Supporting Logical Execution Time in Multi-Core POSIX Systems

Davide Bellassai[1,2], Alessandro Biondi[1], Alessandro Biasci[2], and Bruno Morelli[2]

[1]Scuola Superiore Sant'Anna, Pisa, Italy
[2]Huawei Research Center, Pisa, Italy

Abstract—Safety-critical automotive applications require predictable and deterministic execution to not miss the timing requirements. Logical Execution Time (LET) is a paradigm already established in the automotive industry to improve the predictability and correctness of time-critical applications. Despite LET being already part of the AUTOSAR Classic standard, no prior work has addressed the design of this model on POSIX-based operating systems, which will be the base of next-generation automotive Electronic Control Units (ECUs). This paper proposes and discusses possible LET design approaches for these novel systems. Different implementations are then evaluated and compared through the WATERS Challenge automotive application running on a multi-core heterogeneous hardware platform.

Keywords— LET, POSIX, Multicore, Logical execution time, Real-time, Embedded systems

1 Introduction

For decades, the automotive domain has been a very conservative industry, with software functionalities operated by simple Electronic Control Units (ECUs). However, the recent increase in the number and complexity of in-vehicle functionalities are driving a technological shift towards the integration of multiple functionalities on novel heterogeneous multi-core platforms. During this transition and integration, it is of paramount importance to preserve the non-functional requirements (e.g. predictability, determinism, causality) of the safety-critical functionalities.

Logical Execution Time (LET) is a paradigm originally proposed in the context of the GIOTTO framework [1] to eliminate output jitter and thus guarantee time determinism in control applications. LET, in essence, delays the output of each task (of its Runnables, in the automotive domain) at the end of the task's period, thus reducing the jitter and increasing the predictability of the communication. This paradigm can be also used to maintain the causality of legacy code when moving from single-core to multi-core platforms. Specific designs also allow to enforcement of freedom-from-interference (FFI) by removing or controlling memory contention [2]. More recently, the original paradigm has been extended to scenarios where the time for communication is not negligible. To this aim, System-Level LET (SL-LET) [3] has introduced the concept of "timezones" and an additional "interconnect" task to formalize the communication delay.

AUTOSAR (AUTomotive Open System ARchitecture) [4] is an European consortium born in 2004 to create a standard and interoperable software architecture for automotive ECUs. The original specification (named AUTOSAR Classic [5]) provides a reference model and programming API for ECUs executing a tiny hard real-time operating system (RTOS) and signal-oriented communications. Recently, there has been a growing interest in the LET paradigm in the automotive field, where software is heavily composed of control applications and determinism is a key factor to guarantee safety. As a result, the AUTOSAR Classic standard provides a model and API for the LET paradigm [6], leaving each vendor free to design its implementation.

The recent exponential increase in complexity of automotive systems, due to the integration of novel functionalities like assisted or autonomous driving, has forced the consortium to create an additional standard, called AUTOSAR Adaptive [7]. The software stack for this kind of ECUs consist of a general-purpose OS based on the POSIX API (e.g. Linux) and a set of C++ libraries to support multi-thread applications. In addition, the original signal-oriented paradigm has been replaced by a modern service-oriented architecture (SoA) [8]. This novel standard, however, does not include the LET paradigm due to the inherent difficulties in implementing this model on a dynamic general-purpose OS based on the POSIX standard [9].

Contribution. In this paper, we propose and analyze two different approaches to designing the LET paradigm on dynamic POSIX-based operating systems. In particular, we propose a design at both kernel space and user space. After formalizing and explaining the design of the communication protocol, the synchronization mechanism, and the involved data structures, we implement the two designs and make a performance comparison on a real multi-core platform running an automotive application (namely, the WATERS Challenge [10]).

Paper structure. The paper is organized as follows. Section 2 provides the needed background information. Section 3 explains the model and the notations used in the paper. Section 4 illustrates the design and the formalization of the proposed approach, including the communication mechanism and the dynamic protocols. Section 5 presents the implementations of the LET model for both user and kernel space, illustrating all the main functionalities through pseudo-code. Section 6 provides the evaluation of the two implementations and a performance comparison. Finally, Section 7 discusses the related work, and Section 8 concludes.

2 Background and Problem Definition

Many control applications are still designed using the Bounded Execution Time (BET) programming model, where the response time of a task may vary with each execution but never exceeds

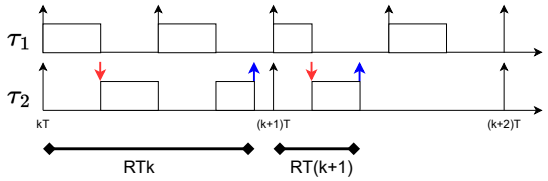

Fig. 1: The BET model of execution. Red and blue arrows denote input/output operations respectively performed by tasks.

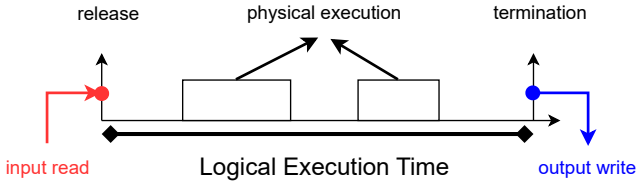

Fig. 2: Logical Execution Time paradigm

the Worst-Case Response Time (WCRT) [11]. One problem that affects this model is of course the jitter of the response time. In the presence of jitter, causality in the task chain can be broken, leading to a degradation of the performance of the control [12]. Even worse, whether jitter occurs or not depends on the actual execution time of the involved tasks and other conditions (e.g. system load), making the system behavior not deterministic. Moreover, the problem of jitter increases with the number of cores, since more tasks can execute in parallel and interfere with each other on shared hardware resources like memory.

Figure 1 shows an example of execution using the BET model where two periodic tasks are scheduled using rate monotonic scheduling [13]. Comparing the two activations of τ_2, we can see that the output of task τ_2 experiences a significant jitter due to the interference of τ_1, which has a variable execution time and executes at higher priority.

Another issue of the BET model is the loss of causality that can occur when moving from single-core to multi-core platforms, which can lead to an increase of an end-to-end latency of a given cause-effect chain [11].

2.1 Logical Execution Time

In the original LET proposal [14, 15], the execution of tasks is predictable and deterministic, with preservation of the order of execution. Determinism is applied also for communication and actuation times.

Figure 2 shows how the LET paradigm works. Input and output operations are performed at the beginning and at the end of the period, respectively. One approach to implementing such semantics is to use additional local variables whose content is filled/copied only at the period boundaries. The computation of the task within the period is done only using local variables.

It is easy to infer that, in the presence of a task chain, the LET paradigm causes an increase in end-to-end latency. Following the work presented by Davare et al. [16] and considering that LET the output is provided at the end of the task period (with respect to the semantic described in [2]), in a chain p of tasks the end-to-end latency (as defined [16]) is bounded by

$$E2E_p = \sum_{k:\tau_k \in p} 2T_k, \tag{1}$$

where the notation $\tau_k \in p$ indicates that task τ_k belongs to chain p and T_k is the period of τ_k. In this work, we use the

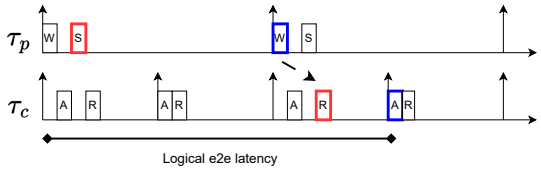

Fig. 3: Schedule of LET communication with GIOTTO semantics. The producer task has a period T_p=2T_c. The marked squares in red and blue are operations that contribute to end-to-end latency respectively for sensing/read and actuation/write operations. Legend: S = sensing, R = read, A = actuation, W = write.

original GIOTTO semantics [1], which assumes a negligible time (i.e. zero logical time) for input/output operations [15, 17]. However, in real implementations, communication phases must be scheduled for execution. The order of execution of input/output operations leads to different time properties. To ensure determinism, this semantics specifies an order of execution for the input and output operations using LET, which can be recapped as follow:

- Perform data writing and control output operations;
- Perform input and data read operations.

This order is applied for every instance of each task. However, if more than one task activates at the same time, input and output operations are grouped and executed together.

Figure 3 illustrates a scheduling example of LET communication between a producer τ_p and a consumer τ_c with GIOTTO semantics. All communication operations are performed at the beginning of each period, giving precedence to writing and actuation operations. The end-to-end latency with which the system reacts to the control input is deterministic — i.e., it is independent of the tasks' response time and equal to the sum of the periods of both T_p and T_c, as long as the tasks complete their execution before the release of their next instance, ignoring the time required to perform input/output operations.

The LET paradigm also represents a possible solution to restore causality when moving legacy software from single-core to multi-core platforms [18, 19]. Moreover, it is possible to exploit the additional benefit of scheduling precisely in time the accesses to the communication variables and to perform buffer optimization to improve the efficiency and the performance of the LET communication [2].

2.2 LET on POSIX Systems

As already mentioned, LET has been already standardized in the AUTOSAR Classic specification [6] and, accordingly, many different implementations nowadays exist. However, the automotive industry is putting more and more interest in the emerging AUTOSAR Adaptive standard (AP) [7]. This novel specification aims at addressing the requirements of modern automotive software but does not yet provide sufficient mechanisms to guarantee determinism in publish-subscribe communications. Indeed, the first attempt at improving the determinism of execution and communication (namely, DeterministicClient) might be soon removed from the specification due to overengineering and poor adoption.

In this work, we tackle the challenge of supporting the LET paradigm in dynamic software systems, defining the formalism for dynamic protocols required to handle the dynamicity of the system and ensure determinism. In particular, the major challenges are related to **(i)** allow tasks to join the LET paradigm at runtime, **(ii)** provide a core synchronization mechanism able to guarantee the respect of LET semantics, and **(iii)** ensure that

LET communication operations are performed at the right times to guarantee deterministic behavior. The implementation is designed for POSIX-compliant systems, leaving the possibility of porting the approach to AP.

3 System model

We focus on a system where real-time tasks communicate through the producer-consumer paradigm [20] exchanging messages based on topics. In our model, an application consists of a set of n periodic real-time tasks $\tau_i = (C_i, T_i, D_i, R_i)$, each one characterized by a Worst-Case Execution Time (WCET) C_i, a period T_i, a relative deadline D_i and a response time R_i. The application is executed on a platform that comprises m cores denoted as $P = \{P_1, P_2, ..., P_m\}$, and a global memory M. According to producer-consumer relationships, tasks can also be organized into *chains*, where the k-th task in the chain produces data to be consumed by the $(k+1)$-th one in the same.

We assume that tasks are executed on a real-time POSIX operating system that supports the following scheduling policies: partitioned Earliest Deadline First (pEDF), First-In-First-Out (FIFO), Round Robin (RR), and hierarchical scheduling. In particular, the real-time tasks are scheduled using pEDF. For each core, there are two instances of FIFO schedulers, one of which is dedicated to the scheduling of LET tasks and has higher priority than pEDF. Other non-real-time tasks are scheduled using the other instance of the FIFO scheduler, which has lower priority w.r.t. all the other schedulers, except for the dummy one.

It is important to notice that, in the absence of any task to schedule, the kernel will execute the dummy process, which will eventually cause a process context switch.

Tasks are mapped into different cores at the design time of the application and task migration is not available. The communication model used in this work is based on the principles of the topic-based publish-subscriber model [21]. Producer and consumer tasks share a topic and communicate through the use of messages implemented as data structures. For each topic, it is possible to have only one producer and multiple consumers.

Table 1 summarizes the notation used in the paper.

Sym.	Description
$b_{i,l}$	Private buffer of task i for topic l
$Q_{(i,t)}$	Activated tasks at time t on core i
L_{r_i}	Set of reg. tasks to LET paradigm on core i
$C(t)$	Cores involved in LET op. at time t
L_b	Set of dereg. task to LET paradigm
Γ_L	LET task implemented for user-space
γ_i	Alias of task i
L_{Γ_L}	LET task private set of registered tasks
$S_o(t)$	Reg. set for output at time t
m_{reg}	Mutex for reg./dereg. to L_{r_i}
$S_i(t)$	Reg. set for input at time t
m_{ch}	Mutex for reg./dereg. to L_b
F_r	Flag registration notification
$spin_b$	Spinlock for barrier synchronization
F_d	Flag deregistration notification
$spin_s$	Spinlock for cores synchronization

TABLE 1: List of the main symbols.

4 System Design

In this section, the design of the system is explained, starting from the communication mechanisms adopted to the formalization of all the protocols introduced to integrate the LET paradigm into POSIX-compliant dynamic systems.

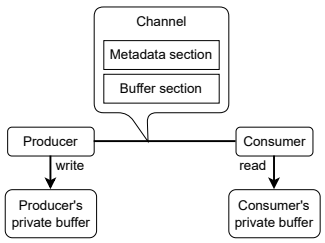

Fig. 4: Interaction between producer and consumer based on a communication channel.

4.1 Communication Mechanism

Communication between tasks is essential whenever there is a task chain with data-dependency constraints. For each topic l, a task τ_i is provided with a private buffer $b_{i,l}$. Each time a producer must publish new data, it writes the message on its private buffer. In the same manner, consumers read from their private buffer for new messages. The design of the communication mechanism is inspired by the implicit communication mechanism present in Classic AUTOSAR [5], where runnables read from and write into global variables only at the beginning and at the end of their execution respectively. All other operations are performed on local copies, avoiding interfering with other runnables. The code implementing the read from and write into shared global variables is generated as part of the RTE code at the beginning and the end of the runnables. In the proposed approach, private buffers act as local copies of the variables that can be accessed anytime during the execution of tasks, without incurring any interference from other tasks. Since tasks can access only their private buffers, a mechanism is required to transfer the message from the producer to consumers' private buffers. Hence, communication channels are used.

As shown in Figure 4, a channel is a logical entity, representing a topic, acting as a communication bridge between tasks. When new data are available on the producer's private buffer, the message is copied into the message buffer owned by the channel during the LET output phase. On the other hand, during the LET input phase, the message is copied from the channel's message buffer to the consumers' private buffers. Each channel is represented as a data structure containing the information explained below.

4.1.1 Metadata Section

This section contains information related to the channel itself and all the producer and consumer tasks that adhere to the communication. The information contained in the metadata section is the following:

- **Topic**: it is used as a unique identifier of the channel in the system. Each time tasks want to join a specific channel, the topic must be specified with a string;
- **Status**: it specifies whenever new data are made available from the producer. It is used to optimize the communication phase in the presence of oversampling of the consumers, avoiding accessing aged data. The status of the channel is updated by the producer only if new data are produced;
- **Message size**: it specifies the data size of the message exchanged between producer and consumers;
- **Timestamp**: it specifies when consumers are ready to accept new data. It is used to optimize the communication phase, avoiding the producer to overwrite unused data.
- **Members**: it specifies how many tasks use the channel;
- **Producer CPU**: it specifies the CPU where the producer is executing. It is used to check anytime a consumer

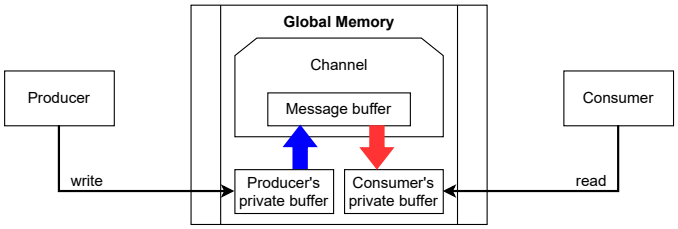

Fig. 5: Communication protocol. Blue and red arrows represent LET output and LET input phases, respectively.

joins the channel if it executes on the same CPU of the processor, allowing hence to enable intra-core communication;

- **Buffers references queue**: it keeps track of all the buffers related to the tasks that joined the channel. This queue is used to manage intra-core communication;
- **Consumers references queue**: it keeps track of all the consumers that adhere to the channel;
- **Producer reference**: it keeps track of the producer for that channel.

4.1.2 Buffer Section

This section contains the references to the buffers of the channels:

- **Message buffer**: it is a reference to a buffer, owned by the channel, where the message is stored. It can be accessed only during the communication phases. Not the producer, nor the consumers are allowed to access directly this buffer;
- **Spare buffer**: it is a reference to a buffer used for intra-core communication to overcome communication problems when producers and consumers have non-harmonic periods.
- **Unused buffers list**: it is used to keep track of all unused tasks' private buffers that are registered to the channel and adhere to intra-core communication.
- **Updated buffer pointer**: it is a pointer to the last updated buffer by the producer.

Figure 5 represents a communication flow between producer and consumer tasks. During their executions, tasks can access anytime their private buffer to write output or read input. In such a way, private buffers emulate the behavior of local variables of the tasks, even if they are not part of the task's stack. Tasks involved in communication are unaware of LET operations, hence do not care to notify other tasks for communication.

4.2 Intra-core communication

Depending on the hardware, cores can be equipped with a private scratchpad memory that can be accessed without any contention from other cores [22]. It is possible to exploit a core's private memory to improve efficiency in communication between tasks that execute on the same core, avoiding unnecessary global memory accesses. If the architecture does not provide the core's private memory, it is possible to emulate such behavior to improve the intra-core communication mechanism, even if buffers are allocated in global memory.

4.2.1 Pointer-swap protocol

Communication between tasks scheduled on the same core is operated by the pointer-swap protocol, which aims at mimicking zero-time and zero-copy communication policies [23].

The protocol from [23] was extended to support non-harmonic tasks, introducing a spare buffer for each channel.

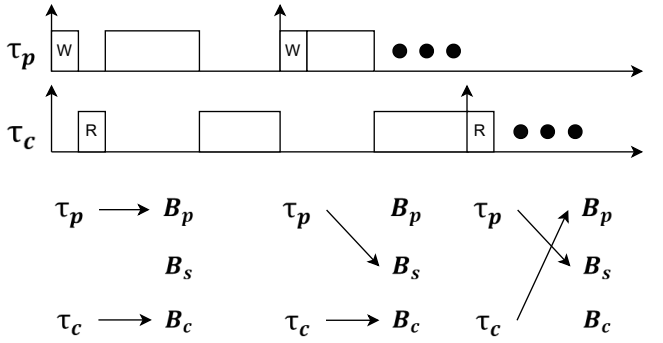

Fig. 6: Extended pointer-swap protocol with channel's spare buffer. Black arrows denote which buffer is accessed by each producer and consumer task's instance. Legend: W = write, R = read.

Each time a LET communication must be performed, pointers to the buffers used by tasks, avoiding expensive operations to copy data.

The key idea of the protocol is to allow the producer task to always access a buffer that is currently unused, while consumer tasks can access always the latest buffer updated by the producer (provided that it is not currently used by the producer itself). Figure 6 shows how the proposed extended pointer-swap protocol works. Consider a producer τ_p with period T_p associated with a buffer B_p and a consumer τ_c with period T_c associated with a buffer B_c. Denote with B_s the spare buffer of the communication channel. Assume also that τ_p is scheduled with higher priority w.r.t. τ_c and both tasks activate at time $t = 0$. During the first instance, both the producer and the consumer access their corresponding buffers without any interference. In the second instance of the producer task, its buffer is swapped with the channel's spare buffer, leaving the previously used buffer free. Finally, in the second instance of the consumer task, its buffer is swapped with the latest buffer freed by the producer, which allows access to the latest updated data. Note that multiple consumers can access the same buffer without compromising the data integrity since consumers can only perform read operations.

The intra-core communication protocol is formalized as follows:

R1 At its creation, the spare buffer is empty as the list of unused buffers and the updated buffer pointer.

R2 During the LET output phase, the producer's private buffer is swapped with the spare buffer of the channel, and referenced by the last updated buffer pointer.

R3 During the LET input phase, the consumer's private buffer is marked as empty and swapped with the channel's spare buffer. If the spare buffer is already marked as empty, the consumer's private buffer is inserted into the unused buffer list, referring to the last updated buffer as the new consumer's private buffer.

It is possible to generalize the protocol for multicast communications in a task chain with n tasks providing a lower bound for required buffers. Consider that all consumers, whose period is less or equal to the one of the producer, share the same last updated buffer. Define with N_c the number of such consumers, hence the number of buffers required is $n - N_c + 1$. For this purpose, the unused buffer list of the channel keeps track of all the consumers' private buffers that are not used in any LET communication operations.

4.3 Dynamic Membership Protocol

Unlike static systems, where the configuration occurs before execution, on POSIX-compliant systems the configuration can change at run-time. For this reason, a set of protocols is required that allow tasks to dynamically join the LET paradigm and ensure the respect of LET semantics when performing communication operations. The key idea is to allow tasks to register themselves to the LET paradigm anytime during their execution. Registration on behalf of other tasks is not permitted. In the same manner, each task can only deregister itself from the LET paradigm at any time. For the registration, tasks must register also at least one topic used for the communication. However, tasks can repeat the registration process multiple times during their execution, to register different topics at different time instances. On the opposite, deregistration can be performed only once as it deregisters the task to the LET paradigm as well as all its related topics. The Dynamic Membership Protocol (DMP) manages all requests from tasks that want to join the LET paradigm, providing registration and deregistration functionalities. The set L_{r_i} is used to keep track of all the tasks, running on core i, that adhere to LET. Whenever a task τ sends a join request, a related alias γ is created and used as a new entry for the registration set. Logically, the alias is a reference to the task which can be implemented as a data structure that represents a registration element composed of different fields described as follows:

- **Task name**: it specifies the name of the task to which the alias is referred;
- **activation time**: it specifies the activation time of the referred task;
- **period**: it specifies the period of the referred task;
- **channels list**: it specifies the list of all the channels to which the task is linked. The list is sorted to place first all the channels where the task is the producer, then all the channel where the task interact as a consumer;
- **buffers list**: it specifies the list of all the private buffers that are related to the task. Note that the order of the private buffers follows the one used for the channels list. In particular, for each channel on the channels list, also a private buffer related to that topic exists. The relation between the channel and the topic is specified as the position number of the list;
- **Current CPU**: it specifies the CPU where the task executes;
- **Registration number**: it refers to a global progressive number assigned to each task that is registered to the LET paradigm.

To exploit parallel execution during the registration phase, there exists a set L_{r_i} for each core. To complete the registration phase, tasks must also register the communication channels related to the topics that they treat. For this purpose, the set L_b is used to keep track of all communication channels created by registered tasks. Unlike the set of registered tasks L_{r_i}, the set L_b is global for all the cores and shared during the registration phase. Hence, synchronization mechanisms must be taken into account when managing the set of registered channels. Below the formalization of the registration and deregistration protocols follows.

The registration protocol complies with the following statements:

- **R1** In the first place, the alias of the task is created, together with its private buffers;
- **R2** Then, the communication channels are linked to the alias. With link operation, the reference of the channel is stored in the $channel\_list$ field of the alias, allowing direct access to the channel. Otherwise, if the channel is not existing, it is created and registered to L_b. registration fails due to the lack of free memory available, the registration phase is aborted;
- **R3** The alias created in the first step is registered to L_{r_i}. If a failure occurs, the registration of the channels to L_b committed in the previous step is revoked and the registration phase is aborted;
- **R4** Mutual exclusion mechanisms are used to protect the registration of the task's alias to L_{r_i} from concurrent execution of tasks running on the same core, while the registration of communication channels to L_b from concurrent execution between all tasks in the system.

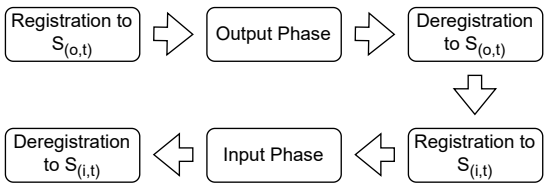

Fig. 7: Illustration of the six phases of inter-core synchronization protocol.

The deregistration protocol complies with the following statements:

- **R1** In the first part of the deregistration phase, the alias of the task is removed from L_{r_i}, and all its private buffers are destroyed;
- **R2** All communication channels are then unlinked and, if needed, deregistered from L_b;
- **R3** Mutual exclusion mechanisms are used to protect the deregistration of tasks' aliases to L_{r_i} from concurrent execution on the same core, while the deregistration of communication channels to L_b from concurrent execution between all cores.

4.4 Inter-Core Synchronization Protocol

As stated in chapter 2, GIOTTO semantics is required to preserve causality and ensure a deterministic behavior [1]. Since tasks belonging to the same task chain can be executed in parallel on different cores, the respect for GIOTTO semantics is ensured through an inter-core synchronization protocol. Due to the dynamicity of the system, it is not possible to predict offline when cores will perform the LET communication phase. Denote with $S_o(t)$ and $S_i(t)$ the set of cores that, at time t, aim to perform the output and input operations, respectively. Define with $LET_{cycle}(t)$ the LET cycle as the amount of time it took to perform read and write operations on all cores that started at time t. In particular, define with $\Delta W_{(i,t)}$ the amount of time used to perform write operations for all tasks activated on core i at time t, and with $\Delta R_{(i,t)}$ the amount of time used to perform read operations for all tasks activated on core i at time t, hence:

$$LET_{cycle}(t) = \sum_{i=0}^{M-1} \Delta W_{(i,t)} + \Delta R_{(i,t)}. \tag{2}$$

As depicted in Figure 7, the inter-core synchronization protocol can be divided into six phases which comply with the following rules:

- **R1** At the beginning of the $LET_{cycle}(t)$, both $S_o(t)$ and $S_i(t)$ must be empty, i.e. $S_o(t) = \emptyset$ and $S_i(t) = \emptyset$;
- **R2** Before starting the output phase, each core $P_i \in C(t)$ must register to $S_o(t)$. Once registration is done, the output phase can be undertaken;
- **R3** Before starting the input phase, each core $P_i \in C(t)$ must register to $S_i(t)$ and deregister to $S_o(t)$. Despite this order of execution is not mandatory, it enforces GIOTTO semantics.

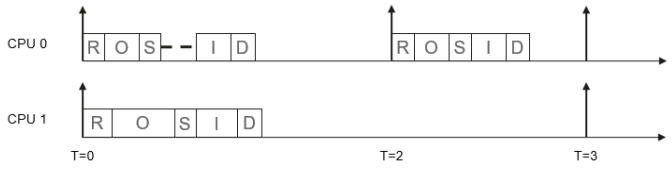

Fig. 8: Example of schedule on two cores with the inter-core synchronization protocol. Task running on processor P_1 has activation at times 0 and 2, while task running on processor P_2 only at time 0. Legend: R = registration to $S_o(t)$, O = output phase, S = switch phase, I= input phase, D = deregistration to $S_i(t)$.

- **R4** Input phase can start only when $S_o(t) = \emptyset$ and $S_i(t) = C(t)$. Otherwise, cores already registered to $S_i(t)$ must perform a busy wait;
- **R5** After input phase is terminated, each core $P_i \in C(t)$ deregister to $S_i(t)$.

The example in Figure 8, illustrates how inter-core synchronization protocol works.

Consider two processors P_0 and P_1 executing tasks with different activation times. According to the rules of inter-core synchronization protocol, at time $t = 0$ both processors are required to perform LET communication operations, hence $C(0) = \{P_0, P_1\}$. In this example, after the output phase, deregistration to $S_o(t)$ and registration to $S_i(t)$ are merged in a single phase called switch phase (S). Once P_0 ends the switch phase, it must wait for P_1 before starting the input phase. After P_1 ends the switch phase, both processors can start the input phase, followed by the deregistration phase to $S_i(t)$. At $t = 2$, the execution flow is the same as $t = 0$, but since only P_0 is involved in the LET communication phase, there is no busy-wait synchronization.

5 IMPLEMENTATION

The overall implementation of the LET paradigm is based on different functions that take care of: **(i)** initialization of all the data structures, **(ii)** implement the DMP, **(iii)** implement the inter-core synchronization protocol and **(iv)** implement the communication mechanism between tasks. The communication function represents the main difference between kernel-space and user-space implementations. In particular, the kernel-space communication phase is executed within a kernel routine, while the user-space implementation exploits a dedicated task for each core, called LET task and denoted with Γ_L, running at the highest priority of the system that takes care of the communication between tasks. Input and output operations implement the buffer optimization technique [2], according also to the intra-core or inter-core communication mechanism. In particular, for the second case, the communication is based on simple memory copies.

5.1 Dynamic Membership Protocol

The implementation of the DMP can be divided into two different functions executing: **(i)** the registration phase and **(ii)** the deregistration phase. In the kernel-space implementation, these functions are available as system calls. On the opposite, in user-space implementation, these functionalities are available as simple functions embedded in a C library. It is important to note that the implementation of both registration and deregistration functions are valid for kernel space and user space, with minor differences.

Algorithm 1 shows the pseudo-code of the implementation for the registration function. The arguments required for the registration phase are related to **(i)** the core number where the task is running, **(ii)** the topic of the channel, **(iii)** the reference to the task's parameters, **(iv)** the reference to the alias, if previously created, **(v)** the type of access to the channel and, finally, **(vi)** the size of the data exchanged through the channel. Note that the reference to the task's parameters can be specified with a data structure containing the activation time, the period, and the task's name. The type of access to the channel, instead, specifies if the registering task is a producer or a consumer. According to the formalization of the registration phase described in Section 4.3, the first operation is to create the alias and the private buffers of the task (line 2-5). If the alias passed as an argument is not valid, it is created starting from the parameters of the task and linked to the buffer. Then, the channel is created, if needed, and linked to the alias (line 8-12). Finally, the alias is registered to the list L_{r_i} if not done yet (line 15-17). Note that the function at line 19 differs from kernel-space and user-space implementations. In particular, in the first case, the finalization of the registration can be performed simply by updating the task's data structure and linking the alias to it. In the second case, it is possible to finalize the registration with a notification to Γ_L. Such notification is implemented using a shared flag, denoted with F_r, set every time a new task requests to finalize the registration, and is reset once Γ_L finalizes the registration.

Algorithm 1 Registration pseudocode

```
1:  function LETREGISTRATION(P_i, l, p_τj, γ_j, at, s)
2:      if γ_j == null then
3:          alias γ_j = CreateAlias(p_τj);
4:      end if
5:      buffer b_j,l = CreatePrivateBuffer(s);
6:      LinkPrivateBuffer(γ_j.buffers_list, b_j,l);
7:      MutexLock(m_ch);
8:      ch = SearchChannel(L_b, l);
9:      if ch == null then
10:         ch = CreateAndRegister(l, L_b);
11:     end if
12:     Link(ch, γ_j, b_j,l, at);
13:     MutexUnlock(m_ch);
14:     MutexLock(m_reg);
15:     if TaskNotRegistered(γ_j, L_ri) then
16:         RegisterAlias(γ_j, L_ri);
17:     end if
18:     MutexUnlock(m_reg);
19:     FinalizeRegistration();
20: end function
```

Algorithm 2 illustrates the pseudo-code of the implementation of the deregistration phase for both kernel space and user space. The arguments required by the deregistration function are **(i)** the core number where the task executes and **(ii)** the alias obtained during the registration phase. Note that, for the kernel-space implementation, the alias is not passed as an argument, but instead retrieved through the kernel data structure of the task. When a task is deregistered it is removed from the list L_{r_i} first (line 3). Then all the channels related to the task are unlinked from the alias and, for each channel, the number of active members is decreased. If a channel has only one member, it can be deregistered from list L_b and deleted. When deleting a channel, also all the buffers in the unused buffers queue are deleted, as well as the message buffer and the spare buffer. For each channel unlinked, the correspondent private buffer belonging to the task requesting the deregistration is deleted (line 7-14). Once all the channels are unlinked, the alias is deleted (line 16). Also, in this case, the implementation of the function at line 5 differs from kernel space and user space. In the first case, it is possible to update the task's data structure

Algorithm 2 Deregistration pseudocode

```
1: function LETDEREGISTRATION(P_i, γ)
2:     MutexLock(m_reg);
3:     RemoveToList(γ, L_{r_i});
4:     MutexUnlock(m_reg);
5:     FinalizeDeregistration();
6:     MutexLock(m_ch);
7:     for each ch ∈ γ.channel_list do
8:         if ch.members == 1 then
9:             DeleteChannel(ch, L_b);
10:        else
11:            ch.members = ch.members - 1;
12:        end if
13:        DeleteBuffer(γ.buffers_list);
14:    end for
15:    MutexUnlock(m_ch);
16:    Delete(γ);
17: end function
```

by unlinking the alias, meaning that the task is not registered anymore to the LET paradigm. For user-space implementation, instead, the deregistration to the list L_{r_i} is completed with a notification to the LET task with a shared flag denoted as F_d, set every time a new task is deregistered. The flag is reset once Γ_L finalizes the deregistration.

5.1.1 Race conditions analysis

Registration and deregistration operations can be executed in parallel with tasks executing in different cores, leaving the possibility that race conditions occur. In particular, multiple tasks may register different channels at the same time, leading to a possible data inconsistency when sliding the list L_b. Even if the L_{r_i} list is private to the core, during the registration of the alias a task can be preempted by higher priority tasks, leading to data inconsistency. Hence, to avoid race conditions, registration operations to each list L_b and L_{r_i} are protected using mutual exclusion mechanism. In particular, the mutex used for the registration and deregistration of the channel is shared between all cores, since the list L_b is global. The mutex used for the registration to the list L_{r_i}, however, is local for each core. In both kernel-space and user-space implementations, the mutex m_{ch} is not shared with the LET communication function. In particular, the kernel routine and the LET task directly access the channels during the communication phase using the links stored in the alias registered to the set L_{r_i} instead of scrolling the list L_b, which avoids race condition between communication and registration phases. For what concerns the mutex m_{reg}, in the kernel-space implementation is possible to access the alias directly through the reference stored in the task data structure, avoiding hence the need to contend the mutex. As opposed, the LET task needs to access the list L_{r_i} to find the alias of newly registered tasks, requiring hence to acquire the mutex. Thus, a scheduling point is introduced inside the LET task, which can be preempted in favor of tasks that need to finish the registration to the list L_{r_i}, increasing the overhead introduced by the LET communications.

5.2 Inter-core Synchronization Protocol

The inter-core synchronization protocol is embedded inside the LET communication function. Also in this case, it can be divided into two different functions executing: (i) the registration to the writing phase and (ii) the registration to the read phase. The protocol is implemented without violating the formalization described in chapter 4.4 and respecting the GIOTTO semantics.

5.2.1 Temporal Drift Problem

During the normal execution of the system, multiple tasks scheduled in different cores can have the same activation time. In this scenario, the cores must be synchronized to execute the LET communication to guarantee the respect of GIOTTO semantics. However, synchronization can fail due to temporal drift that can occur when handling events. Take, for example, two tasks τ_i and τ_j executing on different cores P_0 and P_1, with the same activation time. When both tasks are activated, P_0 and P_1 must execute the LET communication function registering to the write phase first, and then to the read phase. However, one of the processors can execute faster than the others due to a different computational load performed before the LET communication function. In this scenario, the "fastest" processor starts immediately the write and read operations, since the "slowest" one is not yet registered. Hence, the GIOTTO semantics is not respected. To avoid synchronization failures, an algorithm is required to guarantee the correct synchronization between cores. The algorithm is formalized as follows:

- For each core, a timestamp related to the nearest activation of LET communication is assigned.
- For each execution of the LET communication function, each core checks if it is already registered to the set $S_o(t)$. If not, it registers itself and other cores with the same timestamp value.
- Once the read phase is executed, each core updates its timestamp with the nearest LET communication activation time.

5.2.2 ICSP with Regulated Memory Access

This implementation is inspired by the work done in [2], in which it is explained how to regulate the global memory accesses by the LET communication function to avoid memory contention. We adapted the original algorithm designed for AUTOSAR Classic [5] system for a POSIX-like system. In particular, in the original implementation, an absolute order is established between cores accessing the global memory, assigning a priority to each of them. However, in this work it is assumed that the number of tasks joining the LET paradigm can vary over time, creating new LET transactions and making it difficult to adopt an absolute priority order. Hence, the algorithm adopted is based on a relative order for cores that access the global memory based on the FIFO policy. Each core is provided with a ticket, that is retrieved every time it is required registration to $S_o(t)$. The ticket is implemented as a data structure containing the following fields:

- **CPU id**: it represents the core number of the owner of the ticket;
- **wait spinlock**: it is used to perform busy wait for both the write and read phases.

During the initialization phase, one ticket is created for each core with its spinlock initialized as busy. Every time a core needs to perform a write or read phase, its ticket is added to the FIFO list. Vice versa, when the core exits the LET communication phase, the ticket is removed from the FIFO list. The algorithm used to implement the inter-core synchronization protocol with regulated access to memory is formalized as follow:

- To start write phase, the core P_i must register the related ticket to the set $S_o(t)$. Write operations can immediately start only if the ticket of P_i is the first element of $S_o(t)$ and $S_i(t) \neq \emptyset$. Otherwise, it waits its turn with a busy wait;
- To start the read phase, the core P_i must register the related ticket to the set $S_i(t)$. Read operations can immediately start only if $S_o(t) \neq \emptyset$ and the LET communication function is the first element of $S_i(t)$. Otherwise, it waits its turn with a busy wait;

Algorithm 3 Write synchronization with RMA pseudocode

```
1: function WRITESYNCHRONIZATION(P_i)
2:     SpinLock(spin_s);
3:     ticket tk = RetrieveTicket(P_i);
4:     RegisterCores(S_o(t));
5:     if tk is first in S_o(t) then
6:         SpinUnlock(spin_s);
7:         return ;
8:     end if
9:     SpinUnlock(spin_s);
10:    SpinLock(tk.spinlock);
11: end function
```

Algorithm 3 illustrates the pseudo-code of the synchronization protocol with regulated memory access works. The function at line 4 implements the algorithm for the temporal drift problem. In case the core P_i is the first of the list, according to the FIFO policy, exit the synchronization function to execute the writing phase (line5-8). Otherwise, it waits on the spinlock of its ticket until another core unlocks it (line 9). The registration to the set $S_o(t)$ is protected with mutual exclusion mechanism to avoid cores registering multiple instances of the same ticket.

Algorithm 4 Read synchronization with RMA pseudocode

```
1: function READSYNCHRONIZATION(P_i)
2:     SpinLock(spin_s);
3:     ticket tk = RetrieveTicket(P_i);
4:     DeregisterCore(S_o(t), tk);
5:     RegisterCore(S_i(t), tk);
6:     if S_o(t) ≠ ∅ then
7:         SpinUnlock(spin_s);
8:         ticket tk_f = first element of S_o(t);
9:         SpinUnlock(tk_f.spinlock);
10:        SpinLock(tk.spinlock);
11:    else if S_i(t) ≠ ∅ then
12:        SpinUnlock(spin_s);
13:        ticket tk_f = first element of S_i(t);
14:        SpinUnlock(tk_f.spinlock);
15:        SpinLock(tk.spinlock);
16:    else
17:        SpinUnlock(spin_s);
18:        return
19:    end if
20: end function
```

Algorithm 4 shows the read phase in which the core first deregisters from the output phase and registers to the input phase (line 4-5). Cores can proceed for the read phase only if $S_o(t)$ and $S_i(t)$ are empty (line 16-18). Otherwise, it will first unlock the next core waiting for the output phase (line 8-9) or the ones waiting for the read phase, if $S_o(t)$ is empty (line 13-14). Note that, since the spinlock of the ticket is initialized as busy, every lock operation will result in a busy wait.

Once the read phase is terminated, the next element of the set $S_i(t)$ is unlocked if it is not empty. Otherwise, if $S_o(t) \neq \emptyset$, is unlocked the first element waiting for the writing phase. This last case can happens only if multiple LET cycles overlap.

5.2.3 ICSP with Synchronization Barrier

The second approach for the inter-core synchronization protocol exploits the synchronization barrier between cores, which does not avoid memory contention but is easier to implement. Denote with $spin_b$ the mutex used as a barrier. Define also a counter c_t, which keeps track of the number of cores registered to $S_o(t)$ at time t. The pseudo-code in Algorithm 5 illustrates the implementation of the synchronization for the writing phase.

Algorithm 5 Write synchronization with Barrier pseudocode

```
1: function WRITESYNCHRONIZATION(P_i)
2:     SpinLock(spin_s);
3:     if P_i ∉ S_o(t) then
4:         RegisterCores(S_o(t));
5:         UpdateCounter(c_t);
6:         SpinInit(spin_b, LOCKED);
7:     end if
8:     SpinUnlock(spin_s);
9: end function
```

The function at line 4 is the same used for Sec. 5.2.2. It is possible to notice that, during the registration to write phase, only the first core attempting to register to the set $S_o(t)$, registers also all other cores that require to perform LET communication, updates the counter of cores registered, and locks the barrier mutex (line 3-7). Note also that, since the synchronization barrier is applied only before the read phase, cores can execute the writing phase in parallel. The pseudo-code in Algorithm 6 illustrates the implementation of the $ReadSynchronization$ function.

Algorithm 6 Read synchronization with Barrier pseudocode

```
1: function READSYNCHRONIZATION(P_i)
2:     SpinLock(spin_s);
3:     DecrementAndRead(c_t);
4:     SpinUnlock(spin_s);
5:     if c_t ≠ 0 then
6:         SpinLock(spin_b);
7:         SpinUnlock(spin_b);
8:     else
9:         SpinUnlock(spin_b);
10:    end if
11: end function
```

The first operation to perform is to decrease the counter of registered cores (line 3). If the counter reaches the value zero, then the core P_i is the last one deregistering from the set $S_o(t)$. Hence, it will unlock the first core blocked on barrier mutex (line 9). Otherwise, it will wait on the barrier, and notify the next core waiting on the barrier once it has been unlocked from the busy wait (line 6-7). In this way, once all cores awaken from the barrier, they will start the read phase in parallel. Note that, since the barrier mutex is first locked during the write synchronization, every attempt to lock it in the read synchronization will end with a busy wait. It is important to notice that, if multiple LET cycles overlap, it is possible to incur violations of GIOTTO semantics. Hence, it is required that only one LET cycle at a time must be processed for this implementation to work correctly.

5.3 LET communication kernel routine

The key idea is to minimize the delay between task activation and effective LET communication operations. Each task activation is handled as an event that is fired when the activation time of the related task occurred. Hence, the routine is executed after all time events are collected. However, since not all events are related to tasks' activations, the LET communication routine must be skipped if not required. For this purpose, a queue is used to keep track of tasks that are activated at certain

times. Denote with $Q_{(i,t)}$ the queue of tasks activated on the processor P_i at time t. Each time a fired event is recognized as an activation event of a task τ_i that is registered to the LET paradigm, the related alias γ_i is inserted in the queue $Q_{(i,t)}$, ordered based on the registration number of the alias, enforcing time determinism. This process is implemented as a function that is called inside the handler of the activation time event. The pseud-code in Algorithm 7 illustrates the algorithm for the kernel routine that takes charge to perform the LET communication operations in kernel space implementation.

Algorithm 7 LET kernel routine pseudocode

```
1:  function LETOPERATIONS(P_i, t)
2:      if Q_(i,t) ≠ ∅ then
3:          WriteSynchronization(P_i);
4:          for each γ ∈ Q_(i,t) do
5:              OutputOperations(γ);
6:          end for
7:          ReadSynchronization(P_i);
8:          for each γ ∈ Q_(i,t) do
9:              InputOperations(γ);
10:         end for
11:         UpdateNextActivation();
12:     end if
13: end function
```

As it is possible to see from the pseudo-code, the communication operations take place if there are tasks activated at time t (line 2). The function at line 11 is undertaken to update the timestamp related to the next activation of the kernel routine. If the inter-core synchronization protocol used is with regulated memory access, this function also updates the ticket related to the core executing the kernel routine.

5.4 Design of LET task

For user-space implementation, the key idea is to exploit a dedicated task for each core, denoted with Γ_L, running at the highest priority of the system, to perform LET communication operations. In the original work [2], the dedicated task is designed as a Generalized Multiframe Task [24] and implemented as a periodic task with a period of 1 millisecond. In this work, however, the LET task is designed as a non-real-time task, scheduled under FIFO policy. This choice has been made according to the kernel used for this work, where it is possible to set the FIFO scheduler with higher priority w.r.t. EDF scheduler. Moreover, designing Γ_L as a periodic task under the EDF scheduler does not guarantee the priority required for the execution.

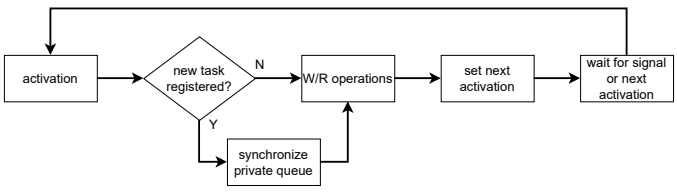

Fig. 9: Block diagram of LET task.

Once the task is activated, it performs a synchronization phase with L_{r_i} searching for new registered or deregistered tasks. In particular, the LET task uses a private queue of aliases, denoted as L_{Γ_L}, to store information related to all the tasks joining the LET paradigm. The private queue is sorted in ascending order concerning the activation time, updated at each iteration, of the tasks related to the aliases. The synchronization phase must be performed only when a new task registers or deregister from L_{r_i}, otherwise, it can be skipped. Then, the communication phase is undertaken, which involves the inter-core synchronization, the output, and the input phase. After the communication is completed, the timestamp related to the nearest activation time of Γ_L is updated, according to the algorithm explained in chapter 5.2.1. Finally, the task self-suspend waiting for an external signal upon new registration or deregistration of tasks to the LET paradigm or the time event related to the nearest activation time of the LET communication phase. Note that, since the task can be activated upon signal related to the registration and deregistration phase, it is possible that no communication operations must be performed. The pseudo-code in Algorithm 8 of the LET task is illustrated.

Algorithm 8 LET task body pseudocode

```
1:  function LETTASKBODY(P_i)
2:      do
3:          MutexLock(m_reg);
4:          SynchronizeQueue(F_r, F_d, L_r_i, L_Γ_L);
5:          MutexUnlock(m_reg);
6:          Q_(i,t) = SelectActivatedTasks(L_Γ_L, t);
7:          if Q_(i,t) ≠ ∅ then
8:              WriteSynchronization(P_i);
9:              for each γ ∈ Q_(i,t) do
10:                 OutputOperations(γ);
11:             end for
12:             ReadSynchronization(P_i);
13:             for each γ ∈ Q_(i,t) do
14:                 ReadOperations(γ);
15:             end for
16:             UpdateQueue(L_Γ_L);
17:         end if
18:         t = SelectNewActivationTime(L_Γ_L);
19:         TimedWait(V_cond, t);
20:     while 1
21: end function
```

The task takes as an argument the core number where it is executed. It is important to remark that it is allowed only one LET task for each CPU. The function related to the synchronization phase requires, as parameters, the flags used to notify new registration or deregistration events, introduced in chapter 5.1, the set of all registered tasks, and the private queue of the LET task (line 4). Note that, in the same manner of L_{r_i}, also the set L_{Γ_L} is private for each LET task. Note that the lock and unlock operations represent a scheduling point. In general, the LET task cannot be preempted by any task during communication operations. However, since the synchronization phase involves resources shared with other tasks running at a lower priority, it is reasonable to allow preemption only if required for tasks to complete the registration phase. Then, all the aliases of tasks activated at time t are selected for LET communication operations if any (line 7-17). In particular, all the activation times of the aliases in the queue $Q_{(i,t)}$ are updated according to the related period and inserted in the queue L_{Γ_L} which will be reordered according to the ascending activation time policy (line 16). Finally, the new activation time of the LET task is selected from the first element of the set L_{Γ_L}, and a timed wait operation is performed (line 18, 19). This last operation is undertaken using a timed condition variable V_{cond} and the timestamp of the new activation time. Each time a new task registers or deregister to the LET paradigm, it sends a notification on V_{cond}, which will trigger the activation of the LET task. However, if no registration or deregistration operations are undertaken, the wait operation will go into a timeout when

reaching the new instance of the LET communication phase.

6 Experimental Evaluation

This section describes an experimental evaluation that was carried out to determine the feasibility of the proposed approach and its impact on timing performance. Both kernel-space and user-space LET implementations discussed in chapter 5 are evaluated on an automotive application generated from a model provided by Bosch for the WATERS 2017 challenge [10], which is representative of a realistic engine control application. The evaluation is made running the application on a Xilinx ZCU102 board equipped with a Zynq Ultrascale+ MPSoC that provides a quad-core Arm®Cortex®-A53 running at 1.2 GHz, a dual-core Cortex-R5F real-time processor, FPGA programmable logic and 4 GB of RAM, connected to a Lauterbach TRACE32®PowerTrace to perform debugging and tracing. The kernel adopted for the implementations and the evaluations is a proprietary PSE53 [9] kernel module designed for the automotive field. Every time it performs a process context switch, the kernel considered for the evaluation also invalidates the cache to enhance inter-process isolation.

6.1 Waters Challenge

The WATERS 2017 challenge included a model of an engine control application made up of 1250 runnables organized into 21 tasks/ISRs that access 10000 labels. Approximately 5000 labels are fixed, whereas the others are actual communication variables. The model specifies which labels each runnable access, the type of access (read or write), and the number of accesses. It also provides runnable execution times, net of memory access, and memory contention times. The task periods and the ISRs' minimum inter-arrival times are also provided. The model includes a quad-core platform with statically assigned tasks. Since the original work was designed for AUTOSAR, some modifications were necessary to generate the application adapted for the PSE53 kernel and the implementations proposed, and the platform used. Despite the original model being designed for a quad-core platform, this work is based on the implementation of the challenge for ERIKA OS [25], where only three cores are used. Hence, one core and the corresponding tasks have been discarded. Moreover, all the ISRs tasks have been treated as periodic tasks using, as periods, the same parameters proposed in [2]. The tasks execution flow has been preserved, as all the data dependency between them. However, all the labels used in a single communication have been grouped in a single data structure as a unique message exchanged between tasks. This operation was required to be compliant with the implementations of LET for both kernel space and user space. Each task registers itself at the beginning of its period and deregisters only before its destruction. All the memory access parameters are maintained according to the original model.

6.2 Experimental Results

Experiments have been performed to evaluate the performance and make a comparison between user-space and kernel-space implementations w.r.t. the overhead introduced by the LET communication operations. Moreover, further analysis of performance has been conducted in order to evaluate the impact of the different inter-core synchronization protocols implementations described in chapter 5.2. All the evaluations are performed running all the tasks of the application with 2000 iterations, giving the possibility for all tasks to execute with the same activation time at least three times.

Figure 10 shows the average overhead introduced for both user space and kernel space implementations. From an initial analysis, it is possible to notice that the overhead introduced in the kernel-space implementation is lower w.r.t. the user-space, regardless of the inter-core synchronization mechanism adopted. The overhead increases not only with the number of tasks executed on each core but also with the amount of data exchanged. A deeper analysis has been performed to understand where the LET communication functions spent most of the time in execution.

L	Protocol	C#	Average Execution Times [μs]				
			Reg.	Writes	Dereg.	Reads	Update
K	SB	1	0.18	2.28	0.25	0.2	0.36
		2	0.2	3.18	1.61	0.74	0.44
		3	0.2	12.96	3.6	5.6	0.36
	RMA	1	0.26	2.44	1.44	0.2	0.51
		2	7.12	2.09	1.58	0.62	0.57
		3	19.31	12.5	6.92	5.37	0.51
U	SB	1	0.61	2.04	22.2	0.55	0.95
		2	0.62	2.9	122.06	0.67	0.75
		3	0.7	11.59	82.9	6.54	1.38
	RMA	1	0.68	2.19	26.13	0.6	1.13
		2	115.59	2.6	16.5	0.7	0.93
		3	98.98	11.28	9	6.05	1.53

TABLE 2: Average execution times of the different phases of LET communication. Legend: L = Level, K = Kernel, U = User, C# = Core Number, SB = Synchronization Barrier, RMA = Regulated Memory Access.

Table 2 shows the execution times, for each core, of all phases composing the communication function, from kernel-space and user-space implementing both synchronization barrier and regulated memory access mechanism related to Figure 10 Configuration 1). For kernel space, the third core experienced the highest overhead, where the main computation happens during the write operations. However, using the synchronization barrier, the time spent in synchronization is definitely lower w.r.t. the regulated memory access, which requires some microseconds to execute. in particular, with the RMA protocol, the highest execution time is recorded during the deregistration phase. Hence, it is possible to infer that memory contention, which is currently using a synchronization barrier, inflates less on overhead introduced w.r.t. a protocol that avoids memory interference but imposes an access order. For what concerns the LET task on user-space implementation, the main computation time is spent in the inter-core synchronization protocol. Note that, with both synchronization mechanisms, the major overhead occurs when waiting for other LET tasks before the read phase. The reason behind this behavior is due to the process context switch to the dummy process that happens since there are no tasks belonging to the WATERS challenge to execute, which cause also an invalidation of the cache. In general, however, it is reasonable to infer that the user-space implementation suffers from higher overhead due to longer execution time when acquiring spinlocks w.r.t. the kernel-space and for delays that occur when the cache is invalidated. To analyze better the behavior of the communication phase, three more tasks are added to the application, with the lowest priority possible, that work as idle tasks in order to avoid any invalidation of the cache due to context switch between processes.

As it is possible to see from Figure 10 Configuration 2), the overhead of the LET communication phase for both kernel-space and user-space is lower w.r.t. The analysis shown in Figure 10 Configuration 1), due to the absence of the delay introduced with the invalidation of the cache, results also in a lower time spent in write and read operations.

Since the overhead introduced by the LET communication phase directly impacts all the tasks executing on the systems, an analysis of the response time of eleven representative tasks of the challenge model is carried out in Figure 11.

The analysis is performed by taking the normalized response time of the tasks w.r.t. their periods and comparing the

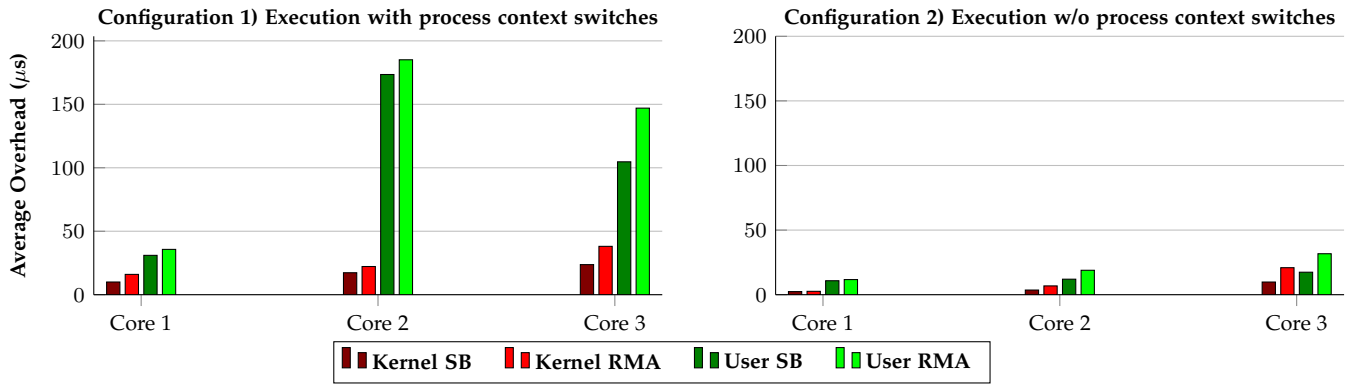

Fig. 10: Comparison of LET communication function between kernel-space and user-space, considering the inter-core synchronization protocol with both implementations based on synchronization barrier (SB) and regulated memory access (RMA). In Configuration 1) the analysis with the system incurring in process context switches, and in Configuration 2) the analysis with the system avoiding any process context switches.

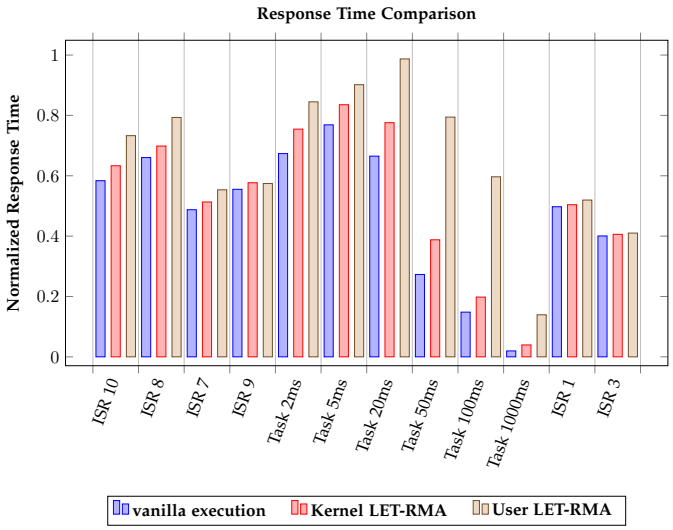

Fig. 11: Longest observed normalized response time under vanilla communication compared with both user-space and kernel-space LET-RMA.

Level	Protocol	Core #	Average Execution Times [μs]	
			Writes	Reads
Kernel	SB	1	2.28	0.2
		2	3.18	0.74
		3	12.96	5.6
	SB OPT	1	2.33	0.2
		2	3.68	0.27
		3	5.1	0.26
User	SB	1	2.04	0.95
		2	2.9	0.67
		3	11.59	6.54
	SB OPT	1	2.07	0.58
		2	2.68	0.6
		3	4.78	0.62

TABLE 3: Average execution times of read and write operations under synchronization barrier protocol with and without mapping optimization. Legend: SB = Synchronization Barrier.

execution with and without LET communication operations, running the application in the worst scenario where the cache is invalidated every time there is a process context switch. The response time of all tasks running under the LET paradigm considering both kernel-space implementation and user-space implementation is slightly higher. In particular, user space implementation suffers from a task's context switch every time the LET task is woken up and a delay introduced when the cache is invalidated, which increases the overhead introduced. Hence, as a conclusion, it is reasonable to infer that kernel-space implementations do not degrade the performance of the application, giving however the benefit of a deterministic execution, while user-space implementation can seriously put in danger the correct execution of the application.

Finally, one last analysis has been performed to analyze the behavior and the performance obtained by exploiting the intra-core communication mechanism. Hence, tasks have been remapped to cores according to a data-driven policy, executing tasks with the highest communication load in the same core. In particular, the $Task10ms$ is moved from the third core to the second core. As in the previous case, the application is executed in the worst scenario where the cache is invalidated, to highlight the benefit of the intra-core communication mechanism. Table 3 shows the performance, in terms of average overhead introduced, with the remap of the tasks for both kernel-space and user-space implementations.

It is possible to notice that exploiting the intra-core mechanism allows, for both kernel-space and user-space implementations, to reduce the computational load of the communication phase on the third core, obtaining an improvement of about 60% for the write operations and 94% for the reads operations w.r.t. the default mapping configuration, maintaining the computational load on the second core more or less unchanged. This result shows that the implementations described in this work are strongly dependent on the mapping of the tasks to the cores, giving new parameters of optimization for increased performance of the system.

7 Related Work

Menard et al. [26] analyzed the issue of the non-determinism problem in AP. In particular, they took as an example the emergency brake assistant available in the AUTOSAR Adaptive Demonstrator and showed how the Reactors model can be effectively used to enforce determinism. The semantics of the Reactors model (forming also the basis of the Lingua Franca framework [27]) is indeed well-defined and very powerful. However, this novel programming model could meet the reluctance of the automotive industry which is quite conservative and already familiar with the LET model.

An implementation of the LET paradigm for predictable execution of AUTOSAR applications is developed in [2] where the authors highlight different LET semantics and how they affect the performance achieved and propose an implementation that allows scheduling memory access to avoid memory contention. Even if this approach has been developed for Classic AUTOSAR applications, it contains key concepts useful for the implementation of the LET paradigm for dynamic POSIX-compliant systems. In particular, it has been taken into consideration for the development of the Regulated Memory Access protocol explained in section 5.2.2.

Several efforts have been spent by Kai-Björn Gemlau et al. [3] for the development of the LET paradigm for distributed-systems architecture in the automotive field. In this work, the LET paradigm has been generalized to extend it to a distributed system and implemented at the system level, preserving the determinism and the causality during the execution of AUTOSAR applications with data dependencies and running on different ECUs. Also, in this case, the solution proposed targets the AUTOSAR Classic platform and hence it is not suitable for dynamic systems.

Yano et al. [28] presented a paper that explains a new theoretical algorithm for the parallel and distributed execution of real-time tasks using multi-/many-core platforms under the LET paradigm while avoiding memory contention, which is then implemented in the later work of the same author [29]. One of the problems addressed is the overhead introduced by the LET communication operations which are assumed to be undertaken on a dedicated core and can cause deadline-miss. In our work, we mitigate the problem of overhead by splitting the LET communication workload on all the cores. In this way, the LET operations are undertaken only in these cores that require communication, avoiding unnecessary waiting time on cores that do not.

Another relevant work is the one made by Kluge et al. [30] which aims to implement the LET model on a time-predictable platform. The authors used MOSSCA [31], which is a factored operating system [32], with the assumption that each core has its local memory where the operating system and application code and data can be stored. Moreover, since all communication is based on explicit messages, a NoC is required with the assumption to provide hard real-time guarantees. The authors made also an evaluation of the overhead introduced by all the relevant code that takes charge to perform LET communications, making a comparison between execution with global memory and local scratchpad memory. A comparison with the evaluations performed with our work, however, is not possible due to the different assumptions and design of the implementations proposed, which fit the real-time operating system used.

Finally, the work done by Henzinger et al. [14] related to the GIOTTO framework has been taken into account to understand better the semantics adopted for the LET paradigm.

8 Conclusion and Future Work

In this work, we presented a design for a practical implementation of the LET paradigm on POSIX-compliant systems for multi-core platforms. We analyzed the behavior of the system considering the additional computational load introduced by LET communication. The kernel space implementation always performs better w.r.t. the user space implementation, introducing an overhead that does not harm the correct execution of the system. On the other hand, the user-space implementation can achieve results similar, but slightly worse, to the kernel space only when no process context switch is present. Otherwise, the system's temporal constraints can be broken due to significant delays introduced. Despite our kernel-space implementation being strictly dependent on a proprietary real-time operating system, the approaches presented in the paper and, in particular, the findings related to the user-space implementations can be generalized to every Posix-like operating system supporting real-time schedulers.

Possible future work can involve a system-level LET for a distributed system with an AUTOSAR Adaptive system as a target for a possible deployment.

References

[1] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," in *Embedded Software: First International Workshop, EMSOFT 2001 Tahoe City, CA, USA, October 8–10, 2001 Proceedings 1*. Springer, 2001, pp. 166–184.

[2] A. Biondi and M. Di Natale, "Achieving predictable multicore execution of automotive applications using the let paradigm," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2018, pp. 240–250.

[3] K.-B. Gemlau, L. Köhler, R. Ernst, and S. Quinton, "System-level logical execution time: Augmenting the logical execution time paradigm for distributed real-time automotive software," *ACM Transactions on Cyber-Physical Systems*, vol. 5, no. 2, pp. 1–27, 2021.

[4] AUTOSAR, "The autosar standard, version 4.3." [Online]. Available: http://www.autosar.org

[5] ——, "Autosar classic." [Online]. Available: https://www.autosar.org/standards/classic-platform

[6] ——, "Specification of timing extensions." [Online]. Available: https://www.autosar.org/fileadmin/standards/R22-11/CP/AUTOSAR_TPS_TimingExtensions.pdf

[7] ——, "Autosar adaptive." [Online]. Available: https://www.autosar.org/standards/adaptive-platform

[8] C. Scordino, A. G. Mariño, and F. Fons, "Hardware acceleration of data distribution service (dds) for automotive communication and computing," *IEEE Access*, vol. 10, pp. 109 626–109 651, 2022.

[9] IEEE, ""std 1003.13-2003 standardized application environment profile (aep) - posix(tm) realtime and embedded application support"." [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1342418

[10] A. Hamann, D. Dasari, S. Kramer, M. Pressler, F. Wurst, D. Ziegenbein., "WATERS industrial challenge 2017." [Online]. Available: https://waters2017.inria.fr/challenge/#Challenge17

[11] L. Köhler, P. Hertha, M. Beckert, A. Bendrick, and R. Ernst, "Robust cause-effect chains with bounded execution time and system-level logical execution time," *ACM Transactions on Embedded Computing Systems*, 2022.

[12] A. Cervin, D. Henriksson, B. Lincoln, and K.-E. Årzén, "Jitterbug and truetime: Analysis tools for real-time control systems," in *Proceedings of the 2nd Workshop on Real-Time Tools*. Copenhagen, Denmark, 2002.

[13] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[14] T. A. Henzinger, C. M. Kirsch, M. A. Sanvido, and W. Pree, "From control models to real-time code using giotto," *IEEE Control Systems Magazine*, vol. 23, no. 1, pp. 50–64, 2003.

[15] C. M. Kirsch and A. Sokolova, "The logical execution time paradigm," *Advances in Real-Time Systems*, pp. 103–120, 2012.

[16] A. Davare, Q. Zhu, M. Di Natale, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli, "Period optimization for hard real-time distributed automotive systems," in *Proceedings of the 44th annual Design Automation Conference*, 2007, pp. 278–283.

[17] M. Beckert, M. Möstl, and R. Ernst, "Zero-time communication for automotive multi-core systems under spp scheduling," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2016, pp. 1–9.

[18] S. Resmerita, A. Naderlinger, M. Huber, K. Butts, and W. Pree, "Applying real-time programming to legacy embedded control software," in *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*. IEEE, 2015, pp. 1–8.

[19] J. Hennig, H. von Hasseln, H. Mohammad, S. Resmerita, S. Lukesch, and A. Naderlinger, "Towards parallelizing legacy embedded control software using the let programming paradigm, in 2016 ieee real-time and embedded technology and applications symposium (rtas)," *IEEE Computer Soc*, pp. 1–1, 2016.

[20] D. Zmaranda, G. Gabor, and A. Nicula, "Producer-consumer paradigm in real-time applications," *Journal of Computer Science and Control Systems*, no. 1, p. 83, 2009.

[21] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM computing surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.
[22] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *Proceedings of the tenth international symposium on Hardware/software codesign*, 2002, pp. 73–78.
[23] M. Beckert and R. Ernst, "The IDA LET machine—an efficient and streamlined open source implementation of the logical execution time paradigm," in *International Workshop on New Platforms for Future Cars (NPCar at DATE 2018)*, 2018.
[24] S. Baruah, D. Chen, S. Gorinsky, and A. Mok, "Generalized multiframe tasks," *Real-Time Systems*, vol. 17, pp. 5–22, 1999.
[25] [Online]. Available: https://retis.santannapisa.it/~a.biondi/LET/
[26] C. Menard, A. Goens, M. Lohstroh, and J. Castrillon, "Achieving determinism in adaptive autosar," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 822–827.
[27] M. Lohstroh, C. Menard, S. Bateni, and E. A. Lee, "Toward a lingua franca for deterministic concurrent systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 4, pp. 1–27, 2021.
[28] A. Yano, S. Igarashi, and T. Azumi, "Contention-free scheduling algorithm using let paradigm for clustered many-core processor," in *2021 IEEE/ACM 25th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE, 2021, pp. 1–4.
[29] ——, "Let paradigm scheduling algorithm considering parallel processing on clustered many-core processor," *Journal of Information Processing*, vol. 30, pp. 646–658, 2022.
[30] F. Kluge, M. Schoeberl, and T. Ungerer, "Support for the logical execution time model on a time-predictable multicore processor," *ACM SIGBED Review*, vol. 13, no. 4, pp. 61–66, 2016.
[31] F. Kluge, M. Gerdes, and T. Ungerer, "An operating system for safety-critical applications on manycore processors," in *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE, 2014, pp. 238–245.
[32] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos) the case for a scalable operating system for multicores," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 76–85, 2009.