Logical Execution Time Implementation and Memory Optimization Issues in AUTOSAR Applications for Multicores

Alessandro Biondi, Paolo Pazzaglia, Alessio Balsini, Marco Di Natale
Scuola Superiore Sant'Anna, Pisa, Italy
E-mail: {alessandro.biondi, paolo.pazzaglia, alessio.balsini, marco.dinatale}@sssup.it

***Abstract*—The 2017 FMTV challenge has been extended to consider with better precision the details of the HW platform; the need for synthesis and optimization methods, and also introduces for consideration the Logical Execution Time (LET) model. In this paper we highlight some of the problems and issues that relate to the implementation of the LET model and then we present and compare two approaches for optimizing the placement of the labels in memory, including the time analysis methods that will be used for the system. The paper concludes with a discussion on the next steps and other fundamental issues that are related to the general problem of optimizing the placement of computations in multicore platforms.**

I. Introduction

The FMTV 2017 challenge consists of a timing analysis problem in which the AUTOSAR model of a set of cooperating tasks in a fuel injection application is deployed onto a 4-core platform. The objective of the challenge is to study the possible conditions for the implementation of the Logical Execution Time (LET) model in the runnables communication and to provide methods for the analysis of the memory allocation of the communication variables (labels) in the model. The variables need to be allocated in the available memory spaces (local and global) of the AURIX microcontroller.

The LET model was introduced as part of the Giotto programming paradigm [1] as a method to eliminate output jitter and provide time determinism in the code implementation of controls. In essence, the LET delays the program output of a task (or any function executed inside the task) at the end of the task period, trading delay for output jitter.

The analysis of the LET implementation is performed under the assumption of the mechanisms and tools that are typical of an AUTOSAR process. In AUTOSAR, the computation functions are called runnables and the communication implementation is provided by a layer of code automatically generated by tools: the Run Time Environment or RTE. The consideration of the AUTOSAR process greatly influences the implementation options. For the LET implementation we discuss two possible implementation options: one that is compatible with the current mechanisms and tools of the standard AUTOSAR process, the other with a simple extension to the AUTOSAR implicit communication implementation (providing for a much more efficient solution).

For the label placement optimization problem, we discuss a simple method to bound the worst case latency when real-time tasks access a memory bank possibly competing with other tasks. Using the provided bound for the memory latency, we developed two algorithms to solve the problem: a simple Genetic Algorithm solution and an MILP formulation.

We provide the results of these two optimization methods with an additional discussion on how to tackle the runnable placement optimization problem, which is most likely the most relevant design issue for a system like this.

II. System Model and Notation

The challenge model is a case study from the Amalthea EU project and it is in large part compliant with the AUTOSAR metamodel. As such, the model adopts from AUTOSAR definitions and most of the semantics for the activation and communication of functions (runnables in AUTOSAR). An attempt at the formal characterization of the challenge model is the following.

Task and runnable model. A task τ_i is composed of an ordered sequence of n_i runnables $\rho_{i,1}, \ldots, \rho_{i,n_i}$, each of which has its execution time $\mathcal{C}_{i,j}$, defined as a truncated Weibull distribution. For the purpose of worst-case analysis, the worst-case execution time (WCET) $C_{i,j}$ and a best-case execution time $c_{i,j}$ may be computed from the distribution $\mathcal{C}_{i,j}$. Each runnable $\rho_{i,j}$ may read or write labels from a set $\mathcal{L} = \{l_1, l_2, \ldots, l_p\}$. Each label l_i is characterized by a type and a size (an integer number of bytes). Each task is defined by a tuple $\tau_i = \{\mathcal{C}_i, T_i, \mathcal{L}_i, D_i\}$, where $\mathcal{C}_i$ is the execution time distribution of the task, simply computed as the convolution of the distribution of the task runnables (by extension C_i and c_i are the worst and best case task execution times); T_i is the period or minimum inter-arrival time of the task activation event(s); $\mathcal{L}_i$ denotes the set of labels accessed by τ_i; and D_i is the relative (to the activation time) deadline. When applicable, relative deadlines are constrained to be smaller than or equal to periods, i.e., $D_i \leq T_i$. $N_{i,v}$ denotes the number of times τ_i accesses label $\ell_v \in \mathcal{L}_i$.

In the worst case (the reasoning also applies to other types of analysis but we only discuss the worst-case analysis here), the execution time $C_{i,j}$ of a task may be expressed as the sum of the runnable execution times in the task. The execution times provided with the challenge do not include the execution cost to read and write the memory labels.

The scheduling of each task is also controlled by its scheduling mode (cooperative or preemptive) and its priority π_i, with preemptive tasks having higher priority than cooperative tasks, and cooperative tasks only preempting each other at runnable boundaries.

We denote as $R_{i,j}$ the worst-case response time of the j-th runnable of task τ_i, while $r_{i,j}$ denotes its best-case response time. $hp^P(i)$ and $hp^C(i)$ denote the set of preemptive and cooperative tasks, respectively, having priority greater than τ_i. We denote as $hp(i) = hp^P(i) \cup hp^C(i)$ the union of the two disjoint sets.

Platform model. There are $m = 4$ identical processors $P_1, \ldots, P_m$. There are four local memories $M_1, \ldots, M_m$ (one for each core) and a global memory M_{m+1}. The platform disposes of a crossbar switch that provided point-to-point communication channels between each core and each memory. Concurrent accesses to memory are arbitrated with a FIFO queue.

Task and label allocation model. The allocation of the tasks is *fixed* and given in the provided Amalthea model. $P(\tau_i)$ denotes the processor to which τ_i is allocated; $\Gamma(P_k)$ denotes the set of tasks allocated to processor P_k; and $\Gamma(\tau_i)$ denotes the set of tasks allocated to the same processor to which τ_i is allocated. An allocation of the labels is also provided in the Amalthea model. The following notation is used when discussing the label allocation. $\mathcal{M}_k$ denotes the set of labels allocated to memory M_k. λ^R denotes the delay introduced by task during a conflict to a remote memory, while λ^L denotes the delay introduced by task during a conflict to its local memory. The maximum time needed to access a word into memory M_x from processor P_x is denoted by

$$\Delta_{k,x} = \begin{cases} \delta^L & \text{if } k = x \text{ (local memory)}, \\ \delta^R & \text{otherwise.} \end{cases}$$

where δ^R denotes the time needed to access a remote memory (GRAM or local RAM of another processor), and δ^L denotes the one needed to access the local memory M_k. We assume $\lambda^L = \delta^L$ and $\lambda^R = \delta^R$. Based on the challenge information, the memory access and conflict times are $\lambda^L = 1$ cycle and 5 ns; $\lambda^R = 9$ cycles and 45 ns;

Finally, with respect to a given allocation of the labels, the time $MA_{i,v}$ needed to access label $\ell_v \in \mathcal{L}_i$ from task τ_i is defined as $MA_{i,v} = \Delta_{k,x}$ where $P_k = P(\tau_i)$ and M_x is the memory in which ℓ_v is allocated. The same terminology applies to runnables.

A. The LET model of execution

The Logical Execution Time model was probably first presented as part of the Giotto project [1]. The objective of the LET model is to add time determinism to periodic computations by eliminating the output jitter.

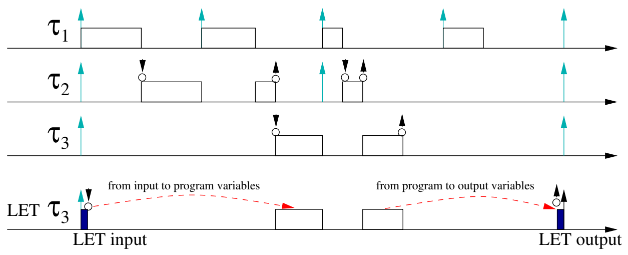
τ_1
τ_2
τ_3
LET τ_3
from input to program variables
from program to output variables
LET input
LET output

Fig. 1. The LET model of execution.

The LET execution model can be summarized as depicted in Figure 1. In the figure, the output of task τ_2 (denoted by the upward arrow at the end of the box representing the task execution) has a significant jitter. Because of variable interference from τ_1, it occurs late in the first task instance and much earlier in the second. The LET solution is shown in the bottom timeline for task τ_3 (taken as an example). The input of the task data is performed at the task activation, and the output is performed at the end of the task execution period. All task inputs are stored in local variables at the task activation. Similarly, all outputs need to be stored in local variables and will be actually output only by the LET code at the end of the cycle. This requires to allocate memory for local variables mirroring all input and output variables.

Several mechanisms can be used to enforce the LET synchronization of input and output operations, as a hardware or software implementation. In essence, LET is a sample and hold mechanism with synchronized execution of the input and output part. As such it is not too dissimilar to mechanisms used to enforce flow preservation in the implementation of synchronous models [2], [3]. When LET is implemented in SW (the HW implementation would not affect the design analysis or the challenge goals) assuming a typical AUTOSAR development model (for more information on the related assumptions please refer to [4], [5]), there are two main options:

- LET is implemented as part of the Run-Time Environment (RTE) with support from the basic SW;
- LET is implemented at the application level by a set of dedicated runnables.

In both cases, since it requires a dedicated set of tasks (and the corresponding scheduling configuration), the LET implementation will most likely be modeled as a set of RTE or application-level input and output tasks. Since it is required that the input and output operations of these tasks are executed as close as possible to the start and end of period instants, these tasks should be characterized by a very short WCET and a very high priority level. This has several implications that are further discussed in the implementation section.

- There may be more than one task dedicated to the input and output sampling for LET execution. If this is the case, then these tasks will internally preeempt each other and the design of this additional set of tasks may be a subproblem in its own.
- The execution of the output task (or action) at the end of the period may be very difficult to obtain wth conventional scheduling strategies ("as late as possible execution" is typically not supported). In this case the output task needs to be actually executed at the beginning of the next cycle, possibly in conjunction with the corresponding input task (in a back-to-back fashion).

III. Timing analysis with memory contention

This section presents a response-time analysis for tasks under partitioned fixed-priority scheduling that explicitly accounts for the delay introduced by *memory accesses* and their corresponding *memory contention*. The same analysis can be extended to runnables in a seamless manner.

Under the assumption of constrained deadlines, the worst-case response time of a task τ_i is bounded by the least positive fixed-point of the following recurrent equation:

$$R_i = W_i + \sum_{\substack{\tau_j \in hp(\tau_i) \\ \tau_j \in \Gamma(\tau_i)}} \left\lceil \frac{R_i}{T_j} \right\rceil W_j + MC_i(R_i) \qquad (1)$$

where $W_i = C_i + \sum_{\ell_v \in \mathcal{L}_i} N_{i,v} \cdot MA_{i,v}$ (i.e., the worst-case execution time of the task plus the cost for accessing its labels) and $MC_i(R_i)$ represents the delay due to memory contention incurred by τ_i and all the high-priority tasks, which *transitively* affect the response time of the task under analysis.

Since memory contention is resolved according to the FIFO policy, a safe bound on the term $MC_i(R_i)$ can be obtained by simply *inflating* the terms W_i to account for $m-1$ contentions for each memory access. However, this approach may lead to excessive pessimism, thus resulting in very coarse upper-bounds on the response times.

In this work, we use the *inflation-free analysis* [6], [7] to bound the blocking times for a synchronization protocol for multiprocessor systems. Inflation-free analysis explicitly accounts for each memory access that may originate a contention while task τ_i (under analysis) is pending. To this end, we proceed by bounding the maximum number of accesses $NRA_{k,x}(t)$ issued by tasks executing on the remote processors $P_k \neq P(\tau_i)$ to each memory M_x in an *arbitrary* time window of length t, that is

$$NRA_{k,x}(t) = \sum_{\tau_j \in \Gamma(P_k)} \sum_{\ell_v \in \mathcal{L}_j \cap \mathcal{M}_x} \left\lceil \frac{t + R_j}{T_j} \right\rceil N_{j,v}. \qquad (2)$$

Note that the above equation considers the sum over *all* the tasks allocated to P_k as they can produce memory contention independently of their priority (FIFO arbitration). The term $\lceil (t + R_j)/T_j \rceil$ is a safe bound on the maximum number of jobs of $\tau_j \in \Gamma(P_k)$ in any time window of length t [6], [7].

Similarly, we also bound the number of accesses $NLA_{i,x}(t)$ issued by the local processor $P(\tau_i)$ to each memory M_x in an *arbitrary* time window of length t while τ_i is pending, that is

$$NLA_{i,x}(t) = N_{i,v} + \sum_{\substack{\tau_j \in hp(\tau_i) \\ \tau_j \in \Gamma(\tau_i)}} \sum_{\ell_v \in \mathcal{L}_j \cap \mathcal{M}_x} \left\lceil \frac{t}{T_j} \right\rceil N_{j,v}. \qquad (3)$$

Due to the FIFO arbitration and the fact that the memory accesses are non-interruptible, it follows that **(i)** each memory access issued by a remote processor can delay *at most* one access issued by the local processor and **(ii)** each access issued by the local processor can be delayed by *at most* one remote access *per processor*; hence the following bound holds:

$$MC_i(t) = \sum_{P_k \neq P(\tau_i)} \sum_{x=1}^{m+1} \min\left\{NRA_{k,x}(t), NLA_{i,x}(t)\right\} \cdot \Lambda_{k,x}, \qquad (4)$$

where the term $\Lambda_{k,x}$ is provided to distinguish the delay introduced by the memory contentions as a function of each pair (P_k, M_x), and is defined as

$$\Lambda_{k,x} = \begin{cases} \lambda^L & \text{if } k = x \text{ (local conflict)}, \\ \lambda^R & \text{otherwise.} \end{cases}$$

Equation (4) can be used in Equation (1) to bound the response times of the tasks. The term $NRA_{i,k,x}(t)$ depends on the response time of the tasks allocated to the remote processors: this additional recursive dependency can be addressed with an iterative loop in which Equation (1) is solved for all the tasks until *all* the response-time bounds R_i converge. Such an iterative loop starts with $R_i = C_i$ for all tasks τ_i.

IV. Implementing and Analyzing the Logical Execution Model in AUTOSAR

This section discusses solutions for the implementation and the analysis of the LET model in AUTOSAR.

A. LET implementation as part of the RTE generation

The discussion on the implementation of the LET model cannot be undertaken without the joint consideration of the typical AUTOSAR model for the generation of task code and the execution of runnables. AUTOSAR has two models of communication. In the explicit model (top of Figure 2) the copy of the data in the communication variables is performed at the time each runnable invokes the communication API function. The implementation of the LET model in this case, would require the definition of two LET runnables that act as proxies for the read and write operations. The reader and writer runnable should execute according to the pattern defined in the following section.

In the implicit model, even if a read or write operation is invoked by the runnable in the middle of its execution, the actual code implementing the read from and write into the shared variables is automatically generated as part of the RTE code at the beginning and at the end of the runnable code. The result of the read operation is sampled at the beginning of the runnable execution and then stored in a local variable for the duration of the runnable execution. Similarly, the write value is locally stored in a variable and then output by RTE code after the runnable execution (shown in the middle of Figure 2, the darker rectangles before and after the runnable execution represent the RTE code). If the RTE generation tools are not modified, the LET implementation in this case would require yet another set of runnables and an additional set of variables, which is clearly a source of additional memory and time overhead.

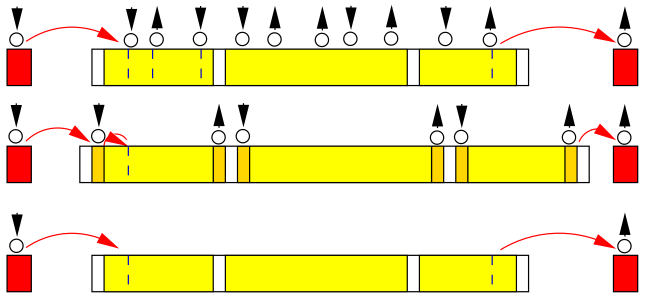
Fig. 2. Code implementation of the LET model of execution with explicit or implicit communication.

However, it is relatively straightforward to see how a simple modification of the RTE generation process for the implicit communication model would be the best solution. A simple RTE generation option could result in moving the input and output to the LET tasks rather than the runnable boundaries. The RTE generator could generate the LET input and output tasks together with the other RTE-generated code.

B. Reader and Writer Tasks, Definition and Analysis

The general architecture and scheduling of the input and output tasks in a LET model is discussed at lenght in [8]. In their work, input and output tasks are scheduled together with mode change tasks assuming a time-triggered schedule with jitter constraints for the input and output operations.

In the case the tasks are implementing AUTOSAR runnables, the input and output tasks can serve all the tasks executing at the same rate. Of course, if a task has runnables executing at multiples of the task period, the corresponding input and output sections can be skipped when unnecessary. The input and output LET tasks may be scheduled using the AUTOSAR time-triggered mode when available, in order to ensure the output task is executed right before the end of the period of the tasks it serves. In a priority-based schedule, like the one assumed in the challenge, the output and input tasks may be joined and executed back to back at the beginning of each period. To arbitrate among the input and output operations for the tasks executing at different rates there are several options between two extremes: one is to have a single LET task executing at the greatest common divisor of the task periods (most likely inefficient for the challenge model). The other extreme is to have a LET task for each period. Of course partial groupings may be possible and may be more efficient in some cases.

In our challenge model we assume a LET task for each period. LET tasks have higher priorities than the other tasks (to enforce the precedence constraints) and we assign their relative priorities according to Rate-Monotonic.

V. The challenge model

The provided challenge model has a set of special characteristics that affect the analysis and optimization methods and strongly characterize the obtained results.

First and foremost, the tasks are allocated on the cores accroding to a specific strategy. The first core only executes interrupt service routines. The same is true for the fourth core, that also executes a 10ms periodic task. The second core only executes (most likely details are not provided) a variable rate task (triggered at predefined angles in the engine rotation), and a very high rate 1ms periodic task. All the other periodic tasks are executed by the third core.

Using the worst-case execution times provided in the model, the system is definitively in *overload* with the following per-core utilizations: 0.97, 1.336, 1.068 and 1.179. We attribute the large overload in the second core to the modeling strategy adopted for the `Angle_sync` task. We deem such a task to be an engine-triggered task with variable activation rate and speed-dependent adaptive beahavior. The provided model mostly likely considers a minimum inter-arrival time for the maximum engine speed, and a WCET computed for the most time consuming operating mode. This is pessimistic and explains the overload. The explicit consideration of the *adaptive variable-rate* (AVR) task model [9] would improve the analysis precision.

To optimize the system configuration based on the worst case behavior, we need to restore feasibility and to definie a suitable cost function. To restore feasibility, we consider the mean execution times in place of the worst-case.

As a cost function, after the discussion on the forum and based on the recommendations of the organizers, we selected the maximum normalized (with respect to the deadline) response time of the tasks, as in the function

$$\mathcal{C} = \max_{\tau_i \in \Gamma(P_k), \forall P_k} \frac{R_i}{D_i}, \tag{5}$$

VI. End-to-end latency

We adopted the analysis provided in [9] to compute the end-to-end latency of the effect chains.

However, in order to consider the influence of sporadic computational activities, the best-case response time computation for a runnable $\rho_{i,j}$ must be corrected as follows:

$$r_{i,j} = \sum_{h=1}^{j} c_{i,h} + \sum_{k \in hp(i)} N_k^{act} c_k \tag{6}$$

where N_k^{act} is defined as:

$$N_k^{act} = \begin{cases} \lceil \frac{r_{i,j}}{T_k} \rceil - 1 & \text{for periodic tasks;} \\ 0 & \text{for sporadic tasks.} \end{cases} \tag{7}$$

The ISRs have been considered as sporadic tasks: this choice has been adopted because the maximum inter-arrival time of the ISRs provided in the Amalthea model seems too close to the minimum one.

In this paper, the computation of end-to-end latencies is provided only for the case of explicit communication. The case for LET-based communication is straightforward (modulo some minimal interference caused by high-priority LET tasks).

1) Effect Chain 1: In the effect chain 1, all runnables belong to the same task (*Task_10ms*, allocated to core 3). As there is backward communication between the third and the fourth runnable, this adds a one cycle delay until the last datum is read. Therefore, the worst-case end-to-end latency of this effect chain by L2F can be computed as:

$$L_1^{L2F} = T_{10ms} + R_{10ms,107} \tag{8}$$

This result is valid also when considering the L2L semantics. As for the F2F semantics, the analysis needs to consider a one cycle delay for the first runnable, that is:

$$L_1^{F2F} = 2T_{10ms} + R_{10ms,107}. \tag{9}$$

2) Effect Chain 2: Runnables in this chain belong to different tasks with different rates. In this case, the end-to-end latency calculation should also consider the over-sampling effect between pairs of consecutive runnables. By the L2F semantics, we obtain:

$$\begin{aligned} L_2^{L2F} &= R_{100ms,7} + \min(T_{10ms} - r_{10ms,19}, T_{100ms}) \\ &+ R_{10ms,19} + \min(T_{2ms} - r_{2ms,8}, T_{10ms}) + R_{2ms,8} \end{aligned}$$

As for the F2F semantics, due to the over-sampling effect, there are no input overwritings, hence the end-to-end latency is simply given by:

$$L_2^{F2F} = L_2^{L2F} + T_{100ms}.$$

Finally, the end-to-end latency computation for the L2L semantics requires to verify Condition (8) from [9], for any pair of consecutive runnables.

$$L_2^{L2L} = R_{100ms,7} + \widehat{n}_1 \cdot T_{10ms} - r_{10ms,19} + R_{10ms,19} \\ + \widehat{n}_2 \cdot T_{2ms} - r_{2ms,8} + R_{2ms,8}.$$

3) Effect Chain 3: Also in this case, runnables belong to different tasks with different rates. Task periods have increasing values, leading to an under-sampling effect.

By the L2F semantics we obtain:

$$L_3^{L2F} = R_{700/800us,3} + \min(T_{2ms} - r_{2ms,3}, T_{700/800us}) + \\ R_{2ms,3} + \min(T_{50ms} - r_{50ms,36}, T_{2ms}) + R_{50ms,36}\ \mu s$$

Due to the sporadic nature of the first runnable, we assume $T_{700/800us} = 800\ \mu s$ in order to maximize latency.

The end-to-end latency by the F2F semantics requires to add one cycle delay with respect to L2F and to verify Condition (8) from [9] for any pair of consecutive runnables.

$$L_3^{F2F} = T_{700/800us} + \overline{n}_1 \cdot T_{700/800us} + R_{700/800us,3} + \\ \overline{n}_2 \cdot T_{2ms} + R_{2ms,3} + R_{50ms,36} = 75559\ \mu s.$$

Finally, the end-to-end latency for the L2L semantics is equal to the L2F case, because no output is overwritten due to the under-sampling effect.

VII. Optimizing the placement of Memory labels

This section discusses possible approaches to compute the optimal placement of label and label copies (for LET) in memory. We tried two possible solutions (MILP and Genetic Algorithm) for the case of explicit communication and LET-based communication.

A. Genetic algorithm

Due to the extremely large set of labels to be positioned, a metaheuristic has been chosen to find a sufficiently good solution. A Genetic Algorithm (GA) approach has been found to be the most suitable candidate for this problem. Hereafter the structure of the algorithm is briefly presented.

Using the common nomenclature for GAs, we define a possible label placement as an *individual* $\mathcal{I}$. Each individual is encoded as an ordered string of 10000 RAM ids (*genes*), representing the position of each label in the memories. The set of individuals (called *population*) is firstly initialized randomly. At every step we evaluate each individual with a *fitness* function that is the cost function identified for the challenge: $F(\mathcal{I}) = \mathcal{C}(\mathcal{I})$ i.e., the maximum normalized response time among all tasks with the labels positioned as in $\mathcal{I}$.

At every iteration, the solutions are reordered by following their fitness values $F(\mathcal{I})$ (the smaller the better) and divided in three subsets: **(i)** reproductive survivors (*elite*), **(ii)** non-reproductive survivors, and **(iii)** extinct individuals. The next generation is created by selecting random couples of *parents* between the elite group, that generate new individuals using a *crossover* function: this strategy swaps randomly selected blocks of genes between the parents and save the resulting solutions as a new individuals. At the same time, extinct individuals are removed from the population.

The exploration of new individuals in the solution space is guaranteed by using also a certain number of *mutation* functions, which randomly change a limited (and casually chosen) number of genes in the population. Each function has different activation probabilities and consist in random changing label positions, moving labels from one memory to another one, and spreading labels from one memory to all the others. On the other hand, in order to maintain a sort of *elitism*, a (small) number of clones of the best solutions are copied in the next generation without mutating.

B. MILP formulation

The formulation of the problem as a *mixed-integer linear program* (MILP) required facing with several challenges that cannot be discussed here due to lack of space. For the same reason, the complete MILP formulation, with the corresponding proofs of the constraints, is omitted. However, it is worth discussing two approximations that have been applied to the analysis of Section III in order to express the response-time bounds in a linear form.

First, instead of searching for the least positive fixed-point of Equation (1), we adopted the approximated response-time analysis proposed by Park and Park in [10]. Under rate-monotonic scheduling, the authors showed (with an experimental evaluation) that their approximation introduces an extremely limited error ($\leq 1\%$) with respect to the exact response-time analysis. By extending Theorem 4 in [10] to cope with the analysis presented in Section III, the response time of a task τ_i (if schedulable with a constrained deadline) is bounded by

$$R_i = \min_{t \in S_i} \left\{ r_i = W_i + \sum_{\substack{\tau_j \in hp(\tau_i) \\ \tau_j \in \Gamma(\tau_i)}} \left\lceil \frac{t}{T_j} \right\rceil W_j + MC_i(t) : r_i \leq t \right\} \tag{10}$$

where $S_i = \left\{ \bigcup_{\substack{\tau_j \in hp(\tau_i) \\ \tau_j \in \Gamma(\tau_i)}} \left\lfloor \frac{T_i}{T_j} \right\rfloor T_j, T_i \right\}$.

This approximation results very useful for encoding the response-time bound into a MILP as **(i)** it allows getting rid of the typical *integer* variables that are needed to model the term with the ceiling of Equation (1) (note that the terms in the set S_i are all *constants*, hence that term is in turn a constant); and **(ii)** it allows avoiding the need for *quadratic* constraints, which is implied by the fact that the terms W_j must be optimization variables (note that their values depend on the label placement).

Second, to avoid requiring additional integer variables, the term $NRA_{k,x}(t)$ of Equation (2) has been over-approximated by replacing R_j with D_j.

Finally, it is worth mentioning that we leveraged on lower-bounds of the response times to reduce the number of MILP variables (and the corresponding constraints) that must be provided to encode Equation (10). The lower-bounds have been computed by accounting for one clock cycle for each access to a label, which corresponds to the best case where labels allocated in local memory and no contention is possible. Such bounds allows reducing the elements into the set S_i.

A taste of the MILP formulation. A binary variable $A_{v,x}$ has been provided to each couple of label ℓ_v and memory M_x, with the interpretation that $A_{v,x} = 1$ iff ℓ_v is allocated to M_x. Such variables have been constrained such that $\sum_{x=1}^{m+1} A_{v,x} = 1$ holds for each label $\ell_v \in \mathcal{L}$.

The actual worst-case execution time W_i of a task τ_i allocated to processor P_k can then be expressed with the following linear constraint:

$$W_i \geq C_i + \sum_{x=1}^{m+1} \sum_{\ell_v \in \mathcal{L}_i} N_{i,v} \cdot \Delta_{k,x} \cdot A_{v,x}.$$

The objective of the MILP formulation is to *minimize* Equation (5).

VIII. Experimental Evaluation

A. LET model implementation

For the purposes of this challenge, the LET model has been implemented only for the runnables involved in effect chains; the only jitter-sensitive parts of the system.

The effect chains are composed of 10 runnables and 7 labels. The approach proposed in this paper for the LET implementation requires adding high-priority LET tasks dedicated to copying and writing data implied in the effect chain. Runnables belonging to the same task need only one collective task, thus only 5 LET tasks must be added to the system. As every label needs two local copies (one for reading one for writing), the total number of labels is increased by 14.

B. Optimal label placement: MILP formulation

The MILP formulation has been solved with IBM CPLEX on a machine equipped with an 8-core processor Intel(R) Xeon(R) E5-2609v2 running at 2.50GHz. The solver is able to immediately found (very first iterations) a feasible solution for the label placement.

For the case of explicit communication, the optimal placement is computed in about 1 hour and 20 minutes, but the solver is able to provide a sub-optimal solution with a guaranteed gap to the optimum lower than 1% in *less than two minutes*. The value of the objective function for the optimal solution is 0.8505. Recomputing the objective function with the analysis presented in Section III we obtain 0.849555: this result confirmed the effectiveness of the approximation adopted in the MILP formulation. Using the label placement provided in the Amalthea model of the challenge, the objective function is 1.32634: hence, our solution provides an improvement that is larger than 35%.

The label placement for the optimal solution is illustrated in Figure 3 (dark bars). As shown in the graph, most of the labels are allocated in the local memory of the first core (LRAM0).

For the case of LET communication, the optimal placement is computed in about 1 hour and 50 minutes. Similarly to the first case, the solver is able to provide a sub-optimal solution with a guaranteed gap lower than 1% in *less than seven minutes*. Using the label placement provided in the Amalthea model of the challenge, and placing the 14 labels required for implementing the LET communication as in our solution (as they do not exist in the challenge data), the objective function is the same for the case of explicit communication.

Surprisingly, the label placement is completely different from the one computed for the case of explicit communication (see Figure 3, light bars), as most of the labels are allocated in

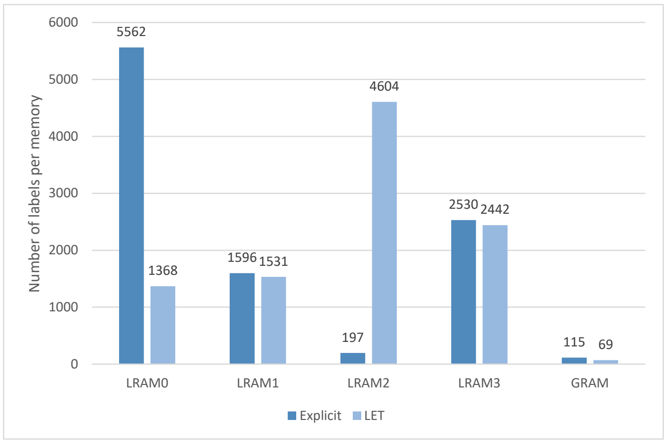

Fig. 3. Placement of the labels for the case of explicit (dark bars) and LET-based (light bars) communication.

the local memory of core 2 (LRAM2). This result is attributed to the fact that the analysis is dominated by the `Angle_sync` task and the 1ms periodic task on core 1: as a consequence, the optimization algorithms will mostly optimize the labels used by these tasks, while the placement of the other labels is almost indifferent (with possibly few exceptions due to memory conflicts).

C. Optimal label placement: Genetic Algorithm

The Genetic Algorithm approach has been implemented in C++ and executed on an Intel(R) i7 4790K running at 4GHz. The population has been set to 200 individuals $\mathcal{I}$, all initialized randomly, and the termination condition of the algorithm has been defined to complete 30000 iterations. A feasible solution (with objective function < 1) is usually reached after the first 2000 iterations. The algorithm takes approximatively 5 seconds per iteration, with a completion time of less than 40 hours. For each communication semantic we produced approximately 20 distinct simulations.

For the case of explicit communication, the best results obtained with the Genetic Algorithm is an objective function of 0.85161, while for the LET communication the value is 0.85173. The solutions obtained are only slightly worse than the ones found with the MILP formulation, but required a large running time to be computed.

D. End-to-end latencies

End-to-end latencies of the effect chains have been computed using the optimal label placement that has been obtained with the MILP formulation. The best and worst case response times of the runnables under explicit communication are reported in Table I, while the corresponding latencies of the effect chains are reported in Table II.

Under LET-based communicaiton, once a label placement that guarantees the task schedulability is found (as done by the proposed optimization algorithms), the end-to-end latencies can be computed in a straightforward manner. Further investigation may target the integration of the end-to-end latencies as constraints in the MILP formulation, with the objective of computing label placements that guarantee specific timing constraints related to the effect chains.

TABLE I
RESPONSE TIMES (μSEC) FOR THE RUNNABLES IN THE EFFECT CHAINS UNDER EXPLICIT COMMUNICATION.

Runnable name	**Best RT**	**Worst RT**	**Period**
Runnable10ms149	2077.68	4118.33	10000
Runnable10ms243	3315.99	6328.08	10000
Runnable10ms272	3644	7175.57	10000
Runnable10ms107	1367.16	2745.87	10000
Runnable100ms7	152.335	13820.5	100000
Runnable10ms19	298.715	650.74	10000
Runnable2ms8	55.465	117.12	2000
RunnableSporadic700us800us3	17.815	27.11	700
Runnable2ms3	20.025	42.385	2000
Runnable50ms36	1070.3	13089.6	50000

TABLE II
END-TO-END LATENCIES FOR THE EFFECT CHAINS UNDER EXPLICIT COMMUNICATION.

Chain index	**Latency type**	**Latency value (μs)**
1	L2F	12746
1	F2F	22746
2	L2F	26234
2	F2F	126234
2	L2L	154234 ($\widehat{n}_1 = 11$ and $\widehat{n}_2 = 5$)
3	L2F	15959
3	L2F	75559 ($\overline{n}_1 = 2$ and $\overline{n}_2 = 30$)

IX. Discussion of Results and Conclusions

In this work, we presented the methods and algorithms to provide an AUTOSAR-compliant implementation of the *logical execution time* (LET) model and to optimize the placement of memory labels in the system.

We believe the methods are applicable, perform quite satisfactorily, and can provide early indication on the quality of a given mapping configuration (or find a very effective one). However, there are several limitations in the challenge configuration that affect the results.

First and foremost, given the cost function and the current task set (and its core allocation), the analysis is dominated by the `Angle_sync` task and the 1ms periodic task on the same core. This means that the optimization algorithms will in practice mostly optimize the labels used by these tasks and be almost indifferent to the others (this explains why the LET solution and the solution without LET appear quite different with very similar cost values).

Also, this reifорces the concept, already illustrated in the previous challenge submission, that the runnable and task placement dominates all other considerations and is therefore our primary objective for future research. A discussion of possible forumlations and the additional issues and problems can be found in [11].

References

[1] T. A. Henzinger, C. M. Kirsch, M. A. A. Sanvido, and W. Pree, "From control models to real-time code using giotto," in *Control Systems Magazine, IEEE*, 2003.

[2] G. Wang, M. Di Natale, and A. Sangiovanni-Vincentelli, "Improving the size of communication buffers in synchronous models with time constraints," in *IEEE Transactions on Industrial Informatics*, vol. 5 (3), 2009, pp. 229–240.

[3] H. Zeng and M. Di Natale, "Mechanisms for guaranteeing data consistency and flow preservation in autosar software on multi-core platforms," in *6th IEEE International Symposium on Industrial Embedded Systems (SIES), Vasteras, Sweden*, June 2011.

[4] M. Di Natale and A. Sangiovanni-Vincentelli, "Moving from federated to integrated architectures in automotive: The role of standards, methods and tools," in *Proceedings of the IEEE*, vol. 98 (4), 2010, pp. 603–620.

[5] A. Ferrari, M. Di Natale, G. Gentile, G. Reggiani, and P. Gai, "Time and memory tradeoffs in the implementation of autosar components," in *Design, Automation and Test in Europe Conference. DATE'09*, April 2009.

[6] A. Wieder and B. Brandenburg, "On spin locks in AUTOSAR: blocking analysis of FIFO, unordered, and priority-ordered spin locks," in *RTSS'13*.

[7] A. Biondi and B. Brandenburg, "Lightweight real-time synchronization under P-EDF on symmetric and asymmetric multiprocessors," in *ECRTS'16*.

[8] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: A time-triggered language for embedded programming." in *Proc. International Workshop on Embedded Software (EMSOFT), volume 2211 of LNCS*, Springer, Ed., 2001, pp. 166–184.

[9] A. Biondi, M. D. Natale, and G. Buttazzo, "Response-time analysis for real-time tasks in engine control applications," in *Proceedings of the 6th International Conference on Cyber-Physical Systems (ICCPS 2015)*, Seattle, Washington, USA, April 14-16, 2015.

[10] M. Park and H. Park, "An efficient test method for rate monotonic schedulability," *IEEE Transactions on Computers*, vol. 63, no. 5, 2014.

[11] A. Biondi, M. Di Natale, Y. Sun, and S. Botta, "Moving from single-core to multicore: initial findings on a fuel injection case study," in *SAE Technical Paper, SAE Conference, Detroit, USA*, April 2016.