Achieving Predictable Multicore Execution of Automotive Applications Using the LET Paradigm

Alessandro Biondi and Marco Di Natale
Scuola Superiore Sant'Anna, Pisa, Italy
E-mail: {alessandro.biondi, marco.dinatale}@sssup.it

***Abstract*—Next generation automotive applications require support for safe, predictable, and deterministic execution. The Logical Execution Time (LET) model has been introduced to improve the predictability and correctness of time-critical applications. The advent of multicore architectures, together with the need to ensure time predictability despite the complex memory hierarchy and the hardware resources shared by the cores, is an additional motivation for the use of the LET paradigm in conjunction with a suitable scheduling and memory access model. In this paper, we show how an implementation of the LET model on actual multicore platforms for automotive systems brings the potential to improve time determinism at the price of a modicum run-time overhead. Multiple implementation options are discussed using the automotive AUTOSAR model and operating system standard, and a realistic application defined by Bosch for the 2017 WATERS challenge. Experimental data of executions on the Infineon Aurix platform show the feasibility of the proposed approach. The paper also provides a discussion on further implementation optimizations and other issues related to the general problem of memory-aware analysis of automotive applications on multicores.**

I. Introduction

The introduction of safety-critical functions in automotive systems, together with the advent of multicore platforms, brings the need to rethink the development and execution paradigms for embedded functionality. Developers need high levels of predictability, testability, and ultimately determinism in the execution of their code. The LET model was introduced as part of the GIOTTO framework [1] to eliminate output jitter and provide time determinism in the code implementation of controls. Recently, there has been a renewed interest in the LET execution paradigm by automotive electronics vendors, as witnessed by the recent WATERS challenges [2].

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 LET model is also characterized by an execution model of functional units with execution order (causality) constraints. The adoption of this model brings to the foreground not only the concept of timeliness, but also of causality, which is typical of synchronous languages and their implementations.

A key observation is that the LET execution model not only avoids output jitter but has the additional benefit of scheduling precisely in time the accesses to the communications variables. This can be extremely valuable in the multicore execution of tasks communicating remotely. Several techniques have been proposed to analyze the time performance of real-time tasks on multicores in the face of the sharing of memory and other hardware resources, including interconnects, arbiters and I/O devices. Unfortunately, COTS multicore platforms are not designed with the aim of providing predictability, with the consequence that conventional analysis techniques can be at best pessimistic. The LET execution model can improve and restore predictability by controlling the time when memory resources are accessed.

For modern automotive systems, the AUTOSAR standard [3] provides a reference model for the development of applications, including a model of the functions and the tasks, a standard API for communication and execution, and a standard platform architecture. In AUTOSAR, the application consists of a set of communicating runnables grouped into tasks and statically allocated and scheduled on the system cores. The AUTOSAR model is based on the concept that the task model and the communication implementation are automatically generated by dedicated tools based on configuration information, the model of the application, and platform constraints. Such aspects are of paramount importance when designing a LET implementation for automotive applications.

This paper. In this paper, we draw analogies from all these concepts and propose an integrated approach to face the problem of implementing and scheduling task communications in multicores. We first provide a characterization of possible variants of the LET paradigm. Next, we discuss the implementation of the LET paradigm in agreement with the AUTOSAR model and API on a multicore platform that is very common in the automotive domain and representative of typical HW configurations: the Infineon Aurix microcontroller. Then, we provide an analysis of possible actual implementation options based on the ERIKA RTOS (compliant with the OSEK automotive standard and a de-facto representative of the typical behavior of AUTOSAR OS kernels). Finally, we provide our results on the evaluation of a code implementation of the application proposed by Bosch in the context of the WATERS 2017 challenge [2], executed with our LET implementation on the Aurix. Other related issues will be shortly discussed but are not the main concern of this work, including the schedulability analysis with explicit consideration of memory contention.

II. Modeling and Background

This paper considers applications composed of a set of n periodic tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$, each characterized by a worst-case execution time (WCET) C_i, a period T_i, and a relative deadline $D_i \leq T_i$. A bound on the response time of τ_i is denoted by R_i. The tasks execute upon a platform that comprises m processors $P_1, \ldots, P_m$, with 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 enables point-to-point communication between each core and each memory. Concurrent accesses to memories are arbitrated with a FIFO policy. Blocking memory access is assumed, i.e., no write or read buffers. Tasks are scheduled according to *partitioned fixed-priority* scheduling, and $hp(i)$ denotes the set of tasks with higher priority than τ_i. Each task is statically allocated to a

given processor $P(\tau_i)$. The symbol Γ_x denotes the set of tasks allocated to the processor P_x, while $\Gamma(\tau_i)$ denotes the set of tasks allocated to the same processor to which τ_i is allocated.

As a representative model for automotive AUTOSAR applications, each 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 WCET $C_{i,j}$. The WCET of a task τ_i is simply computed (as a first-order approximation) as the sum of the WCETs of its runnables.

Runnables communicate by means of *labels*: variables that can be read and written in an atomic manner. Each runnable $\rho_{i,j}$ may read or write labels from a set $\mathcal{L} = \{\ell_1, \ell_2, \ldots, \ell_q, \ldots\}$. Each label ℓ_q is characterized by a size (an integer number of bytes no larger than the processor word) and an access cost λ_q. $\mathcal{L}_i$ denotes the set of labels accessed by task τ_i, which can be constructed by looking at all the labels accessed by the runnables in τ_i. Each label is written by at most one task, while it can be read by multiple tasks. Labels that are written and read by tasks on different cores are mapped to the global memory, while all the other labels (including constant data) are mapped to the local memories (including their duplicates). The set of labels mapped in global memory and accessed by τ_i is denoted by $\mathcal{L}_i^G \subseteq \mathcal{L}_i$. Task τ_i accesses label ℓ_v at most $N_{i,v}$ times. For a given pair of communicating tasks, a producer τ_P and a consumer τ_C, $L^W(\tau_P, \tau_C)$ denotes the set of labels that are written by τ_P and read by τ_C. $L^R(\tau_C, \tau_P)$ denotes the set of labels that are read by τ_C and written by τ_P. In order to compare the effects of different memory access policies, the WCETs do not include the execution cost to read and write the memory labels.

A. Logical Execution Time

The LET model we assume is inspired by the original proposal in [1]. However, in Section III we discuss other semantics and implementation options that are still inspired by the need for predictable and deterministic execution. In addition, we include a model for the implementation of the LET execution paradigm in the context of the AUTOSAR standard. For this option, we adopt from AUTOSAR definitions and most of the semantics for the activation and communication of functions (runnables in AUTOSAR).

Functional and runnable model. In the original LET proposal, the execution of functions is characterized by a predictable and deterministic execution that preserves the order of execution of the functions and provides for deterministic communication and actuation times. In the LET model, the system is a network of functional blocks $\mathcal{B} = \{b_1, b_2, \ldots, b_n\}$. Communicating blocks may be related by execution order constraints (expression of causality). Each block is characterized by a periodic activation and execution. Each block can perform multiple reads and multiple writes. Communication may occur between blocks with different periods, and each writer can have multiple readers for the same piece of information. In the LET execution model, blocks are executed by tasks (or threads) and their input and output operations are grouped together at the task level.

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

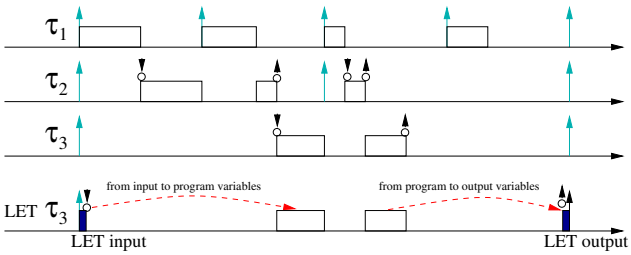

Fig. 1. The LET model of execution. The short arrows upon the dots denote the input/output operations performed by the tasks.

input of the task data is performed at the task activation, and the output is performed at the end of the task period. All task inputs are stored in local variables at the task activation. Similarly, all outputs need to be stored in local variables and are 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. In essence, LET is a sample and hold mechanism with synchronized execution of the input and output part.

III. LET Semantic Options

The following sections present and discuss three different LET semantics characterized by different timing properties and implementation concerns using a simple *running example*.

Running example. Consider a producer task τ_P communicating with a consumer task τ_C by means of a shared label ℓ. Task τ_P acquires input data from a sensor, then it elaborates the data producing an update for ℓ. In a dual manner, τ_C reads data from ℓ, performs further elaboration on such data, and then performs a control output operation.

A. The GIOTTO LET semantic

In abstract terms, the LET paradigm assumes that the input/output operations happen in *zero time*. However, in a real implementation, the actual input/output operations must be scheduled for execution. The order with which they are executed influences the timing properties of the systems, especially when flow preservation along communication chains is required. To ensure time determinism, the GIOTTO programming paradigm [4] specifies an order of execution for the writes and reads of blocks communicating using LET to enforce causality (see GIOTTO micro steps in [4]).

Without delving too much in details, the order of execution in GIOTTO can be recapped as follows: **(i)** first, data write and control output (i.e., actuation) operations are performed, then **(ii)** input (i.e., sensing) and data read operations are undertaken. This order is applied at every periodic instance of the tasks in the system and considers the input/output operations of all the tasks in a holistic manner, i.e., if the period instances of two tasks begin at the same time, then the communication is collapsed within a pair of phases (i) and (ii), each comprising the communication operations for *both* tasks.

Figure 2 illustrates an example schedule of LET communication with GIOTTO semantic. The communication phases are scheduled at the beginning of each periodic instance, which is compatible with the case in which they are performed by

a high-priority task. As shown in the figure (dashed arrow), write operations have precedence on read operations, and the third periodic instance of τ_C reads the data written by the first instance of τ_P.

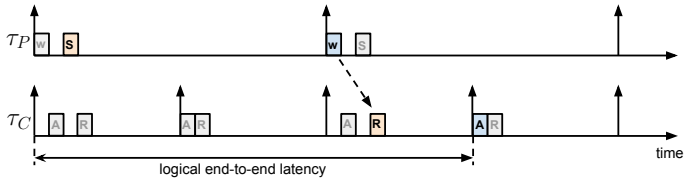

Fig. 2. Example schedule of LET communication with GIOTTO semantics. The producer task τ_P has a period of $T_P = 4$ ms while the consumer task τ_C has a period of $T_C = 2$ ms. Legend: W = write, A = actuation, S = sensing, R = read. The operations that do not contribute to the end-to-end latency indicated in the figure are colored with light grey.

As long as the tasks complete their execution before the release of their next instance (i.e., according to the implicit-deadline model), and ignoring the time needed to perform the actual input/output operations, the end-to-end latency with which the system *reacts* to the control input is *deterministic*, i.e., it is independent from the tasks' response times and equal to $T_P + T_C$.

The same semantic can be realized by scheduling the input/output operations at different times than the ones in Figure 2: implementation issues related to the scheduling of LET communication are addressed in Section V.

B. Interleaved LET communications

By altering the order with which the input/output operations are performed, it is possible to obtain different end-to-end latencies. For instance, consider the case where the LET communication phases are grouped by tasks, i.e., input and output operations are interleaved. This case is compatible with a LET implementation where each task delegates the LET communication for its input/output operations to a dedicated high-priority task.

Figure 3 illustrates an example schedule of LET communication where the input/output operations of task τ_C have precedence on those of τ_P, i.e., they follow the rate-monotonic order (note the periods of the tasks in the figure caption). As it can be observed from the figure, differently from the case discussed in the previous section, the third periodic instance of τ_C is not able to read the data produced by the first instance of τ_P. This happens because the read operations of τ_C are scheduled *before* the write operations of τ_P. As a consequence, the data produced by the first instance of τ_P are only available to the fourth instance of τ_C, which determines an increase of the end-to-end latency with which the system reacts to the control input. Specifically, the latter becomes $T_P + 2T_C$.

C. LET for task chains

In the particular case in which a producer task τ_P only communicates with a consumer task τ_C, the LET model can be dropped for the internal communication of the chain and restored only at its boundaries, by enforcing an order of execution with an explicit activation signal. Under this scenario, the tasks have the same period $T_P = T_C = T$, but the consumer task τ_C incurs in *release jitter*, which depends on the response time of τ_P.

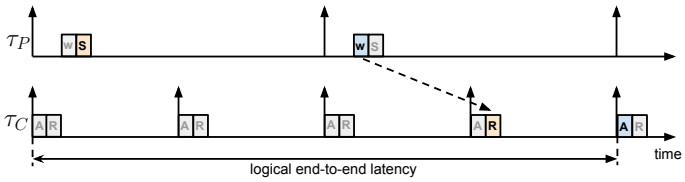

Fig. 3. Example schedule of LET communication with interleaved communication phases. The producer task τ_P has a period of $T_P = 4$ ms while the consumer task τ_C has a period of $T_C = 2$ ms. The same legend of Figure 2 applies.

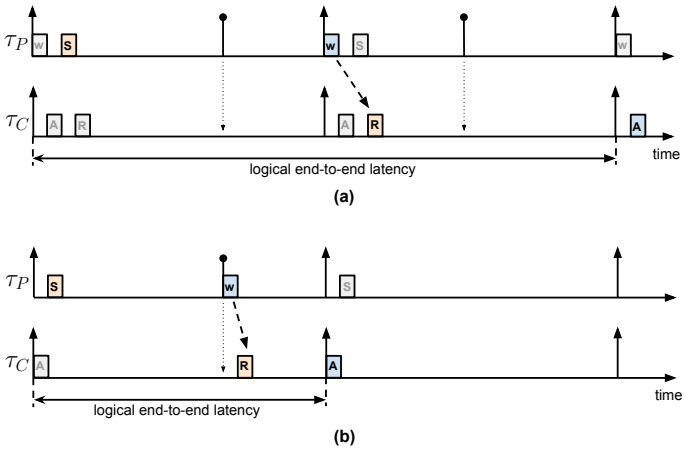

Fig. 4. Two examples of LET communication for a task chain. When it completes its execution, the producer task τ_P activates the consumer task τ_C (dotted arrow). Both the tasks have the same period, but τ_C incurs in release jitter. The same legend of Figure 2 applies. The marker with a large dot indicates the completion of a job of τ_P. Inset (a) depicts the case where the GIOTTO semantic is applied, while inset (b) depicts an alternative case where data write and read operations are scheduled when τ_C is activated.

Figure 4(a) illustrates an example schedule for the considered task chain where LET communication follows the GIOTTO semantic introduced in Section III-A. Since the communication phases are performed at the beginning of the periodic instances, the data produced by the first instance of τ_P is available to the second instance of τ_C. As a result, the (logical) end-to-end latency is equal to $T_P + T_C = 2T$. However, this latency may be reduced while still preserving the flows of data values between two consecutive instances of such tasks, i.e., the data produced by a job of task τ_P must be available to the successor job of τ_C explicitly activated at the completion of τ_P. As illustrated in Figure 4(b), data writes and reads are performed at a time instant (e.g., at the response time of τ_P as in the figure) within the period of the tasks. Nevertheless, the LET paradigm can be retained for external inputs and outputs, thus maintaining predictability for the control timing. In other words, this scheme can be seen as a LET paradigm applied in a holistic manner to the task chain, rather than to each individual task. As depicted in Figure 4(b), the resulting end-to-end latency with which the system reacts to the control input is equal to T.

IV. Realizing LET with GIOTTO Semantics on Multicores

This section presents a method for realizing the LET communication with GIOTTO semantics on a multicore platform. To generalize the proposed method, the following sections consider

the abstract platform model in Section II. The method is later instantiated for a real platform in Section V. The local copies of the labels required by the LET are allocated to local memories, i.e., a task τ_i running upon processor P_k and accessing a label ℓ_q disposes of a local copy for ℓ_q, named $\ell_{i,q}$, allocated to M_k. Since application tasks work only on local copies, their execution is not affected by memory contention. The global communication labels are allocated to the global memory M_{m+1}. Contention in the access to such labels is avoided by the LET communication mechanism at the price of a (predictable) synchronization delay. For the sake of simplicity, only data read and write operations are considered: possible improvements and optimizations are discussed at the end of this section.

A. LET as an opportunity to avoid memory contention

A major issue in executing real-time applications upon multicore platforms is the contention of architectural shared resources in the memory hierarchy (e.g., levels of caches and global memories). Works in the literature [5], [6] addressed such a problem by proposing clever solutions to improve the predictability of memory traffic.

As discussed in Section III-A, LET communication can be realized by scheduling data write and read operations at various time instants provided that the order of their execution preserves the specified causality. However, scheduling the communication phases at the beginning of the periodic instances of tasks, as illustrated in Figure 2, carries considerable benefits in controlling the memory traffic. In fact, this approach allows *localizing* the memory accesses within precise time windows that are determined by the task periods, and allows to arbitrate the access to the global memory M_{m+1}.

Consider a label ℓ that is read by two tasks executing upon two different cores. The tasks dispose of local copies that must be updated by the LET communication mechanism by reading the global copy of ℓ, which is mapped to the global memory M_{m+1}. Although the read operations can be performed in parallel on the two cores, each benefiting of a low latency in accessing the corresponding local memory where the private copies of ℓ are allocated, their timing is mutually coupled due to the potential contention in accessing the global memory. Without a proper synchronization mechanism, in the worst-case the memory accesses issued by one core can interfere with the other, and viceversa, leaving room for pathological scenarios that inevitably affect the tasks' response times. Conversely, by localizing the accesses to the global memory in precise time windows, the interference generated by memory contention can be avoided by design.

As a drawback, the execution of the communication phases at the release of periodic task instances requires specific jobs with priority higher than all the tasks. This determines a priority inversion, as the LET communication for a low priority task delays the execution of a high priority task. The impact of this drawback is discussed in the experimental results in Section VIII.

B. Timing of LET communications

First, it is necessary to identify the subset of memory accesses that are *required* to safely realize the LET paradigm.

Depending on the task periods, a producer does not need to always update the shared copies of the accessed labels at every periodic instance. For example, consider a producer task τ_P executed with a rate of $T_P = 2$ ms that is communicating with a consumer task τ_C running with a rate of $T_C = 10$ ms. Suppose also that both the tasks are synchronously released at the system startup. As a function of the ratio of their periods, for each job of τ_C there are $T_P/T_C = 5$ jobs of τ_P that overlap over time. For a given job J_C of τ_C, the data produced by the first four overlapping jobs of τ_P are never used by J_C, as they are overwritten by the data write operations performed by the *last* overlapping job, i.e., the last job of τ_C that completes no later than the release of the next job of τ_C (following J_C).

In a dual manner, a consumer does not always need to read the shared copies of the labels. By leveraging these observations, it is possible to derive an analytical characterization of the timing of LET communications.

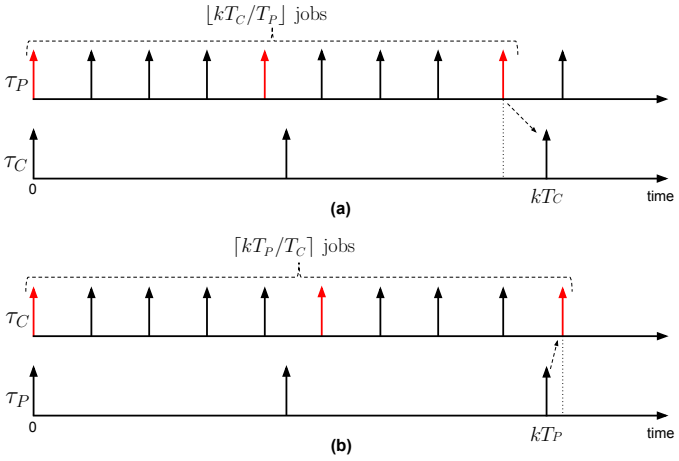

Fig. 5. Illustration of the timing of LET communications. Inset (a) depicts the case of write operations, while inset (b) depicts the case of read operations. To preserve the LET semantic, it is sufficient that only the jobs in red perform the update of (resp., the read from) global copies. Dashed arrows indicate the communications that involve the k^{th} job of the consumer task (inset (a)) and the producer task (inset (b)).

Timing of write operations. Consider a producer task τ_P communicating with a consumer task τ_C, both synchronously released at time $t = 0$. If the period of τ_P is larger than (or equal to) the period of τ_C, i.e., $T_P \geq T_C$, then a job of τ_C will always be released between two jobs of τ_P. As a consequence, the producer task must update the global copies of each label $\ell \in L^W(\tau_P, \tau_P)$ at *every* periodic instance. Otherwise, if $T_P < T_C$, then multiple jobs of τ_P can overlap with one job of τ_C. It is then sufficient that the global copies are updated by the last overlapping job that completes before the release of a job of τ_C. For the generic k^{th} job of τ_C, which is released at time kT_C, there are $\lfloor kT_C/T_P \rfloor$ periodic instances of τ_P that are fully contained within the time window $[0, kT_C]$. Hence, the job of interest is the one that completes at time $\lfloor kT_C/T_P \rfloor T_P$. This scenario is illustrated in Figure 5(a).

Generalizing such results, the jobs of τ_P that must update the global copies are those whose periodic instances complete at times $\eta^W_{C,P}(k) \cdot T_P$, for $k \in \mathbb{N}_{\geq 0}$, where

$$\eta^W_{C,P}(k) = \begin{cases} \left\lfloor \frac{kT_C}{T_P} \right\rfloor & \text{if } T_P < T_C, \\ k & \text{otherwise.} \end{cases} \quad (1)$$

Timing of read operations. Consider the same two tasks τ_P and τ_C. If the period of the consumer task is larger than

(or equal to) the one of the producer task, i.e., $T_C \geq T_P$, then a job of τ_P will surely be released between two jobs of τ_C. Consequently, the consumer task must read the global copies of each label $\ell \in L^R(\tau_C, \tau_P)$ at *every* periodic instance. Otherwise, if $T_C < T_P$, it is sufficient that the global copies are read by the *first* job of τ_C that is released after (or at) the release of a job of τ_P. Considering the generic k^{th} job of τ_P, released at time kT_P, there are $\lceil kT_P/T_C \rceil$ periodic instances of τ_C that overlap with the time window $[0, kT_P]$. Hence, the first job of τ_C released after (or at) time kT_P is the one released at time $\lceil kT_P/T_C \rceil T_C$, as shown in Figure 5(b).

Generalizing such results, the jobs of τ_C that must actually read the global copies are those activated at times $\eta^R_{C,P}(k) \cdot T_C$, for $k \in \mathbb{N}_{\geq 0}$, where

$$\eta^R_{C,P}(k) = \begin{cases} \left\lceil \frac{kT_P}{T_C} \right\rceil & \text{if } T_P > T_C, \\ k & \text{otherwise.} \end{cases} \quad (2)$$

When applying the properties identified above to every pair of communicating tasks in the systems, it is clear that LET communication requires a workload with *multiple periodic patterns* (i.e., one for each pair of communicating tasks), which can be realized with a *multiframe* task.

C. Deriving a multiframe task with inter-core synchronization

The generalized multiframe (GMF) task model [7] has been proposed to cope with computational activities that exhibit a variable behavior across multiple instances. Specifically, a GMF task is characterized by an ordered sequence of *frames* each defined by a WCET, an inter-arrival time to the next job, and a relative deadline. A GMF task releases jobs by following the cyclically repeating order of the frames.

The proposed approach to realize LET communication is based on the following design principles:

- **(i)** Synchronous activation of all the tasks in the system (i.e., all the tasks on all the cores are synchronously released at startup time $t = 0$).
- **(ii)** Definition of a GMF task τ_x^{LET} for each processor P_x that performs the copies of labels from the corresponding local memory to the global memory (write operations), and viceversa (read operations). Such tasks run at the highest priority.
- **(iii)** Adoption of an inter-core synchronization protocol to arbitrate the accesses to the global memory performed by each frame of the GMF tasks.

The results derived in the previous section can be leveraged to match principle (ii); that is, as a function of the timing of LET communications, it is possible to identify the time instants at which data write and read operations must be performed. Then, the operations that must be scheduled at the same time instant are merged into a frame of a GMF task. This strategy is summarized in the algorithm in Figure 6.

Given a processor P_x, the algorithm identifies all the time instants in which a producer task $\tau_P \in \Gamma_x$ must update the global copies of the accessed labels, hence writing in the global memory (lines 2-8). In a dual manner, the algorithm proceeds by identifying all the time instants in which a consumer task $\tau_C \in \Gamma_x$ must update its local copies of the accessed labels by reading from the global memory (lines 10-16). Finally, by means of the function called at line 17, the algorithm constructs

```
1:  procedure GENERATEGMFBEHAVIOR(P_x)
2:      for each (τ_C, τ_P) ∈ Γ × Γ_x do
3:          for each t_k = η^W_{C,P}(k) · T_P, k ∈ ℕ_{≥0} do
4:              for each ℓ_q ∈ L^W(τ_P, τ_C) do
5:                  schedule_write(t_k, ℓ_q = ℓ_{P,q})
6:              end for
7:          end for
8:      end for
9:
10:     for each (τ_C, τ_P) ∈ Γ_x × Γ do
11:         for each t_k = η^R_{C,P}(k) · T_C, k ∈ ℕ_{≥0} do
12:             for each ℓ_q ∈ L^R(τ_C, τ_P) do
13:                 schedule_read(t_k, ℓ_{C,q} = ℓ_q)
14:             end for
15:         end for
16:     end for
17:     build_frames()
18: end procedure
```

Fig. 6. Algorithm to generate the behavior of a GMF task that implements LET communications on processor P_x.

the frames of the GMF task by (i) looking at all times instants t_k for which there is at least one operation scheduled, and (ii) for each of such time instants, defining a frame that has as workload all the corresponding write operations followed by the read operations. The times t_k to be considered in the algorithm can be limited to the *hyperperiod* of all the tasks in the system. A practical implementation of the GMF tasks will be discussed in Section V.

To avoid contention in global memory, the execution of the GMF communication tasks on each core are strictly synchronized. Flow preservation requires that all tasks complete before the end of their period and that all writes are performed before the corresponding reads. To ensure that writes are performed before reads, the GMF communication tasks execute all their writes in a strict order (following principle (iii)). When all writes are completed, the GMF tasks execute the read operations in order. The order of execution may be different for each GFM execution instance, given that some GMF from some core may not need to read or write for a given periodic instance. The resulting protocol to regulate the access to global memory complies with the following rules:

- **R1** For each execution instance of a GMF task τ_x^{LET} released at time t_k, two sets of processors are defined: $W_x(t_k)$ and $R_x(t_k)$.
- **R2** Before performing the write operations scheduled in a frame, τ_x^{LET} must wait until all the write operations scheduled at time t_k for the GMF tasks of the processors in $W_x(t_k)$ are completed.
- **R3** Before performing the read operations scheduled in a frame, τ_x^{LET} must wait (i) that all the write operations scheduled at time t_k are completed, and (ii) that the read operations scheduled at time t_k in the GMF tasks of the processors in $R_x(t_k)$ are completed.
- **R4** The GMF tasks busy-wait to guarantee rules R2 and R3.

The corresponding pseudocode for the frames (instances) of the GMF tasks is illustrated in Figure 7.

The sets $W_x(t_k)$ and $R_x(t_k)$ determine the order with which the communication operations are performed. A simple definition for such sets can be devised by enforcing a fixed global order of processors, where some processors can be skipped when their corresponding GMF task does not have

```
1: procedure FRAME(P_x, t_k)
2:     wait(W_x(t_k))
3:     do_write_operations(t_k)
4:     wait_all_writes()
5:     wait(R_x(t_k))
6:     do_read_operations(t_k)
7: end procedure
```

Fig. 7. Pseudocode for the frame released at time t_k of the GMF task running on processor P_x.

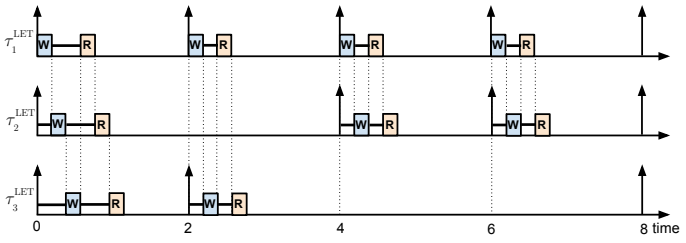

Fig. 8. Example schedule of three GMF tasks with inter-core synchronization. The GMF task on processor P_1 is fully periodic. The GMF task on processor P_1 has three frames released at times 0, 4, and 6, while the GMF task on processor P_2 has two frames released at times 0 and 2. For simplicity, the hyperperiod is 8 and all the frames execute just one write and one read operation.

to perform communication. The resulting scheme is a token passing with busy-waiting. A practical implementation of the proposed protocol is presented in Section V. An example schedule of the GMF tasks is illustrated in Figure 8, where the sets referred in rules R2 and R3 for the frames at time 0 are defined as follows: $W_1(0) = R_1(0) = \emptyset$, $W_2(0) = R_2(0) = \{P_1\}$, and $W_3(0) = R_3(0) = \{P_1, P_2\}$.

The proposed approach requires that the frames of the GMF tasks are sufficiently spaced, i.e., the minimum inter-arrival time between two frames released on any processor is significantly larger than the longest time a frame takes to complete all the communication operations.

D. Improvements and Optimizations

The proposed approach can be improved and optimized in several directions. First, it is possible to schedule tasks not affected by global reads and writes during the busy-waiting, hence improving their response times and the processor utilization. The same can be done to cope with control input and output operations as long as the adopted I/O devices are not shared by multiple processors. Second, the algorithm in Figure 6 can be improved to reduce the number of reads from global memory, e.g, when the same label is read by multiple tasks running on the same processor. Third, further parallelism in performing the copies of the labels can be achieved by adopting a DMA and a different label allocation. Lastly, different scheduling schemes can be devised to reduce the interference introduced by the GMF tasks, e.g., by deferring some communication operations.

V. Implementing LET on Aurix Tricore

This section presents an implementation of the approach proposed in the previous section on the popular Aurix Tricore platform by Infineon. The implementation has been performed upon the ERIKA open-source real-time operating system [8], which is certified OSEK/VDX and implements most of the AUTOSAR OS requirements. The code is publicly available on-line [9].

A. The Aurix Tricore platform

The Aurix Tricore is an automotive-grade multicore platform widely adopted as a main processing unit in several types of electronic control units (ECUs), such as for engine control. The Aurix Tricore includes three cores, each associated with a program memory interface (PMI) and a data memory interface (DMI) (Figure 9). The DMI includes a scratchpad memory (i.e., a local memory under the control of the programmer) and a set-associative data cache. The PMI includes a program scratchpad memory and program cache (i.e., to store instructions). The caches can be disabled. The microcontroller also includes a local memory unit (LMU) and a program memory unit (PMU). Despite the name, the LMU is a 32KB memory that is external to the core subsystems, and can be considered as a global memory. The PMU includes a 384KB data flash memory, and two 2MB program flash memories.

In the Aurix platform, the scratchpads are used as the core local memories in the abstract model of Section II, and the LMU is the global memory. Despite their names, the local scratchpads are accessible from any core. In the Aurix, the memory map of the microcontroller allows any core to access any of the above-mentioned memories by means of a cross-bar interconnect. The memory map is the same for all cores. As an example, Table I reports an excerpt from the memory map of the Aurix Tricore TC275 (taken from the corresponding datasheet [10]), which shows the addresses at which the scratchpads of the CPUs are accessible from any core.

TABLE I
Excerpt from the TC275 Memory Map

Address Range	Size	Description	Access Type	
			Read	Write
5000 0000 5001 DFFF	120 KByte	CPU2 Data Scratch-Pad	access	access
6000 0000 6001 DFFF	120 KByte	CPU1 Data Scratch-Pad	access	access
7000 0000 7001 BFFF	112 KByte	CPU0 Data Scratch-Pad	access	access

The access to a scratchpad memory of a remote core does not involve the global memory. The core subsystems are asymmetric. One of the three cores has a different CPU architecture than the others. The cores also differ in the sizes of the local memories. For instance, the first core has a 112KB data scratchpad, while the other two cores have a 120KB data scratchpad. At a high level, the abstract platform model introduced in Section II matches the architectural characteristics of the Aurix Tricore.

B. Implementation

The implementation of the approach proposed in Section IV-C required facing with three major issues: **(i)** the synchronization of the task activations, **(ii)** the efficient realization of GMF tasks, and **(iii)** the implementation of the inter-core synchronization protocol to explicitly regulate the access to the global memory.

Synchronizing the task activations. The first issue has been solved by exploiting the remote procedure call (RPC) features that are available in ERIKA. In accordance to the OSEK/VDX

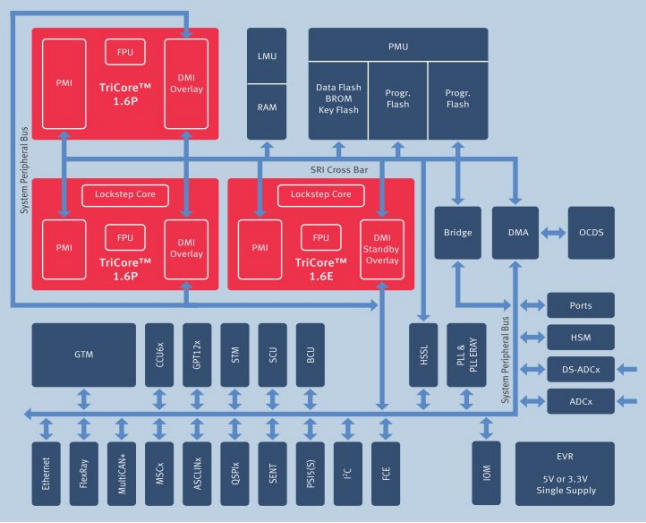

Fig. 9. Architecture of the Aurix Tricore microcontroller (from infineon.com).

standard, *alarms* ar provided to periodically activate tasks. In our implementation, all the alarms are driven by a single OSEK counter, which is realized with a timer that periodically sends interrupts to the first core (with a rate of one millisecond). Using ERIKA RPC features, such alarms can be used to activate tasks on any processor. Inter-core interrupts are leveraged to synchronously activate tasks on remote processors. Hence, in the resulting design, the first core is in charge of activating all the tasks in the system; synchronization of the tasks' periods is ensured because all task activations are generated by the same time reference, modulo some negligible synchronization delay introduced by the RPC mechanism.

Realizing GMF tasks. The realization of the GMF tasks required facing a memory vs. time trade-off. A straightforward implementation of the method proposed in Section IV-C would require the definition of a table that stores the set of labels to be read and written (or a pointer to a function performng the set of reads and writes) for each frame instance of the core GMF tasks up to the hyperperiod of all the tasks in the system. Each frame instance would be characterized by the release time (or the inter-arrival time to the next frame) and a code section with all the communication operations to be executed within the frame. While this choice would have a limited impact in terms of runtime overhead, it is memory eager for realistic applications. First, the required table may be very large for realistic values of the hyperperiod. Second, this method would require a lot of duplicated code among the code sections of the frames, i.e., there may be several frames that perform mostly (if not exactly) the same data write and read operations.

To contain the memory footprint when realizing the GMF tasks, the solution adopted in our implementation is based on providing two counters for each pair of communicating tasks: one for write operations, and one for read operations. Such counters can be used to identify the time instants in which the LET communications for a pair of tasks must be performed. For instance, consider a producer task τ_P communicating with a consumer task τ_C with $T_P < T_C$. By following the timing of LET communications derived in Section IV-B, it is possible to observe that the number of jobs of τ_P between each communication phase is known a-priori and given by Equation (1). Furthermore, note that Equation (1) produces values with a periodic pattern that is repeated every hyper-period of the two tasks: therefore, it is sufficient to consider only the values of $\eta^W_{C,P}(k)$ up to $k = lcm(T_P, T_C)/T_C$, where $lcm(a, b)$ denotes the least common multiple of a and b. Similar observations can also be made for read operations by considering Equation (2).

The key idea of our proposal is to use the counters to count the number of jobs that separate the communication phases. For each processor P_x, the corresponding GMF task has been implemented as a periodic task running with period T_x^{LET} equal to the MCD of the periods of all the tasks executing upon P_x. Each instance of such a task is in charge of decrementing all the above-mentioned counters. Each counter is associated with a code section that implements the corresponding communication phase. Such code sections are executed when the corresponding counter reaches zero, where the latter is re-initialized to the next value. This strategy can be realized with a code generator, as done for the case-study presented in Section VIII.

Inter-core synchronization. Finally, by exploiting the characteristics of the Aurix Tricore, it is possible to devise a lightweight implementation of inter-core synchronization. For each processor P_x, two atomic *spin variables* allocated to the corresponding local memory M_x are provided: one to wait for write operations, and another to wait for read operations. Such variables are initialized to zero. Each frame of a GMF task that has to wait before executing a communication phase (see the algorithm in Figure 7) performs the busy-wait by spinning in a loop executed as long as the spin variable is zero. Leveraging the feature of the Aurix Tricore that allows a core to write in the scratchpad of another core, it is possible to notify a GMF task that is spinning by simply updating one of its spin variables. Note that such notifications do not involve accesses to the global memory. Furthermore, the number of notifications issued in a given time window can be computed off-line as a function of the configuration of the GMF tasks. Since the platform includes a write buffer, the `DSYNC` instruction can be provided after the write on a remote spin variable to flush the write buffer, thus enforcing the consistency of the notification. The GMF tasks perform the busy-waiting by continuously accessing their corresponding local memory: hence, they do not generate memory traffic that compromises the arbitration of the accesses to the global memory. The actual spin variables to be used, and the GMF tasks to be notified, can change frame by frame depending on the desired order with which the cores must access the global memory.

C. Pseudocode of the GMF tasks

Figure 10 reports the pseudocode for GMF task τ_x^{LET} running on processor P_x. First, the function do_write_tick(), decrements all the counters associated with write operations. Then, the function invokes the write operations for the counters that are down to zero (implemented using a bitmask for each task). Subsequently, the task busy-waits on the spin variable spin_P_x_write (line 3) until another core signals its completion, i.e., passing to P_x the token to access the global memory. Once the token has been acquired, depending on the value of the bitmask of each task running on P_x, the set of scheduled write communications are executed by updating the global copies of the corresponding labels (line 5). Once the write phase is completed, another core is notified to proceed with its write

operations (line 6). A similar scheme is provided for read operations (lines 8-12).

To avoid memory interference due to the implementation of the multiframe mechanism, the data (i.e., the counters and the bitmasks) managed by the do_write_tick() and do_read_tick() functions must be allocated to the local memory M_x. Note the execution of such functions (by all the GMF tasks in system) is performed in parallel, thus reclaiming part of (or possibly even all) the time that a task has to busy-wait.

An example of the do_write_tick() function is reported in Figure 11, where the counter associated to a pair of communicating tasks is managed. At line 7, the function modifies the bitmask of a producer task τ_6 to notify that the communication labels read by a consumer task τ_8 must be updated within the current frame of the GMF task. Such operations will then be accomplished by the do_write() function of Figure 10.

```
1:  procedure LET_TASK_P_X( )
2:      do_write_tick()
3:      busy_wait( spin_P_x_write == 0 )
4:      spin_P_x_write = 0
5:      do_write()
6:      notify_next_processor_write()
7:
8:      do_read_tick()
9:      busy_wait( spin_P_x_read == 0 )
10:     spin_P_x_read = 0
11:     do_read()
12:     notify_next_processor_read()
13: end procedure
```

Fig. 10. Pseudocode for the GMF task τ_x^{LET} running on processor P_x.

```
1:  procedure DO_WRITE_TICK( )
2:      ⟨...⟩
3:      cnt_write_T6_T8 = cnt_write_T6_T8 -1
4:      if (cnt_write_T6_T8 == 0) then
5:          k_{6,8} = (k_{6,8} + 1) mod k_{6,8}^{max}
6:          cnt_write_T6_T8 = jobs_T6_T8[k_{6,8}] ·T_6/T_1^{LET}
7:          write_flags_T6 | = TURN_ON_FLAG_T6_T8
8:      end if
9:      ⟨...⟩
10: end procedure
```

Fig. 11. Example of function do_write_tick() showing the management of the counter for the pair of tasks τ_6 (producer) and τ_8 (consumer) with $T_P < T_C$. Variable $k_{6,8}$ is initialized to zero, jobs_T6_T8[k] = $\eta_{6,8}^W(k+1) - \eta_{6,8}^W(k)$, and $k_{6,8}^{max} = lcm(T_6, T_8)/T_8$.

VI. Worst-case analyses with and without LET

This section provides a comparison of possible approaches for bounding the worst-case cost of memory accesses in multicore platforms, including the blockings for contention in the case of any-time (in the context of the task execution) memory accesses and in the case of our proposed LET implementation.

A. Memory-aware response-time analysis for any-time accesses

Following standard response-time analysis, 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:

$$R_i = W_i + \sum_{\substack{\tau_j \in hp(\tau_i) \\ \tau_j \in \Gamma(\tau_i)}} \left\lceil \frac{R_i}{T_j} \right\rceil W_j + MC_i(R_i) \qquad (3)$$

where $W_i = C_i + \sum_{\ell_v \in \mathcal{L}_i} N_{i,v} \cdot \lambda_v$ (i.e., the worst-case execution time of the task plus the cost for accessing the 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.

Memory contention arises when tasks access to communication labels mapped to the global memory. 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. Rather, in this work an *inflation-free* analysis strategy [11], [12] is adopted.

An 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, a bound is derived for the maximum number of accesses $NRA_x(t)$ to the global memory issued by tasks executing on remote processors $P_x \neq P(\tau_i)$ in an *arbitrary* time window of length t, that is

$$NRA_x(t) = \sum_{\tau_j \in \Gamma_x} \sum_{\ell_v \in \mathcal{L}_j^G} \left\lceil \frac{t + R_j}{T_j} \right\rceil N_{j,v}. \qquad (4)$$

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 pending jobs of $\tau_j \in \Gamma_x$ in any time window of length t [11], [12].

Similarly, a bound is derived for the number of accesses $NLA_i(t)$ to the global memory issued by the local processor $P(\tau_i)$ in a *busy-period* of length t where τ_i is pending, that is

$$NLA_i(t) = N_{i,v} + \sum_{\substack{\tau_j \in hp(\tau_i) \\ \tau_j \in \Gamma(\tau_i)}} \sum_{\ell_v \in \mathcal{L}_j^G} \left\lceil \frac{t}{T_j} \right\rceil N_{j,v}. \qquad (5)$$

Due to the FIFO arbitration and the fact that the memory accesses are blocking and non-interruptible, it follows that each memory access issued by a remote processor can delay *at most* one access issued by the local processor. Hence, the following bound for the contention delay holds:

$$MC_i(t) = \sum_{P_x \neq P(\tau_i)} \min\{NRA_x(t), NLA_i(t)\} \cdot \lambda^R, \qquad (6)$$

where λ^R is the cost to access a label in global memory.

Equation (6) can be used in Equation (3) to bound the response times of the tasks. The term $NRA_{i,x}(t)$ 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 (3) 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.

B. Analyzing the proposed LET implementation

By construction, the proposed approach guarantees that all the application tasks execute without incurring in memory contention, as they only access local copies of the labels (allocated to the local scratchpad memories). However, they incur in temporal interference caused by the GMF tasks, which

execute at the highest priority. The interference generated by such tasks can be bounded with established analysis techniques for GMF tasks or more general task models: please refer to [13] for a detailed survey. Bounds on the execution times of the frames can be derived by accounting for the cost of accessing global and local memories, and the time necessary to manage the multiframe behavior (mainly related to functions do_write_tick() and do_read_tick()). Note that the time a frame has to wait before performing the communication actions is determined by the maximum between (i) the spinning time, which depends on the communication actions performed in the other cores, and (ii) the execution times of functions do_write_tick() and do_read_tick().

This approach allows to precisely account for the contention delay incurred by tasks, resulting in a definitively more predictable design compared with the case where the application tasks can access the global memory at any time during their execution, as coped by the analysis presented in the previous section. A fine-grained analysis of the GMF tasks is out of the scope of this paper, and is left as future work.

VII. Implementing LET in AUTOSAR

This section shortly addresses solutions for the implementation of the LET model in AUTOSAR.

In AUTOSAR, runnables communicate by using an API offered by an architecture layer called RTE (Run-time environment). For data-oriented communication, the API offers simple functions for writing to and reading from data objects. The API functions can be *explicit* or *implicit*. In the explicit model, the (shared) communication variable is accessed at the time it is needed (the API function is called) within the execution of the runnable, as it is illustrated at the top of Figure 12. In the implicit model, when a read or write operation is invoked by the runnable in the middle of its execution, the values are read from and written into local copies of the variables. The actual code implementing the read from and write into the shared global 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 copied by the RTE code in the actual global variable after the runnable execution (shown in the middle of Figure 12, the darker rectangles before and after the runnable execution represent the RTE code).

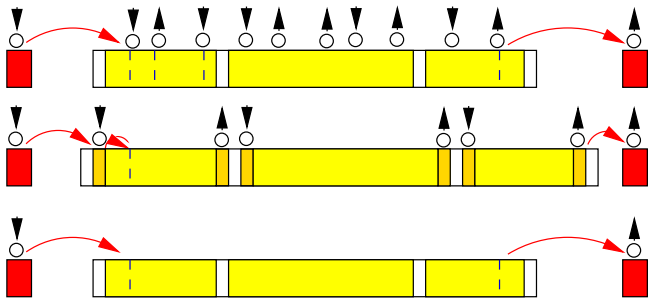

Fig. 12. Illustration of the implementation of the LET model of execution with explicit and implicit communication. Down- and up-arrows denote the input and output operations, respectively.

The simplest way to implement the LET communication paradigm in an AUTOSAR flow is to modify the RTE generation process for the implicit communication model. The RTE generator would add the code performing the global variables inputs and outpust to the multiframe LET tasks instead of placing it at the runnable boundaries (bottom of Figure 12). The RTE generator could generate the LET input and output tasks together with the other RTE-generated code (according to the algorithms outlined in this paper).

VIII. Experimental Evaluation: a Case-Study

This section reports on an experimental evaluation that has been conducted to assess the feasibility of the proposed approach and its impact in terms of timing performance. The LET implementation discussed in Section V has been adopted for a synthetic application that has been automatically generated from a model provided by Bosch for the WATERS 2017 challenge [2], which is representative of a realistic engine control application.

A. The WATERS 2017 challenge model

The WATERS 2017 challenge came with a model of an engine control application consisting of 1250 runnables grouped into 21 tasks/ISRs that access 10000 labels. About 5000 labels are constant, while the others are actual communication variables. The model specifies the labels accessed by each runnable, the type of access (read or write), and the number of accesses. Furthermore, it provides the execution times of the runnables net of memory access and memory contention times. The task periods and the minimum inter-arrival times of the ISRs are also provided. The model comprises a quad-core platform, where tasks are statically allocated.

B. Experimental setup

The tests have been performed on an Infineon TriBoard v2 equipped with an Aurix TC275 microcontroller running at 200MHz and connected to a Lauterbach PowerTrace to perform debugging and tracing. The HIGHTECH Aurix compiler v4.6.3.1 and the ERIKA real-time operating system v2.7 are used, with the default compiler configurations provided for the ERIKA kernel. Data caches have been disabled and the application code is fetched from the PMU (flash memories).

C. Assumptions and Code generator

Some additional assumptions were necessary to generate executable code from the WATERS challenge model. First, while the challenge model is conceived for a quad-core platform, only three cores are available in the Aurix platform. Consequently, one core and the corresponding tasks have been discarded. Second, since our proposals focus on fully-periodic tasks, ISRs have been considered as periodic tasks with rate obtained by rounding their minimum inter-arrival time to the closest multiple of one millisecond. Third, since the challenge model does not specify the memory access patterns (i.e., no runnable code structure is provided), two strategies have been tested: (i) *uniformly-distributed* memory accesses within each runnable with random order, (ii) *grouping* of all memory read operations at the beginning of the runnable, and all write operations at the end of the runnable. No conditional statements within the runnable code have been considered (this information was lacking in the challenge model).

TABLE II
NET EXECUTION TIME (WITHOUT KERNEL OVERHEAD) OF THE FIRST JOB, AND EXECUTION TIMES OF THE FIRST EIGHT JOBS OF THE GMF TASKS.

core	net execution time [μs]
1	3.8
2	108.76
3	148.2

core	execution times [μs]							
1	4.25	1.188	1.438	1.438	1.188	1.938	1.813	1.125
2	136	7.438	7.313	7.313	6.813	8.813	7.688	6.625
3	163.8	57.19	86.13	58.06	85.38	61.12	85.56	56.81

Based on such assumptions, a code generator has been developed. The generator inputs the XML file that encodes the system model and generates C code for each runnable where execution segments are realized with for loops including a `nop` operation in the body. Concerning the application, the generator also generates (i) the definition of all the labels (both the local and the global copies), (ii) the corresponding accesses within the runnable code, (iii) the tasks' code (to call a sequence of runnables), (iv) the OIL configuration for the operating system, and (v) the code to setup the OSEK alarms to periodically activate the tasks.

Furthermore, the generator is in charge of generating the code of the GMF tasks as discussed in Section V starting from the information available in the challenge model (i.e., communication relationships between tasks and task periods). The periods of the GMF tasks (implementing the LET communication) have been configured to the MCD of the periods of the tasks running in the corresponding processors, which resulted in $T_1^{\text{LET}} = 1$ ms, $T_2^{\text{LET}} = 1$ ms, and $T_3^{\text{LET}} = 10$ ms. The inter-core synchronization protocol has been configured with a fixed order between the cores: first P_2, then P_3, and lastly P_1. A core is skipped if it does not release a frame, i.e., P_1 waits for P_3 only one every 10 jobs (note that the period ratio of the corresponding GMF tasks is actually 10). This order has been chosen with the following rationale. As discussed in Section V, the first core is responsible for activating all the tasks, hence it is subject to the highest runtime overhead related to the OSEK alarms. If the first core would be the first one in accessing the global memory, then it would delay all the GMF tasks in the other cores by the time it takes to manage the activation of all the tasks (note that the kernel functionality are executed with higher priority than the tasks). Letting the first core to be the last one in accessing the global memory also determines the benefit that, when managing the task activations, it can reclaim some of the time it would have to busy-wait.

D. Experimental results

Experiments have been performed to measure the execution time of the GMF tasks implementing the LET communication. The results are reported in Table II. The table on the left reports the net execution times of the first frames without the kernel overhead. Note that the first frame is analogous to the one executed at the tasks' hyperperiod, where all the LET communications are performed, and is the heaviest in terms of execution time. Collecting the net execution times for all the frames was beyond the capability of our tracing hardware due to the limited trace buffer of the microprocessor. The execution times, including the kernel overhead, for the first eight frames are reported in the table on the right. The GMF tasks require a relatively small processor utilization (the GMF task of the third core runs at 10 milliseconds). However, as it can be observed from the measurements, the interference generated by the GMF may be harmful for latency-sensitive tasks in the second core (with the first frame) and in the third core. On the other hand, it is important to recall that LET communication introduces the benefit of controlling the accesses to the shared memory

TABLE III
APPLICATION FOOTPRINT WITH AND WITHOUT LET (IN BYTES)

	text	data	bss
LET	393064	4904	88328
Explicit	359872	4784	80752

To better evaluate the overall impact on the tasks' timing performance, the response times of twelve representative tasks of the challenge model have been measured. Both the cases with LET communication and with direct access to the global memory (AUTOSAR explicit communication) have been tested. The two memory access patterns discussed in Section VIII-C have been tested, but no significant difference has been observed. The longest observed response times, normalized to the corresponding task period, for the case of read-execute-write patterns are reported in Figure 13. As it can be observed from the figure, the response times differ by very small amounts. These results demonstrate that LET communication—with all the benefits that it brings in terms of predictability of the timing of control outputs and end-to-end latencies—can be realized without harming the timing of the application with respect to the case of direct accesses to the global memory, which by definition lacks of the benefit provided by LET. We believe that evident benefits in terms of reduced memory contention have not been observed because the tested application is not sufficiently memory-intensive and the limited number of runs with variable execution times (because of the problems in tracing the execution and the limited available time) was not sufficient to explore cases with multiple memory contentions.

The major impact of the realization of LET has been found in terms of memory footprint, which increased by the 7.5% (about 40KB) with respect to the case of AUTOSAR explicit communication. See Table III for the detailed results.

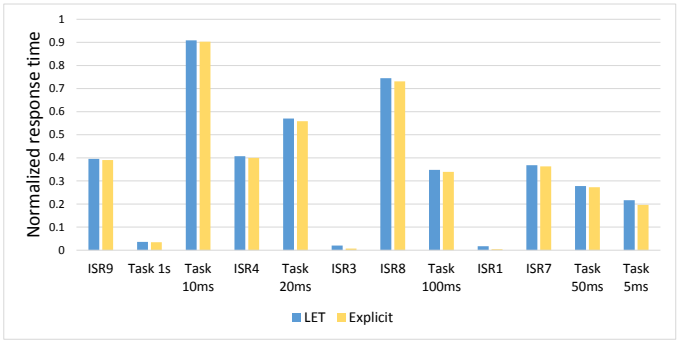

Fig. 13. Longest observed normalized response times under both LET and AUTOSAR explicit communication.

IX. Related Work

The benefits of the LET paradigm for automotive applications have been outlined by Hamann et al. [14], together with an analysis of the end-to-end latencies of communicating tasks that make use of the LET paradigm. However, the authors considered a different implementation model with respect to the one adopted in the present paper, nor they took advantage of the possibility of explicitly controlling the accesses to global memory. Rather, they propose communication mechanisms to guarantee the LET communication flows that are similar to those proposed to guarantee flow preservation in synchronous systems [15]–[17].

Several efforts have been spent in developing techniques to improve the predictability of memory accesses in multicore platforms, but none of them took into consideration the LET paradigm nor adopted the inter-core synchronization scheme proposed in this work. Most close to the present paper, Tabish et al. [18] presented an OS-level technique to preload scratchpad memories (data and instruction) to enable a contention-free non-preemptive execution of tasks. In 2011, Pellizzoni et al. [19] proposed the PREM execution, where tasks access memory only at the beginning at the end of their jobs. Yao et al. [20] presented a scheduling technique to arbitrate with time-division multiplexing the memory accesses performed by PREM tasks.

Alternative approaches have been proposed to regulate the access to shared DRAM memories. Yun et al. [5] proposed a memory bandwidth reservation mechanism that exploits hardware performance counters, while Yun et al. [6] and Kim et al. [21] presented bank-aware memory allocation schemes. Techniques have also been proposed to improve the predictability of cache memories: please refer to the excellent survey by Gracioli et al. [22].

Finally, other authors proposed schedulability analysis techniques that explicitly take into account the memory contention. Most relevant to us are the works of Mancuso et al. [23], which proposed a WCET bound in the presence of a collection of resource management techniques developed within the single-core equivalence project at UIUC, and Davis et al. [24], which adopted a trace-based task model and proposed to account for contention delays at the stage of response-time analysis.

X. Conclusions and Future Work

We presented a scheme for the practical implementation of the LET execution model in multicores. We discussed the benefits arising from the use of LET not only in terms of a predictable model of computation with deterministic output times, but also the potential for scheduling memory accesses avoiding excessive contention. An actual implementation on an automotive platform has been presented and its implementation issues have been discussed. Overall, it emerged that the realization of LET communication requires facing with several challenging design problems, which should possibly be integrated in a holistic synthesis methodology that optimizes the communication infrastructure for a given application. This observation lays the foundations for very interesting future works.

Acknowledgments

The authors like to thank Giuseppe Serano, Errico Guidieri, and Paolo Gai of Evidence S.R.L. for the valuable support provided in setting up the experimental setup, Pasquale Buonocunto, Paolo Pazzaglia, and Alessio Balsini from the ReTiS lab for their work on the parser for the WATERS challenge model, and Infineon for having provided the microcontroller platform.

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

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

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

[5] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 55–64.

[6] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014.

[7] S. Baruah, D. Chen, S. Gorinsky, and A. Mok, "Generalized multiframe tasks," *Real-Time Systems*, vol. 17, no. 1, pp. 5–22, Jul 1999.

[8] ERIKA Enterprise: Open-source RTOS OSEK/VDX kernel. [Online]. Available: http://erika.tuxfamily.org

[9] [Online]. Available: http://retis.sssup.it/~a.biondi/LET/

[10] *AURIX TC27x D-Step - User's Manual, V2.2 2014-12*.

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

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

[13] M. Stigge and W. Yi, "Graph-based models for real-time workload: a survey," *Real-Time Systems*, vol. 51, no. 5, pp. 602–636, Sep 2015.

[14] A. Hamann, D. Dasari, S. Kramer, M. Pressler, and F. Wurst, "Communication Centric Design in Complex Automotive Embedded Systems," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, vol. 76, 2017.

[15] C. Sofronis, S. Tripakis, and P. Caspi, "A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling," in *EMSOFT Conference, Seoul, Korea*, October 2225, 2006.

[16] 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.

[17] 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.

[18] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo, "A real-time scratchpad-centric OS for multi-core embedded systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016.

[19] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for COTS-based embedded systems," in *17th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2011.

[20] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo, "Memory-centric scheduling for multicore hard real-time systems," *Real-Time Systems*, vol. 48, no. 6, pp. 681–715, Nov 2012.

[21] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding and reducing memory interference in COTS-based multi-core systems," *Real-Time Systems*, vol. 52, no. 3, pp. 356–395, May 2016.

[22] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," *ACM Comput. Surv.*, vol. 48, no. 2, Nov. 2015.

[23] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun, "WCET(m) estimation in multi-core systems using Single Core Equivalence," in *ECRTS*, 2015.

[24] R. I. Davis, S. Altmeyer, L. S. Indrusiak, C. Maiza, V. Nelis, and J. Reineke, "An extensible framework for multicore response time analysis," *Real-Time Systems*, Jul 2017.