call the *producer* of that label. We assume that a label that is not consumed by any runnable is an external output of the system (i.e. data sent to an actuator) and that a label that is not produced by any runnable is an external input (i.e. data originating from a sensor). Since variables shared between runnables are all explicitly declared as labels (no hidden side-effects), we can analyze data-dependencies, latencies, etc.
21 tasks. A task is a group of runnables. There are two types of tasks:

- Periodic task: it regroups runnables of the same period. For instance *Task_100ms* contains a set of runnables *Runnable_100ms_X* with $X \in [0, 246]$. Inside a task, runnables are called in sequence, as described in the task *call graph* (in this example, the call graph follows the order defined by the X index mentioned previously). There is a total of 10 periodic tasks as described in table I. There are two types of periodic tasks:
  - Preemptive: when such a task is released, it can preempt at any time any task with a lower priority. All periodic tasks with period lower than 10 ms are preemptive;
  - Cooperative: when such a task is released, it can preempt tasks with lower priority, but only between two runnable executions.
- Sporadic task: it regroups runnables that have the same minimal and maximal inter arrival times. These tasks are named as *ISRX* with $X \in [1, 11]$. These tasks are all of type Preemptive (as defined above).

[1] *ChallengeModel_w_Label- Stats_fixedLabelMapping_App4mc_v072.amxmi.*

B. Platform model

The architecture considered for the challenge [2], [3] is shown below.



Each core X has a frequency of 200MHz and is associated with a private local memory (scratchpad memory) denoted as $LRAMX$. Memory access times, assuming no contention, are the following:

- Access to the local LRAM: 1 cycle;
- Access to a distant LRAM: 9 cycles;
- Access to the GRAM: 9 cycles.

The LRAMs are single-ported, meaning that if two cores access the same local memory concurrently there is conflict that will be solved with a FIFO policy.

C. Execution model

The execution model of AMALTHEA assumes the following rules:

1) **Rule 1 – fixed-priority partitioned scheduling:** tasks are statically allocated to cores, this a priori mapping is given in table I. On each core, a RM (rate monotonic) scheduling policy is applied. Task priorities are thus assigned as follows: the highest priority tasks are the ISR tasks, they are ordered by priority in Table I (decreasing from left to right), then come the periodic tasks (ordered according to the same rule). Preemptions depend on the task type (preemptive or cooperative).
2) **Rule 2 – fixed memory mapping:** Task sections are allocated on the local LRAM of the core where the

Periodic	1ms	6660μs	2ms	5ms	10ms	20ms	50ms	100ms	200ms	1000ms
Number of runnables	41	146	27	22	303	306	45	246	14	43
Allocated core	1	1	2	2	3	2	2	2	2	2

Sporadic	ISR10	ISR5	ISR6	ISR4	ISR8	ISR7	ISR11	ISR9	ISR1	ISR2	ISR3
min	700	900	1100	1500	1700	4900	5000	6000	9500	9500	9500
max	800	1000	1200	1700	1800	5050	5100	6100	10500	10500	10500
Number of runnables	4	5	3	8	7	5	4	2	4	2	3
Allocated core	0	0	0	0	0	0	0	0	3	3	3

TABLE I
TASKS DESCRIPTION

task executes, along with all the labels produced by the runnables of this task. Note that a runnable may need to access to a distant LRAM to read some labels produced by runnables allocated to a different core;

3) **Rule 3 – LET communication (between tasks):** The LET model originates from GIOTTO [4], which follows a time-triggered approach to specify and implement embedded systems. With LET communication tasks read data at their activation and write data (to local LRAM) at their deadline;
4) **Rule 4 – Implicit communication (between runnables of the same task):** The implicit communication semantics, proposed by AUTOSAR, is quite similar to LET except that data is updated at the *start* and at the *end* of the runnable *execution* (instead of at activation/deadline).

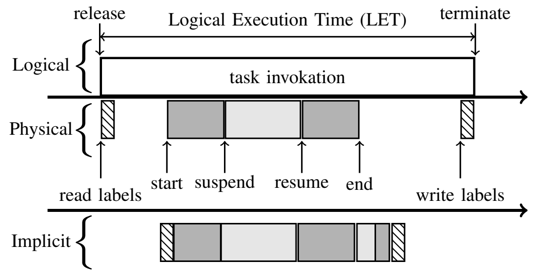


Fig. 1. LET communication and implicit communication

The top part of figure 1, strongly inspired from [5], illustrates the LET semantics; whereas the bottom part illustrates the implicit communication semantics. Dashed rectangles depict at what time labels are updated. According to *Rule 3*, *Task_10ms* has a LET of [0,10]. Thus at every time $t = k \times 10 + \epsilon$ (with $k \in \mathbb{N}$), the executive layer is in charge of copying all labels read by the task runnables into local variables. At every time $t = k \times 10 - \epsilon$ (with $k \in \mathbb{N}\backslash\{0\}$), the executive layer is in charge of copying all local variables produced by the task runnables into labels. Note that since all periodic tasks are synchronous, at time 0 (which is the critical instant) all the labels are copied first before starting the execution of any task. Now, considering the runnables inside *Task_10ms*, according to *Rule 4*, when *Runnable_10ms_X* starts executing, it first copies all the read labels produced by *Runnable_10ms_Y* to local variables. When it completes execution, it updates the its produced labels for the *Runnable_10ms_Y*.

II. PRELUDE IMPLEMENTATION

PRELUDE [6], [7] is a Synchronous Language for programming real-time embedded control systems, which provides real-time primitives to support multi-periodic systems. The *preludec* compiler generates multi-task C code, that is independent of the target scheduling policy and of the number of cores. We propose to specify the AMALTHEA system with PRELUDE and to address part of the challenge thanks to PRELUDE features and tools. In this section, we explain how to translate the system into a PRELUDE program.

A. Restrictions compared to the original specification

***Restriction** 1:* Each sporadic task is translated into a periodic task with period equal to the minimal inter arrival time.

PRELUDE can currently only deal with periodic tasks and extending the language to sporadic tasks would require an important effort.

***Restriction** 2:* Labels types are not considered.

The AMALTHEA model only provides the size of each label (in bits), while PRELUDE requires their data-type. This has very limited impact on the challenge results.

***Restriction** 3:* AMALTHEA tasks do not appear explicitly in the resulting PRELUDE program.

Lifting this restriction would require to represent a task as a hierarchical node. We did not do this part purely due to lack of time. This has no impact on the results detailed in remainder of this paper.

B. Structure of the PRELUDE challenge implementation

First all runnables are declared as *imported node* which is the terminology to express that a program element is implemented by external user-provided code.

Then, we assemble the imported nodes in the *main* node. The *main* node also specifies the system inputs, or *sensors*, and the system outputs, or *actuators*. The code 1 gives an extract of this assembly node. For instance, *Label_5707* is a sensor and *Label_9944* is an actuator. We chose to assign the rate of a sensor based on the rate of its fastest consumer, so for instance *Label_5707* has period of 100000μs. The rate of an actuator is the rate of its producer, so for instance, *Label_9944* has period $5000 \mu s$.

Code 1 (Part of the assembly description in PRELUDE*):*

```
node main(Label_5707: int rate (100000,0); ...)
returns (Label_9944;...)
var Label_65, ...
let
  (Label_65,Label_2560,Label_9604,Label_9660)=
     Runnable_100ms_246(Label_4530,Label_3583,Label_4338,
        Label_316,Label_2354,Label_4455,Label_5707,
        Label_7334, Label_7864);

  (Label_1418,Label_4530,Label_2556,Label_742)=
     Runnable_100ms_239(...);

   ... =Runnable_100ms_36(...,0 fby Label_65,...);

  (Label_2492,Label_3543,Label_4704,Label_9944)=
    Runnable_5ms_0((0 fby Label_744)*^2,(0 fby Label_586)*^2,
         0 fby Label_1394,Label_1779/^10*^3);
  ...
tel
```

The *main* node instantiates the imported nodes and describes their data-dependencies using a set of *equations* (the `let..tel` block). In the remainder of this section we detail our implementation of label communications.

a) Multi-periodic task communications: In data-flow languages, task execution rates are driven by data rates. Furthermore, the *synchronous* semantics of PRELUDE requires that the inputs and outputs of an imported node all have the same rate. For instance, since input *Label_5707* has a period of $100000 \mu s$, so does node *Runnable_100ms_246* and so does label *Label_65*.

The *clock calculus* raises an error if an imported node tries to combine flows of different rates. So, when a runnable consumes a label produced by a runnable operating at a different rate, the programmer must specify how to perform the *rate transition*. In PRELUDE, this is done via the operators $/\hat{}$ and $*\hat{}$: $/\hat{}q$ produces a flow q times slower, while $*\hat{}q$ produces a flow q times faster. The equation of *Runnable_5ms_0* illustrates rate transitions. Let us for the moment ignore the `fby` operator, which will be explained later. Label *Label_744* is accelerated by a factor of 2, because it was produced by the task *Task_10ms*. The label *Label_1779* is decelerated by 10 and accelerated by 3, because it was produced by ISR4 which has a period of $5000/10 \times 3 = 1500 \mu s$.

b) Intra-task communications: In the AMALTHEA model, inside a task runnables are called in sequence and consume labels of runnables of the same task according to the implicit communication semantics (Rule 4). In PRELUDE, the execution order depends on data-dependencies: data-consumer executes after data-producer. For instance, *Runnable_100ms_246* consumes the label *Label_4530* which is produced by *Runnable_100ms_239*. The communication is direct (no operator applied), so *Runnable_100ms_239* must execute before *Runnable_100ms_246*. The labels produced by *Runnable_100ms_246* and consumed by other runnables of the same task period cannot be consumed with a direct communication (in the AMALTHEA model, *Runnable_100ms_246* is the last runnable of the call graph of *Task_100ms*). Therefore, the communication is indirect and requires a delay of one period. This is achieved in PRELUDE by using operator `fby` (for instance *Label_65* for *Runnable_100ms_36*).

c) LET semantics: According to Rule 3 any communication between runnables with different periods must follow the LET model. This is achieved in PRELUDE by adding a `fby` before the rate transition. For instance, for *Runnable_5ms_0*, *Label_744* is first delayed and then accelerated.

C. Automatic translation of Amalthea specification in PRELUDE *program*

We have implemented an automatic translator from AMALTHEA to PRELUDE, which is not restricted to the program provided for the challenge but should also be applicable to other AMALTHEA programs. It was written in Ocaml and consists of about 2000 lines of code, half of which is code reuse from the Prelude compiler. We pruned some data unused for our analysis: the hardware model, the OS model and the event model (which only states that all runnable have deadlines equal to their periods), are ignored. The mapping model is used for analysis but is not part of the resulting PRELUDE program. The PRELUDE program file resulting from the translation is approximately four times smaller than the AMALTHEA program (mainly due to the verbosity of the xml format used by AMALTHEA).

III. LATENCY COMPUTATION

We answer in this section to the challenge issue 4, and we show that we can compute end-to-end latencies directly at the specification level (i.e. directly on the PRELUDE program) without considering the actual implementation and its execution platform.

A. Effect chains and end-to-end latencies

An effect chain is a functional path $C = (r_1 \overset{x_1}{\to} r_2 \overset{x_1}{\to} \dots \overset{x_{n-1}}{\to} r_n)$ composed of n communicating runnables. Each x_i is a *Label* (i.e., a data-flow) produced by the runnable r_i and consumed by the runnable r_{i+1}. An end-to-end latency is the time required for performing this chain of computation, from the start date of r_1 to the end date of r_n.

In the case of a multi-periodic chain (as `EffectChain_2` and `EffectChain_3`), analyzing the end-to-end latency requires to carefully consider how rate transitions are implemented. For instance, let us consider an effect chain composed of two runnables

$$C = (r_1 \overset{x_1}{\to} r_2)$$

such that the period of r_1 is 100ms and the period of r_2 is 200ms. r_1 and r_2 belong to two different tasks. First let us consider that the tasks follow the LET semantics. The dependencies scheme between r_1 and r_2 is depicted figure 2. The PRELUDE model of this chain is

$$C_{LET} = (r_1 \overset{x_1}{\to} \texttt{fby} \, /\hat{}2 \to r_2)$$

As shown figure 2, the first instance of the chain begins at t=0 and goes through r_1^1 (the first occurrence of r_1), x_1^1 (which is delayed by one period compared to r_1^1) and terminates at $t' = 400ms$ at the end of r_2^2 (the second occurrence of r_2). Its latency is $400ms$. Similarly, the second instance of the chain begins at $t = 100ms$, goes through r_1^2, x_1^2 and terminates also at $t' = 400ms$ at the end of r_2^2. Its latency is $300ms$. Thus, the worst case latency of the chain is $400ms$, and it is achieved every $200ms$.

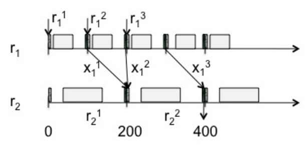


Fig. 2. Latency example in a bi-periodic effect chain with the LET semantics

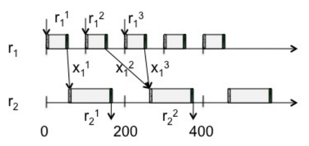


Fig. 3. Latency example in a bi-periodic effect chain with the direct semantics

Let us consider now consider the case of a direct-communication semantics: outputs are produced as soon as the runnables terminate. In that case, r_2 must wait for x_1 before starting its execution. The behavior of the chain following the direct semantics is depicted Figure 3. The PRELUDE model of this behaviour is:

$$C_{direct} = (r_1 \overset{x_1}{\rightarrow} /\hat{}\,2 \rightarrow r_2)$$

In that case, the first instance of the chain goes through r_1^1, x_1^1, r_2^1. The maximal end date of r_2^1 being 200ms, then the maximal latency of the first instance is $200ms$. The second instance of the chain begins with r_1^2 at $t = 100ms$ and terminates at the end of r_2^2 at most at $t' = 400ms$. Its maximal latency is $300ms$. The worst case latency of the chain is then $300ms$.

As shown in these examples, PRELUDE allows to automatically compute maximal end-to-end latencies of functional chains directly at the specification level [8]. We compute latencies based solely on the semantics of the language, thus the computed latency bounds are valid for any correct implementation of the PRELUDE program, independently from the scheduling policies (PRELUDE supports several of them) and from the hardware platform (mono/multi-core, etc). That way, latency requirements can be checked early on in the development cycle, even without the target platform.

B. Effect Chain 1

Let us first consider `EffectChain_1` = $(r10ms149 \overset{L3423}{\rightarrow} r10ms243 \overset{L3968}{\rightarrow} r10ms272 \overset{L2276}{\rightarrow} r10ms107)$ (where *r10msX* is a shorthand for *Runnable_10ms_X* and *LY* for *Label_Y*). This chain only involves runnables of period 10ms, it is a mono-periodic effect chain. The AMALTHEA model specifies that the runnable *r10ms107* is executed before *r10ms272*. Therefore in PRELUDE, *L2276* has to be delayed by a `fby` operator before consumption by *r10ms107*. All the other communications in the chain are direct. The PRELUDE model of this chain is:
$EC_{1,prelude} = (r10ms149 \overset{L3423}{\rightarrow} r10ms243 \overset{L3968}{\rightarrow} r10ms272 \overset{L2276}{\rightarrow} \texttt{fby} \rightarrow r10ms107)$.

The maximal latency of the chain is simply one period, plus one period for each `fby` in the chain, that is 20ms in total.

C. Effect Chain 2

`EffectChain_2` = $(r100ms7 \overset{L4258}{\rightarrow} r10ms19 \overset{L2157}{\rightarrow} r2ms8)$. This chain involves nodes executing at different rates, it is a multi-periodic chain, which contains a slow-to-fast rate transition. We consider that the slow-to-fast rate transitions follow the LET semantics. At each rate transition, we insert a delay (for the LET semantics) and rate transition operators:

- From *r100ms7* to *r10ms19* : *L4258* is delayed by one period of $100ms$, and over-sampled by a factor of 10; the period of the resulting data-flow is $10ms$.
- From *r10ms19* to *r2ms8*: *L2157* is delayed by one period of $10ms$, and over-sampled by a factor of 5.

The PRELUDE implementation of `EffectChain_2` is: $r100ms7 \overset{L4258}{\rightarrow} \texttt{fby} *\hat{}\,10 \rightarrow r10ms19 \overset{L2157}{\rightarrow} \texttt{fby} *\hat{}\,5 \rightarrow r2ms8$.

The PRELUDE end-to-end analysis method [8] begins by computing which occurrences of the first and the last runnable of the chain are related by data-dependencies. Here, because of the LET semantics, the dependencies are the following:

- The earliest occurrence of $r2ms8$ which is impacted by $r100ms7^1$ is $r2ms8^{56}$;
- More generally, the earliest occurrence of $r2ms8$ which is impacted by $r100ms7^i$ is $r2ms8^{56+50*(i-1)}$

The latency of the chain is a constant value $112ms$, i.e., the latest possible end date of $r2ms8^{56}$ minus the earliest possible start date of $r100ms7^1$. Notice how this latency is computed without taking into account the actual scheduling. Only the semantics of the program is considered.

D. Effect Chain 3

`EffectChain_3` = $(r700us800us3 \overset{L4576}{\rightarrow} r2ms3 \overset{L646}{\rightarrow} r50ms36)$. The first runnable of `EffectChain_3` is sporadic. PRELUDE has no dedicated support for sporadic nodes so we implement the sporadic runnables as periodic nodes. To ensure deterministic execution, the translator chooses the minimal inter arrival time. For the worst case latency however, we need to consider the maximal inter arrival time. So for instance, in the case of `EffectChain_3`, *r700us800us3* is implemented as a periodic node of period 800us.

Rate transitions are handled as follows:

- From *r700us800us3* to *r2ms3* : *L4576* is under-sampled by a factor of 5 and afterwards over-sampled by 2; the period of the resulting data-flow is 2ms;
- From *r2ms3* to *r50ms36*: *L646* is under-sampled by a factor of 25; the resulting period is 50ms.

The PRELUDE implementation of `EffectChain_3` is: $r700us800us3 \overset{L4576}{\rightarrow} \texttt{fby} /\hat{}\,5 *\hat{}\,2 \rightarrow r2ms3 \overset{L646}{\rightarrow} \texttt{fby} /\hat{}\,25 \rightarrow r50ms36$.

The dependencies along the chain are the following:

- $r700us800us3^i$ with $i = 1 \ldots 60$ impact $r50ms36^2$;
- $r700us800us3^i$ with $i = 61 \ldots 120$ impact $r50ms36^3$;
- $r700us800us3^i$ with $i = 121 \ldots 185$ impact $r50ms36^4$.

The maximal latency is 103.3, it is achieved for $r700us800us3^{122}$, which starts at 96.8ms and which impacts $r50ms36^{4}$, which itself ends at the latest at $200ms$.

We did an experiment to measure the impact of the LET semantics on the latency. For fast-to-slow rate transitions, we considered an IMPLICIT communication (instead of the LET semantics). The PRELUDE implementation of `EffectChain_3` is then $(r700us800us3 \xrightarrow{L4576} /\hat{}\ 5 *\hat{}\ 2 \rightarrow r2ms3 \xrightarrow{L646} /\hat{}\ 25 \rightarrow r50ms36)$. The maximal latency is achieved for $r700us800us3^{62}$, which starts at 48.8ms and which impacts $r50ms36^{3}$ ending at most at 150ms. The resulting latency is $150 - 48.8 = 101.2ms$. The LET implementation is only $2ms$ greater than the IMPLICIT implantation.

Effect chains	EC_1	EC_2	EC_3
Max latency	20ms	112ms	103.2ms

IV. IMPLEMENTATION OF THE COMMUNICATIONS

We answer in this section to the challenge issues 1 and 2, that is how to implement efficient LET communication and provide an assessment of the overhead in terms of extra cycles and memory.

A. Buffers (or Labels) allocation

a) Current implementation in PRELUDE*:* Communications in the C code generated by the Prelude compiler rely on buffering mechanisms, similar to the protocol by Sofronis et al [9]. The size of the buffers, and the read and write positions inside the buffers for each node execution are determined by the compiler in a way that ensures data consistency. These communication mechanisms also prevent race conditions and priority inversions like the Logical Execution Time model.

The language allows some flexibility in the implementation of the buffer associated to each label and of the read and write operations. Such an implementation is considered correct provided that: 1) it respects causality, meaning that the writer completes before the reader begins (except for `fby` communications); 2) tasks respect their deadlines. The implementation choice has an impact on WCET and performances, since it impacts the memory access contentions, especially for multi-core targets. In the past, we proposed a shared variable based approach [10], and a distributed approach on the many-core Intel Single-chip Cloud Computer (SCC) [11], which has been introduced directly in *preludec* 1.6.0.

More recently, we have specialized this generic distributed approach for the AER model [12] on an ARM-based multi-core very similar to the one of the challenge. This compilation scheme based on AER for synchronous programs has been integrated in the WCC compiler [13] for WCET purpose. The code associated to any runnable is split into three C functions:

- **phase A:** data consumed by an imported node is first read during an acquisition phase A, i.e. copied from the memory where it is stored to a local variable;
- **phase E:** the imported node executes in isolation from other cores and without accessing any shared resources;
- **phase R:** data consumed by other imported nodes is written in the LRAM during this phase.

The phases are then scheduled according to a non-preemptive schedule computed off-line. The model assumes that when a flow is produced by an imported node and consumed with a direct communication, phase R of the producer must execute before phase A of the consumer.

b) (Manual) Adaptation for the challenge: The AER implementation together with the PRELUDE communication protocol fits the challenge semantics. However, it is not optimal in terms of memory size and memory contentions. We propose an AER adaptation that reduces these overheads. First, we split the phase A into two sub-phases:

- *A_let*: labels from other tasks are copied at task activation;
- *A_implicit*: labels from other runnables of the same task are copied at the beginning of a runnable execution.

Code 2 (C wrapped code):

```
extern int Label_744, Label_586, Label_1394,
  Label_1779;
extern int Label_2492, Label_3543, Label_4704,
  Label_9944;

static int lr_Label_744_r5ms0, lr_Label_586_r5ms0,
  lr_Label_1394_r5ms0, lr_Label_1779_r5ms0;

static int lw_Label_2492,
  lw_Label_3543, lw_Label_4704, lw_Label_9944;

void r5ms0_a_let ()
{
  read_int(&lr_Label_744_r5ms0, Label_744);
  read_int(&lr_Label_586_r5ms0, Label_586);
  read_int(&lr_Label_1779_r5ms0, Label_1779);
}

void r5ms0_a_implicit ()
{
  read_int(&lr_Label_1394_r5ms0, lw_Label_1394);
}

void r5ms0_e()
{
  vz_filter_step(lr_Label_744_r5ms0,lr_Label_586_r5ms0,
                 lr_Label_1394_r5ms0, lr_Label_1779_r5ms0,
                 &lw_Label_2492, &lw_Label_3543,
                 &lw_Label_4704,&lw_Label_9944);
}
void r5ms0_r()
{
  write_int(&Label_2492, lw_Label_2492);
  write_int(&Label_3543, lw_Label_3543);
  write_int(&Label_4704, lw_Label_4704);
  write_int(&Label_9944, lw_Label_9944);
}
```

Let us illustrate the wrapping of *Runnable_5ms_0* shown in figure 2. The runnable comes with 4 functions, namely *r5ms0_X* with $X \in$ {a_let, a_implicit, e, r}. In the *A_let* phase, each consumed label *Label_l* is copied in a local variable named *lr_Label_l_r5ms0*. In the *R* phase, each produced label *Label_l* is copied from a local variable named *lw_Label_l*. In the *A_implicit* phase, each consumed label *Label_l* is obtained simply by copying the variable *lw_Label_l*, allocated as a static variable on the local core, in a local variable named as *lr_Label_l_r5ms0*. During the E phase, only local variables are used (lr, lw). We define the following implementation rules:

- **spatial rule**: for each core c, for each *Label_l*,
  - For each consumer runnable *rX* executing on c, there is a local copy *lr_Label_l_rX*;
  - If *Label_l* is mapped on c and produced by runnable r, there is local copy *lw_Label_l* managed by r;
- **temporal rule**:
  - At the activation of a task t, all *rX_a_let* are called in sequence for all runnables *rX* belonging to t;
  - At the start of a runnable r, *r_a_implicit* is first called;
  - At the termination of a task, all *rX_r* are called in sequence.

B. Cost and optimization

In this section, we discuss the communication overheads. First, in terms of memory, the total allocated memory is:

$$\sum_{l\in[1,10000]} (\textit{nb\_consumers(Label\_l)} + \textit{producer(Label\_l)}) \times \textit{size(Label\_l)}$$

where *nb_consumers(Label_l)* is the number of runnables consuming $Label\_l$ and *producer(Label_l)* is equal to 0 if *Label_l* is an external input and 1 otherwise.

We can optimize the memory allocation according to the following rules:

- on each core c, for each label *Label_l*, for each task t consuming *Label_l* and not producing it, there is a single local copy of *lr_Label_l_t*;
- For each task t, for each label *Label_l* produced in t, all consumer runnable will directly use *lw_Label_l*.

Then the required memory will only be:

$$\sum_{l\in[1,10000]} (\textit{nb\_task\_consumers(Label\_l)} \ + 1) \times \textit{size(Label\_l)}$$

where *nb_task_consumers(Label_l)* is the number of tasks consuming *Label_l*.

Now, we consider the memory contentions. Let $n \div k$ denote that the rest of the division of n by k is 0. We propose the following scheduling strategy to reduce contentions:

- Pre-compute off-line the set of activation dates p over the hyper-period $H = lcm(period(t))$. More precisely,

$$\{p \leq H | \exists t,\ p \div period(t)\}$$

- Create for each core c the new tasks c_let_p and c_r_p with the highest priority;
  - For all runnable r executing on c such that $p \div period(r)$, c_let_p calls in sequence all its a_let functions;
  - At the termination $p - \epsilon$, c_r_p calls in sequence the r phases of terminating runnables.
- At each date p, we first execute the phases $c0\_let\_p$ on core 0, then on core 1, then on core 2, then on core 3. Since access to distant memory is done in isolation, we can compute the latency of a read. Thus the code for core i would be: wait($d_{0,p} + \ldots + d_{i-1,p}$); $c0\_let\_p$; wait($d_{i+1,p} + \ldots + d_{3,p}$).

Thanks to this mechanism, the phases E and R will be fully in isolation. Another approach would be to compute the maximal conflicts encountered by each piece of code.

V. Conclusion

In this paper, we presented the main principles of a translation from AMALTHEA to PRELUDE and discussed alternative communication implementations for the challenge. We also presented the C code generation from PRELUDE. Using the translator in combination with the *preludec* compiler actually provides an AMALTHEA to C compilation chain. It took us 2 days to understand the challenge, 4 days to implement the translator from AMALTHEA to PRELUDE, .5 days to analyze latencies and 1 day to design the adaptations and optimizations presented at the end of the paper. An URL for downloading the translator will be posted on the WATERS forum by the end of the submission week.

References

[1] AMALTHEA, "An open platform project for embedded multicore systems," available: http://www.amalthea-project.org.

[2] A. Hamann, D. Ziegenbein, S. Kramer, and M. Lukasiewycz, "Demo abstract: Demonstration of the fmtv 2016 timing verification challenge," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016, pp. 1–1.

[3] A. Hamann, D. Dasari, S. Kramer, M. Pressler, F. Wurst, and D. Ziegenbein, "Waters industrial challenge 2017," 2017.

[4] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: a time-triggered language for embedded programming," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 84–99, 2003.

[5] E. Farcas, C. Farcas, W. Pree, and J. Templ, "Real-time component integration based on transparent distribution," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–7, 2005.

[6] J. Forget, F. Boniol, D. Lesens, and C. Pagetti, "A multi-periodic synchronous data-flow language," in *Proceedings of the 11th IEEE High Assurance Systems Engineering Symposium (HASE'08)*, 2008.

[7] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens, "Multi-task implementation of multi-periodic synchronous programs," *Discrete Event Dynamic Systems*, vol. 21, no. 3, pp. 307–338, 2011.

[8] R. Wyss, F. Boniol, C. Pagetti, and J. Forget, "End-to-end latency computation in a multi-periodic design," in *28th Symposium On Applied Computing (SAC'13)*, Coimbra, Portugal, Apr. 2013.

[9] C. Sofronis, S. Tripakis, and P. Caspi, "A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling," in *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, 2006, pp. 21–33.

[10] J. Forget, F. Boniol, D. Lesens, and C. Pagetti, "A real-time architecture design language for multi-rate embedded control systems," in *Proceedings of the 25th ACM Symposium on Applied Computing (SAC'10)*. ACM, 2010, pp. 527–534.

[11] W. Puffitsch, E. Noulard, and C. Pagetti, "Mapping a multi-rate synchronous language to a many-core processor," in *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'13)*, 2013, pp. 293–302.

[12] G. Durrieu, M. Faugère, S. Girbal, D. Gracia Pérez, C. Pagetti, and W. Puffitsch, "Predictable flight management system implementation on a multicore processor," in *Proceedings of the 7th Conference on Embedded Real Time Software and Systems (ERTS'14)*, 2014.

[13] C. Pagetti, H. Falk, D. Oehlert, and A. Luppold, "Space and time aware compilation of synchronous programs," in *Under submission*, 2017.