WATERS Industrial Challenge 2017 with Prelude
Frédéric Boniol, Julien Forget, Claire Pagetti

To cite this version:
Frédéric Boniol, Julien Forget, Claire Pagetti. WATERS Industrial Challenge 2017 with Prelude. 8th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2017), Jun 2017, Dubrovnik, Croatia. hal-03690252

HAL Id: hal-03690252
https://hal.science/hal-03690252
Submitted on 8 Jun 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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 MALTHEA semantics [1] and of the questions raised by the WATERS challenge [2], [3].

A. Software model

The software model 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 900µs at minimum and 1000µs 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 1. There are two types of periodic tasks:

  1. 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;
  2. 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 ISR with \(X \in [1, 11]\). These tasks are all of type Preemptive (as defined above).

B. Platform model

The architecture considered for the challenge is shown below.

Each core \(X\) has a frequency of 200MHz and is associated with a private local memory (scratchpad memory) denoted as \(LRAM_X\). 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 MALTHEA 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 1. 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 1 (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

\[\text{[1]}\text{ChallengeModel_w_Label_Stats_fixedLabelMapping_App4mc_y072.amxmi.}\]
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).

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 $k \in \mathbb{N}\setminus\{0\}$), 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 100000\(\mu\)s. The rate of an

### Table I

<table>
<thead>
<tr>
<th></th>
<th>1ms</th>
<th>6666(\mu)s</th>
<th>2ms</th>
<th>5ms</th>
<th>10ms</th>
<th>20ms</th>
<th>30ms</th>
<th>45ms</th>
<th>246ms</th>
<th>14ms</th>
<th>43ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of runnables</td>
<td>41</td>
<td>146</td>
<td>27</td>
<td>22</td>
<td>306</td>
<td>306</td>
<td>45</td>
<td>246</td>
<td>14</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>Allocated core</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Sporadic

<table>
<thead>
<tr>
<th></th>
<th>ISR10</th>
<th>ISR5</th>
<th>ISR6</th>
<th>ISR4</th>
<th>ISR8</th>
<th>ISR7</th>
<th>ISR11</th>
<th>ISR9</th>
<th>ISR1</th>
<th>ISR2</th>
<th>ISR3</th>
</tr>
</thead>
<tbody>
<tr>
<td>min</td>
<td>700</td>
<td>900</td>
<td>1100</td>
<td>1500</td>
<td>1700</td>
<td>4900</td>
<td>5000</td>
<td>5000</td>
<td>9500</td>
<td>9500</td>
<td>9500</td>
</tr>
<tr>
<td>max</td>
<td>800</td>
<td>1000</td>
<td>1200</td>
<td>1700</td>
<td>1800</td>
<td>5050</td>
<td>5100</td>
<td>6100</td>
<td>10500</td>
<td>10500</td>
<td>10500</td>
</tr>
<tr>
<td>Number of runnables</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Allocated core</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Logical Execution Time (LET)**

**write labels**

**terminate**
the rate of its producer, so for instance, $Label_{9944}$ has period 5000µs.

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

```plaintext
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_2560 : 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_4744 = 2 . ( 0 fby Label_586 ) = 2 , 0 fby Label_1394 , Label_1779 /’ 10 /’ 3 );
```

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 100000µs, 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 $\sim$ and $\ast$ : $\sim q$ produces a flow $q$ times slower, while $\ast q$ 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 * 3 = 1500µs.

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, lef 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 $C = (r_1 \overset{x_1}{\rightarrow} r_2 \overset{x_2}{\rightarrow} \ldots \overset{x_n}{\rightarrow} r_n)$ 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

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

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

$$C_{LET} = (r_1 \overset{x_1}{\rightarrow} fby / \sim 2 \rightarrow r_2)$$

As shown figure 2, the first instance of the chain begins at t=0 and goes through $x_1^1$ (the first occurrence of $r_1$), $x_1^2$ (which
is delayed by one period compared to \( r_1 \) and terminates at \( t' = 400ms \) at the end of \( r_2 \) (the second occurrence of \( r_2 \)). Its latency is \( 400ms \). Similarly, the second instance of the chain begins at \( t = 100ms \), goes through \( r_2 \), \( x_2 \) and terminates also at \( t' = 400ms \) at the end of \( r_2 \). Its latency is \( 300ms \). Thus, the worst case latency of the chain is \( 400ms \), and it is achieved every \( 200ms \).

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:

\[
C_{\text{direct}} = (r_1 \ x_1 \ / \ 2 \rightarrow r_2)
\]

In that case, the first instance of the chain goes through \( r_1 \), \( x_1 \), \( r_2 \). The maximal end date of \( r_2 \) being \( 200ms \), the maximal latency of the first instance is \( 200ms \). The second instance of the chain begins with \( r_2 \) at \( t = 100ms \) and terminates at the end of \( r_2 \) 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. 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 \( \text{EffectChain}_1 = (r10ms149 \xrightarrow{\text{L}3423} r10ms243 \xrightarrow{\text{L}3968} r10ms272 \xrightarrow{\text{L}2276} r10ms107} \) (where \( r10msX \) is a shorthand for \( \text{Runnable}_{10msX} \) and \( \text{LY} \) for \( \text{Label} \_Y \)). 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 \( \text{fby} \) operator before consumption by \( r10ms107 \). All the other communications in the chain are direct. The PRELUDE model of this chain is:

\[
\text{EC1}_{\text{prelude}} = (r10ms149 \xrightarrow{\text{L}3423} r10ms243 \xrightarrow{\text{L}3968} r10ms272 \xrightarrow{\text{L}2276} \text{fby} \rightarrow r10ms107).
\]

The maximal latency of the chain is simply one period, plus one period for each \( \text{fby} \) in the chain, that is \( 20ms \) in total.

### C. Effect Chain 2

\( \text{EffectChain}_2 = (r100ms7 \xrightarrow{\text{L}4258} r10ms19 \xrightarrow{\text{L}2157} r2ms8) \). 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 \( \text{EffectChain}_2 \) is:

\[
r100ms7 \xrightarrow{\text{L}4258} \text{fby} \ast *10 \rightarrow r10ms19 \xrightarrow{\text{L}2157} \text{fby} \ast *5 \rightarrow r2ms8.
\]

The PRELUDE end-to-end analysis method 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 \( r100ms7 \) is \( r2ms8^{86} \);
- More generally, the earliest occurrence of \( r2ms8 \) which is impacted by \( r100ms7 \) is \( r2ms8^{86+50(i-1)} \).

The latency of the chain is a constant value \( 112ms \), i.e., the latest possible end date of \( r2ms8 \) minus the earliest possible start date of \( r100ms7 \). 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

\( \text{EffectChain}_3 = (r700us800us3 \xrightarrow{\text{L}4576} r2ms3 \xrightarrow{\text{L}646} r50ms36) \). The first runnable of \( \text{EffectChain}_3 \) 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 \( \text{EffectChain}_3 \), \( r700us800us3 \) 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 \( \text{EffectChain}_3 \) is:

\[
r700us800us3 \xrightarrow{\text{L}4576} \text{fby} / \ast 5 \ast *2 \rightarrow r2ms3 \xrightarrow{\text{L}646} \text{fby} / \ast 25 \rightarrow r50ms36.
\]

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\textsuperscript{122}, which starts at 96.8ms and which impacts r50ms36\textsuperscript{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 \( (r700us800us3 \xrightarrow{L6076} / \ast 5 \ast 2 \rightarrow r2ms3 \xrightarrow{L646} / \ast 25 \rightarrow r50ms36) \). The maximal latency is achieved for r700us800us38\textsuperscript{2}, which starts at 48.8ms and which impacts r50ms36\textsuperscript{3} ending at most at 150ms. The resulting latency is 150 \textendash 48.8 = 101.2ms. The LET implementation is only 2ms greater than the implicit implementation.

<table>
<thead>
<tr>
<th>Effect chains</th>
<th>EC_1</th>
<th>EC_2</th>
<th>EC_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max latency</td>
<td>20ms</td>
<td>112ms</td>
<td>103.2ms</td>
</tr>
</tbody>
</table>

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 [8]. 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 direct communications); 2) tasks respect their deadlines. The implementation choice has an impact on WCET and performances, since it impacts the memory access contentsions, especially for multicore 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 prelude 1.6.0.

More recently, we have specialized this generic distributed approach for the AER model [12] on an ARM-based multicore 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:

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

```c
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, &lr_Label_3543, &lw_Label_4704, &lw_Label_9944);
}

void r5ms0_e() {
  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 ∈ \{aₐₗₜₑ, aᵢ mplᵢ cᵢ t, e, r\}. In the Aₐₗₜₑ phase, each consumed label Label_1 is copied in a local variable named lr_Label_1_r5ms0. In the R phase, each produced label Label_1 is copied from a local variable named lw_Label_1. In the Aᵢ mplᵢ cᵢ t phase, each consumed label Label_1 is obtained simply by copying the variable lw_Label_1, allocated as a static variable on the local core, in a local variable named lr_Label_1_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:

$$
\sum_{t \in [1,10000]} (nb\_consumers(Label_l) + producer(Label_l)) \times size(Label_l)
$$

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:

- For 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 $lr_{Label_l} r$;
- 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:

$$
\sum_{t \in [1,10000]} (nb\_task\_consumers(Label_l) + 1) \times size(Label_l)
$$

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 $c.a_{let} p$ and $c.a_{r} p$ with the highest priority:
  - For all runnable $r$ executing on $c$ such that $p \div period(r)$, $c.a_{let} p$ calls in sequence all its $a_{let}$ functions;
  - At the termination $p - \epsilon$, $c.a_{r} p$ calls in sequence the $r$ phases of terminating runnables.

- At each date $p$, we first execute the phases $a0.a_{let} p$ 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($d_{0,p} + \ldots + d_{i-1,p}$); $c0.a_{let} p$; 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 MALTHEA 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 MALTHEA to C compilation chain. It took us 2 days to understand the challenge, 4 days to implement the translator from MALTHEA 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