Optimizing the Functional Deployment on Multicore Platforms with Logical Execution Time

Paolo Pazzaglia, Alessandro Biondi, and Marco Di Natale
Scuola Superiore Sant’Anna
Pisa, Italy
Email: {paolo.pazzaglia, alessandro.biondi, marco}@sssup.it

Abstract—The move to multicore systems requires methods and tools to support the designer in the partitioning of functions among the available cores and the definition of the task model. In this paper we present the formulation of a functional partitioning for real-time systems and we provide an optimization method for an efficient implementation of the Logical Execution Time (LET) paradigm, to enforce causality and determinism in the development of time- and safety-critical applications. A novel schedulability analysis for partitioned tasks executing according to the LET paradigm is also provided. Our methods are applied to the industry-size model of the WATERS challenge and compute solutions that easily outperform the initial solution provided.

Index Terms—LET paradigm; Multicore partitioning; MILP

I. INTRODUCTION

Multicore platforms are replacing conventional single-core architectures in many embedded application domains, requesting the developers to find effective ways to partition functions on the cores. This is true for new systems, and also when porting an existing (single-core) application onto a new platform. The functional allocation may require to partition the existing tasks in a different way or to redefine them, and drives the allocation of the memory that is required by the tasks for computation and communication. In addition, when partitioning a single core application on a multicore, the causal execution order that was previously guaranteed by the priority assignment (and the ensuing scheduling order) may not hold anymore. When causality must be preserved, the move to multicore platforms requires additional mechanisms to synchronize the execution of the functions and the transfer of data. The Logical Execution Time (LET) paradigm [1], [2] is among the solutions that are more popular with the industry (especially automotive), since it offers an execution model that allows for selective intervention and minimal change to the application code.

In this work, we assume availability of an application model consisting of runnables (functions) and tasks executing in accordance with the automotive AUTOSAR standard. Runnables communicate data over shared memory locations, identified as labels. The objective of this paper is then to find an optimal mapping for both runnables and labels of the given application on a multicore platform using the LET paradigm, where the optimality metric is the minimization of the worst-case response time among all the tasks (related to the robustness and extensibility of the mapping and scheduling solution). The final system must satisfy all the temporal constraints (deadlines), as well as preserve the correct behavior (causality) of the functional model of the runnables. This means that every functional dependency of the original system must be guaranteed in every possible execution of the multicore system.

The proposed solution leverages the LET paradigm and the definition of (multiple) synchronization points to achieve both determinism and flexibility in parallelizing the application. An accurate response-time analysis of the resulting architecture is developed and presented. We provide an optimization method based on a formulation of the problem as a mixed-integer linear programming (MILP) mathematical optimization, where the functional dependencies between runnables are used as constraints for the placement of functions and tasks, as well as the placement of labels in memory. As a term of comparison, we also developed an optimization solution based on the application of a genetic algorithm.

II. STATE OF THE ART

The optimal placement of computing functions on parallel processing nodes is among the most researched problems in computer science. For computing functions (or tasks) with real-time constraints (deadlines), the problem is often solved by using heuristics, stochastic optimization techniques, or mixed-integer linear programming, possibly in isolation or even combined among them. For distributed systems with end-to-end deadlines, optimal placement (including priority and period assignments) is computed using genetic algorithms in [3]. Heuristics are used in [4], and a SAT-based approach is proposed in [5] for task and message placement. A schedulability analysis and a partitioning algorithm for parallel tasks without preemptions is presented in [6].

In [2] the problem is discussed when the input is an AUTOSAR model of runnables and the causal order of execution must be preserved. The parallelization of runnables in concurrent platforms under several constraints for reusability (maintaining the task structure) is discussed in [7].

In the context of automotive applications, the LET paradigm receives an increasing interest from both industry and research centers [8]. The LET execution model (originally proposed in the Giotto framework [1]) is used (after minor adaptations to consider task chains) by Hamann et al. [9] to restore a causal order of execution when moving to multicore. The optimization of the memory buffers that are required for the
implementation of the LET communication in the case of oversampling or undersampling reuses the concepts and methods that were originally proposed to guarantee flow preservation in synchronous systems [10], [11], [12]. In [13], the authors propose an approach for mapping legacy code on multicores leveraging clustering heuristics and an implementation of the LET paradigm using the Timing Division Language. However, a formal analysis and the details about the case study application are missing.

Finally, the move to multicore systems requires an analysis of the cost of the accesses to the memory shared at all levels among the cores to provide for determinism. A survey on the techniques that can be used to improve the predictability when accessing cache memories is in [14]. Mancuso et al. [15], propose methods to compute safe WCET bounds when shared memory banks in a multicore platform are managed according to one of several possible resource management techniques (the work was developed within the single-core equivalence project at UIUC). In the trace-based task model proposed in [16], contention delays when accessing memory are included in the formulation of the worst-case response-time. Tabish et al. [17] presented an OS-level technique to preload scratchpad memories (data and instruction) to enable a contention-free non-preemptive execution of tasks. In the PREM execution model [18], memory accesses are only allowed at the beginning and at the end of each job, hence can still happen at variable time instants (depending on scheduling). Conversely, under LET, accesses to memories occur within specific time windows, e.g., the periodic activation times of tasks. Yao et al. [19] presented a scheduling technique to arbitrate with time-division multiplexing the memory accesses performed by PREM tasks.

Alternative approaches to achieve predictable accesses to shared DRAM memories include the memory bandwidth reservation mechanism presented in [20] implemented using hardware performance counters, and the bank-aware memory allocation schemes in [21] and [22].

III. SYSTEM MODEL AND DEFINITIONS

This section introduces the system model, the main definitions, and the notation used throughout the paper. The system model considered here is inspired by the one described in the WATERS 2017 challenge [23], which is representative of a typical control software application in the automotive domain and de facto equivalent to the AUTOSAR standard [24].

A. Task set model

We consider a real-time application composed of $N$ periodic tasks scheduled by fixed-priority scheduling. Each task is denoted by $\Gamma_i$, where the index $i = 1, \ldots , N$ also indicates its priority (i.e., $\Gamma_i$ has higher priority than $\Gamma_j$ if $i < j$). Tasks may be periodic or sporadic: $T_i$ denotes the period (or minimum inter-arrival time if sporadic) of $\Gamma_i$, and $D_i$ its relative deadline. Each execution instance of a task is a job. The response time of each job is the time span between its activation and its completion. We assume implicit deadline tasks, i.e. $D_i = T_i$, $\forall \Gamma_i$, and hard deadline requirements for all tasks.

In agreement with the AUTOSAR standard [24], each task is described as an ordered sequence of runnables (i.e., functions) atomically allocated to it. $R(\Gamma_i)$ is the set of all the runnables in $\Gamma_i$. Each runnable $r \in R(\Gamma_i)$ inherits the same period/inter-arrival time and priority of its task $\Gamma_i$. Runnables read from and write data in variables $\ell$, defined as labels in the original model. Reading and writing may happen anywhere during the runnable execution, and the same label may be accessed multiple times. Each runnable $r_j$ is characterized by a worst-case execution time (WCET) $c_j$ that (i) does not include the time required to access labels in memory, and (ii) corresponds to the case in which the runnable runs in isolation (i.e., without contention on shared memories). For each runnable $r$, the sets $L^R(r)$ and $L^W(r)$ denote the collection of labels that are read and written by $r$, respectively. To ease the presentation of the following results, we assume that all labels have the same size: the results can easily be generalized to the case of labels with heterogeneous sizes, by assigning a different weight to each label, in terms of time spent to read and write it.

B. Communication between runnables

We classify the labels based on the number of runnables that read or write them, identifying (i) read-only, (ii) write-only, and (iii) shared labels.

- **Read-only labels** represent constant values and are read by only one runnable.
- **Write-only labels** represent sinks of measurements (e.g., data shown on control panels/interfaces) and are written by only one runnable.
- **Shared labels** represent variables that are both read and written (i.e., implement communications between functions). They can be read by multiple runnables, but are written by one runnable only.

For shared labels, we refer to the writer runnable as producer, and the ones that read from the label as consumers. In general, a runnable can be both a producer and a consumer with respect to different labels. Data exchange among runnables is formalized in the definition of messages.

**Definition 1 (Message $m$).** A message, denoted with a triplet $m = \{r_p, r_c, t_a\}$, represents a communication between two runnables $r_p$ and $r_c$ via $t_a$, where the producer $r_p$ writes the label $t_a$, and the consumer $r_c$ reads from it.

Messages are classified as inter- or intra-task.

- An inter-task message $m^{EC}$ is exchanged between a producer $r_p$ and a consumer $r_c$ belonging to different tasks (i.e., $r_p \in R(\Gamma_i)$ and $r_c \in R(\Gamma_j)$ with $i \neq j$).
- An intra-task message $m^I$ is exchanged between runnables $r_p$ and $r_c$ belonging to the same task.

Communications are associated with causal dependencies between runnables. Intra-task messages can be further characterized by their timing properties. When the data written by the producer $r_p$ must be read by a consumer $r_c$ executed within the same job, the message is defined as immediate. If the data produced by $r_p$ within the $k$-th job of the task is read by $r_c$
at the next (i.e., \((k+1)\)-th) job, then the message is delayed (immediate and delayed properties can be inferred, e.g., by the execution order of the runnables \(r_p\) and \(r_c\) within the task). A typical use of delayed messages is in digital control feedback loops, where the computed control command is fed back to the actuator with a delay of one step. A particular case of delayed message concerns the case where a runnable reads and writes the same label (e.g., an integrator updating its internal state); in this case, the output value will be its own input in the next job and hence creates a loop message. A visual representation of the different types of intra-task messages is in Fig. 1.

A sequence of runnables communicating through labels is a chain. Chains represent flows of data with immediate causal dependencies, and may be associated with time constraints. An example is a control system composed of multiple functions, where sensor data are filtered, merged, and provided to a job \((k+1)\)-th job

where sensor data are filtered, merged, and provided to a job \((k+1)\)-th job
[49x432]where sensor data are filtered, merged, and provided to a job \((k+1)\)-th job
[49x432]where sensor data are filtered, merged, and provided to a job \((k+1)\)-th job
[49x432]where sensor data are filtered, merged, and provided to a job \((k+1)\)-th job

\(\text{LET}\) paradigm

A typical issue when moving an application from single core to multicore is the loss of causality and the introduction of non-determinism because of the higher parallelism and the possibility of new execution traces. The logical execution time (LET) paradigm, originally introduced as a component of the Giotto framework [1], has gained attention in recent years as a viable candidate for enforcing determinism in multicore applications [2].

According to LET, all communications between tasks are performed only at specific points in time. All inputs to a task (from shared variables) are read at the beginning of a time interval (usually the task period itself), while all outputs are made available to the other tasks only at the end of it, regardless of when they are actually produced. LET communication can be seen as a sample and hold mechanism, where all modifications done to shared values are delayed at the end of the task period: the response time jitter is thus traded for a fixed latency of one period between input and output. This design is used to abstract from the actual response time of each job, providing time determinism and a predictable execution model.

The LET paradigm can be leveraged to schedule memory accesses at well-defined points in time, avoiding the possibility of contention [26]. The implementation of LET requires creating local copies for all the labels involved in LET communication. In this way, runnables only access the memory local to their core. In our framework, the LET copy between shared labels and corresponding local copies is delegated to a dedicated LET communication task \(\Gamma_{L}^{p}\), provided in each core \(P_{p}\). The LET task \(\Gamma_{L}^{p}\) has the sole purpose of copying LET communication variables used by tasks in core \(P_{p}\), and it is executed with highest priority. As such, the LET communication executed on behalf of a low-priority task can generate interference on the execution of a high-priority task.

The definition of the parameters of \(\Gamma_{L}^{p}\) needs to take into account if the core executes sporadic or periodic tasks. In the former case, the LET communication task is designed as a periodic task with period less than or equal to the shortest inter-arrival time of any task in execution on the core to avoid any possible data loss. At every periodic activation, \(\Gamma_{L}^{p}\) copies all the variables that have been updated by the sporadic tasks since their last activation.

In cores executing only periodic tasks (set \(\mathcal{P}^{P}\)), an instance of the LET communication task is executed at specific points in time. Differently from the original formulation of the LET paradigm in [1], we consider the execution of the LET tasks to be synchronized across all cores. This is because, in contrast with [1], we also require the capability of enforcing causality in computation chains. The corresponding time instants when the LET communication task is released are then called synchronization points.
Definition 2 (Synchronization point). A synchronization point is a time instant, common to a subset of all cores, when the LET communication tasks in execution on those cores are synchronously released to update a given subset of shared labels.

In this work, we allow for multiple synchronization points during the execution of a periodic task. The number of synchronization points of periodic task $\Gamma_i$ is denoted by $N_{i}^{S} \geq 1$. Hereafter, we refer to the synchronization points of $\Gamma_i$ using the symbol $s_{i,k}$, where the subscript $k \in [1,N_{i}^{S}]$ is the index of the synchronization point, indicating also their ordering. We consider that the last synchronization point of each task occurs at the deadline, i.e., $s_{i,N_{i}^{S}} = T_i$. It follows that, for each job of $\Gamma_i$, its activation instant is a synchronization point too, since it coincides with $s_{i,N_{i}^{S}}$ of the previous job. On each processor $P_p$, the instants $s_{i,k}$ defined for the container tasks $\Gamma_{i}^{p}$ occur synchronously, i.e., all container tasks execute within the same synchronization points on all cores.

As a consequence, the execution of each container task $\Gamma_{i}^{p}$ is now partitioned into $N_{i}^{S}$ LET intervals, each spanning two synchronization points (one at the beginning and one at the end). An arbitrary $k$-th LET interval of $\Gamma_{i}^{p}$ corresponds then to the interval $[s_{i,k-1},s_{i,k}]$, where $s_{i,0}$ refers to the activation instant of the job. In each LET interval, a subset of the runnables mapped in the container task is executed. Figure 2 shows an example in which a task $\Gamma_i$ in the original system model with its runnables (with their internal causal dependencies shown as arrows) is partitioned for execution in three cores, each with a synchronized container task and a set of synchronization points delimiting three LET intervals in the task execution period. The introduction of multiple synchronization points helps reducing the end-to-end latency in chains and improves parallelism in multicore systems while enforcing intra-task dependencies between runnables [2].

The synchronization points delimiting the LET interval define time barriers or deadlines bounding the execution of the runnables mapped in the interval. A runnable $r$ mapped to the $k$-th interval of the container tasks $\Gamma_{i}^{p}$ must complete its execution no later than $s_{i,k+1}$ time units after the task release.

E. Problem definition

Our objective is to partition the runnables among the cores and allocate the labels on the available memories. The definition of the LET intervals allows to enforce the synchronization of shared data within chains of runnables spanning across several cores, hence enforcing causality, predictable data flows, and deterministic end-to-end delays along the chains. To this end, in our design we assume that all inter-task messages are implemented with LET communication. Furthermore, intra-task messages are implemented using LET only if the runnables in the message are mapped to different cores. Intra-task messages exchanged within the same core are implemented using simple shared variables for the labels. The design problem faced in this work can be summarized as follows:

- For each periodic task $\Gamma_i$: (i) the corresponding container tasks $\Gamma_{i}^{p}$, provided in all cores $P_p \in \mathcal{P}^{p}$, are split into $N_{i}^{S}$ LET intervals; and (ii) each of the corresponding runnables $r \in \mathcal{R}(\Gamma_i)$ must be allocated to one (and only one) LET interval of one (and only one) container task $\Gamma_{i}^{p}$.
- Sporadic tasks are mapped as a whole (i.e., all their runnables on the same core) in one of the cores of $\mathcal{P}^{S}$.

Following the LET implementation of [25], the labels associated with the messages handled using LET are allocated in global memory, while the others are mapped in the local memory of the (unique) core that accesses them. The allocation of the local copies of the labels using the LET communication is discussed in Section IV-E.

In order to guarantee the preservation of the data flows, the causal dependencies in all chains must be enforced and preserved once the system is deployed on a multicore platform. Furthermore, the resulting mapping must guarantee the timing constraints (i.e., deadlines) of all jobs. These constraints are formalized and addressed in the following section, where the proposed design strategy is also presented.

IV. MULTICORE LET DESIGN

A. Characterizing LET communication tasks

The LET communication tasks $\Gamma_{i}^{p}$ are in charge of copying data from local copies of the labels to the actual (shared) labels allocated in global memory, and vice-versa. In order to respect causality and to be compliant with the original LET semantic proposed in [1], each instance of this task must always (i) first update the shared labels with the content of their local copies (write phase), and then (ii) copy new data from shared labels to their local copies (read phase). This order must be preserved across all cores: that is, when multiple cores perform LET communications at the same synchronization point, they need to wait for all cores to finish writing their data via LET communication before starting the read phase.

The LET tasks may incur in contention when simultaneously accessing the global memory. To address this issue, we adopt the synchronization scheme proposed in [26] where the global memory is accessed via a baton-passing protocol with busy waiting. Experiments on the Aurix-Tricore TC277 platform...
showed that the proposed strategy has a very limited overhead (in the order of a few microseconds) [26].

This synchronization scheme requires the definition of a priority ordering among the processors \( P_i \). In this work, we assume that the core indexes reflect this ordering, i.e., \( P_i \) reads/writes before \( P_j \) if \( i < j \). Figure 3 shows a simple example of a schedule involving LET communication between two cores using the synchronization scheme of [26]. For sake of clarity, and to make this paper self-consistent, the behavior of LET tasks based on the synchronization scheme of [26] is summarized:

1. At each synchronization point \( t \), for each processor \( P_p \in \mathcal{P}^P \), an instance of \( \Gamma^P_L \) is released. If multiple synchronization points for different tasks occur synchronously, a single instance of \( \Gamma^P_L \) executes, performing all the copies.
2. The instance of \( \Gamma^1_L \) corresponding to the first processor \( P_1 \) starts writing the data from local copies to shared copies, following the causality order of the tasks in \( P_1 \) that require a LET write in \( t \) (if any), while the other processors busy-wait. When \( \Gamma^1_L \) finishes, the LET writes of the next processor are performed, and so on until all LET tasks \( \Gamma^P_L \) in all processors finish writing.
3. Next, starting again from \( P_1 \), \( \Gamma^1_L \) reads the shared LET data needed by the local tasks (if any), following the causal order. The next processor starts the reading phase of its LET task once the previous one finishes. A processor that completes its LET reads may then start (re-)executing the application tasks.

**B. Skipping unnecessary LET communications**

To reduce the communication burden managed by \( \Gamma^P_L \), label reads and writes are only performed when strictly required to realize the LET paradigm. When the producer and consumer runnables have different periods, if consumers are undersampled or executing with a rate lower than the producer, the producer may skip some updates of the shared labels whenever the data will be overwritten before being used by any reader. In a dual manner, if a consumer is oversampled with respect to the the producer, it may skip some updates of the local copies of labels if they have not been updated since their last access. The reads/writes that can be skipped are not periodic (see [26]) and it is difficult to formulate the access patterns analytically for the purpose of optimization. Hence, an approximate (and pessimistic) evaluation is used.

For each pair of runnables and shared label \( \ell \), an oversampling factor \( \sigma \) represents the minimum number of jobs that separate two consecutive LET communications on \( \ell \) that cannot be skipped to preserve the LET semantics. The oversampling factor is different for the consumer and producer runnables.

**Consumers.** Considering an arbitrary (both intra- or inter-task) message \( m = \{r_p, r_c, \ell_a\} \), the oversampling factor \( \sigma^R(r_c, \ell_a) \) of the consumer runnable \( r_c \) reading \( \ell_a \) is given by

\[
\sigma^R(r_c, \ell_a) = \max \left\{ \left| \frac{T_p}{T_c} \right|, 1 \right\},
\]

where \( T_p \) is the period of the producer \( r_p \) and \( T_c \) is the period of the consumer \( r_c \). Here, if \( T_c < T_p \), the consumer is faster than the producer. The minimum number of jobs between two necessary LET reads is equal to the maximum number of complete periods of the consumer runnable in one period of the producer, which is computed as \( \left| \frac{T_p}{T_c} \right| \). On the other hand, if \( T_c \geq T_p \), then the producer is faster (or with same rate) than the consumer; for this reason the reads are always mandatory and the oversampling factor is set to 1.

**Producers.** The oversampling factor of a producer runnable \( r_p \), writing label \( \ell_a \), denoted by \( \sigma^W(r_p, \ell_a) \), has to cope with all the possible consumers of \( \ell_a \). Defining \( \rho(r_p, \ell_a) \) as the set of messages for which \( r_p \) is the producer runnable and \( \ell_a \) the shared label, \( \sigma^W(r_p, \ell_a) \) is given by

\[
\sigma^W(r_p, \ell_a) = \max \left\{ \min_{m \in \rho(r_p, \ell_a)} \left\{ \left| \frac{T_c}{T_p} \right| \right\}, 1 \right\},
\]

where \( T_c \) and \( T_p \) are the periods of the consumer and producer runnables involved in message \( m \), respectively. In this case, if a consumer is slower than the producer (i.e., \( T_c > T_p \)), the value \( \left| \frac{T_c}{T_p} \right| \) represents the number of instances of the producer runnable completely contained in one period of the consumer. The minimum number of jobs between two required LET writes is equal to the minimum number of complete periods among all its consumers. If at least one consumer has period less than or equal to the producer, writing is always required and the oversampling factor is set to 1.

**C. Precedence constraints among runnables**

The partitioning of runnables among the cores must be defined in accordance with the partial order of execution of the runnables defined by the set of intra- and inter-task messages. Overall, the precedence constraints among the runnables of each task can be effectively described by a directed acyclic graph (DAG). The precedence constraints among runnables can be computed with the following rules (as in [2]), which directly follows from the definitions of immediate and delayed messages introduced in Section III-B.
**Definition 3** (Precedence rules). Given an intra-task message \( m = \{r_p, c_r, \ell_a\} \):

1) If \( m \) is an immediate message, then each job of \( r_p \) must complete writing data on \( \ell_a \) before \( c_r \) reads it. This precedence relation is denoted as \( r_p \prec c_r \).
2) If \( m \) is a delayed message, with \( r_p \neq c_r \), then \( c_r \) must read \( \ell_a \) before \( r_p \) overwrites it in the same job. This precedence relation is denoted as \( c_r \prec^+ r_p \).

Trivially, loop messages (i.e., delayed messages with \( r_p = c_r \)) do not introduce any precedence constraint.

**D. Runnable mapping rules**

In our framework, each container task \( \Gamma^p \) is split into an ordered sequence of \( N^p \) LET intervals. Note that the ordering between the intervals implies that all the runnables allocated to the \( k \)-th interval will be executed before the ones allocated to the \((k+1)\)-th one. As a consequence, the allocation of runnables to the intervals must take into account the precedence constraints discussed in the previous sub-section. To this end, a set of assignment rules must be defined to allocate each runnable to an interval.

**Rule R1.** Consider a pair of runnables \((r_p, c_r)\) such that \( r_p \prec c_r \). If both runnables are hosted on the same core (i.e., they are assigned to the same container task), their execution can be serialized by guaranteeing that \( r_p \) is executed before \( c_r \) in the same interval, or \( r_p \) executes in an interval with lower index than the interval of \( c_r \).

**Rule R2.** Consider a pair \((r_p, c_r)\) such that \( r_p \prec c_r \). If \( r_p \) and \( c_r \) are mapped to different cores, then their communication is realized with LET (see Sec. III-E). Hence, the execution of the two runnables must be separated by a synchronization point. Consequently, if \( r_p \) is mapped in the \( k \)-th LET interval, then \( c_r \) must be mapped in an interval with index \( k' \geq k \).

**Rule R3.** Consider a pair \((r_p, c_r)\) such that \( r_c \prec^+ r_p \). If both runnables are hosted on the same processor, then \( c_r \) executes before \( r_p \) in the same interval, or in an interval with lower index than the interval of \( r_p \).

**Rule R4.** Consider a pair \((r_p, c_r)\) such that \( r_c \prec^+ r_p \). If the two runnables are mapped in different cores, then the communication is realized with LET. The consumed data is updated at the start of the LET interval in which the runnable executes, and the produced data is updated at its end. Hence, if \( c_r \) is mapped in the \( k \)-th LET interval, \( r_p \) must be mapped in an interval with index \( k' \geq k \).

**E. Assigning labels to memories**

Besides the allocation of runnables, the deployment of an application to a multicore platform must also map the labels to the available memories. The labels to be allocated are the ones originally defined for the application plus the local copies that are required by our LET implementation. The following rules define the allocation of the labels to the available memories as a function of the allocation of the runnables that access them.

**Rule L1.** As read-only labels are accessed by a single runnable \( r \) (see Sec. III-B), they are mapped in the local memory of the core in which \( r \) is allocated to.

**Rule L2.** The same of Rule L1 holds for write-only labels, which are also accessed by a single runnable (see Sec. III-B).

**Rule L3.** The labels corresponding to loop messages, i.e., data written and read by the same runnable \( r \), are mapped in the local memory of the core to which \( r \) is allocated.

**Rule L4.** Consider a shared label \( \ell \) involved in intra-task messages only.

1) If and only if all runnables that access \( \ell \) are mapped to the same core \( P_k \), then \( \ell \) is mapped in local memory \( M_k \) (no LET communication is used).
2) Otherwise, LET communication is used, and \( \ell \) is mapped in global memory. As each label has at most one producer, \( \ell \) is accessed by only runnables of the same task \( \Gamma_i \). A local copy \( \ell^k \) of \( \ell \) is created for each core \( P_k \) that hosts at least one runnable \( r \in R(\Gamma_i) \) accessing \( \ell \). The local copy \( \ell^k \) is mapped to memory \( M_k \).

**Rule L5.** Consider a shared label \( \ell \) involved in at least one inter-task message. \( \ell \) is always mapped in global memory as it requires to be managed with LET communication. Then, for each pair of core \( P_k \) and task \( \Gamma_i \), a local copy \( \ell^{k:i} \) of \( \ell \) is created if there exists at least one runnable \( r \in R(\Gamma_i) \) accessing \( \ell \) and allocated to core \( P_k \). The local copy \( \ell^{k:i} \) is mapped to memory \( M_k \).

By construction, the allocation of labels following rules L1-L5 as above guarantees the following properties:

1) each runnable mapped in \( P_k \) accesses only labels mapped in the corresponding local memory \( M_k \);
2) each local copy mapped in \( M_k \), corresponding to a shared label involved in LET communication, is exclusively accessed by runnables of a single container task \( \Gamma^k_i \); and
3) the LET communication task \( \Gamma^p_i \) of core \( P_k \) accesses only labels in memory \( M_k \) and in global memory \( M_G \).

Rules L1-L5 imply a set of constraints on the memory space requirement [27] of each memory, i.e., the memories must be large enough to host all the labels allocated to them. Some of the above rules may be relaxed whenever these constraints cannot be matched at the stage of optimization (e.g., in the presence of small local memories). This option is not addressed in this work and is left as future work.

**V. RESPONSE-TIME ANALYSIS**

According to the LET design presented in the previous section, each synchronization point \( s_{i,k} \) is treated as a passive barrier at which an instance of the LET communication task may be invoked. In order to achieve data consistency, the execution of all runnables mapped to the LET interval immediately preceding \( s_{i,k} \) must complete before the synchronization point. A timing analysis is required to verify this condition.

In the following, a response-time analysis is derived for each LET interval. The analysis leverages the observation that the proposed task design can be modeled as a special case of
the transactional task model proposed in [28], which consists of an ordered sequence of sub-tasks (called children tasks) activated with the same period, but released with different offsets. Indeed, each container task $\Gamma^p_i$ of our model can be mapped to a transaction task where each LET interval of $\Gamma^p_i$ is assigned to a child task of the transaction. The offsets of the transaction tasks correspond to the synchronization points of $\Gamma^p_i$. This observation is formalized as follows.

**Definition 4** (Child task $\tau$). A child task of $\Gamma^p_i$, denoted as $\tau_{i,k}$, with $k \in [1, N^p_i]$, is a periodic task with the same priority of $\Gamma^p_i$, period $T_i$, deadline $D_{i,k} = (s_{i,k} - s_{i,k-1})$, and offset $\phi_{i,k}$, such that $\phi_{i,1} = 0$ and $\phi_{i,k} = s_{i,k-1}$ for $k > 1$.

The relative deadline of each child task $\tau_{i,k}$ coincides with the synchronization point $s_{i,k}$, and corresponds to the activation time of the following child task.

The body of each child task $\tau_{i,k}$ is composed by the subset of runnables mapped in the corresponding $k$-th LET interval of $\Gamma^p_i$. By extending the notation introduced in Sec. III-A to improve readability, the set of runnables executed by a child task $\tau_{i,k}$ is denoted as $\mathcal{R}(\tau_{i,k}) \subseteq \mathcal{R}(\Gamma_i)$. The actual WCET of $\tau_{i,k}$ can then be computed as the sum of the WCETs of the runnables $r \in \mathcal{R}(\tau_{i,k})$, plus the cost of all the memory accesses for the labels in local memory used by the runnables.

$$C^p_{i,k} = \sum_{r_j \in \mathcal{R}(\tau_{i,k})} (e_j + \sum_{\ell_a \in \mathcal{L}(r_j)} A_{j,a} \cdot \lambda_{\ell_a}),$$  \hspace{1cm} (3)

where $\mathcal{L}(r) = \mathcal{L}^W(r) \cup \mathcal{L}^R(r)$, and $A_{j,a}$ is the maximum number of accesses by runnable $r_j$ to label $\ell_a$ in one job. Trivially, if the corresponding LET interval does not contain any runnable, $C^p_{i,k} = 0$.

This model transformation is used in the following for the purpose of response-time analysis.

A. Seeking the worst-case condition

In the general case, during an arbitrary time interval $[0, t]$, the execution of a child task $\tau_{i,k}$ running in processor $P_p$ may be interfered by

- the execution of runnables mapped to high-priority container tasks of the same core $P_p$;
- the LET communication related to the synchronization points in $[0, t]$ of all other container tasks (both with higher and lower priority) running in $P_p$ — this is because the LET task $\Gamma^p_L$ implementing the communication runs with highest priority; and
- the LET communication of tasks running on the other processors $P_q \neq P_p$ whenever the LET task of $P_p$ is busy waiting for their completion — this is required to arbitrate the accesses to the global memory and to preserve the LET semantics (see Sec. IV-A).

By building upon Theorem 2 in [28], it is possible to identify a release pattern that allows bounding the worst-case response time of a child task.

**Theorem 1.** The worst-case interference produced in an arbitrary interval $[0, t]$ by all the container tasks $\Gamma^p_j$, with $q = 1, \ldots, N_P$, to a child task $\tau_{i,k}^p$, with $j \neq i$, cannot be larger than the one generated when the following conditions occur in $[0, t]$:

1) $\tau_{i,k}^p$ is synchronously released at time 0 together with one of the synchronization points of $\Gamma^p_j$;
2) the LET communications related to the first $N^p_j$ synchronization points in $[0, t]$ of all container tasks $\Gamma^p_j$, for $q = 1, \ldots, N_P$, are not skipped.

**Proof.** At each synchronization point related to $\Gamma^p_j$, (i) a child task of $\Gamma^p_j$ is released and (ii) a job of the LET communication task $\Gamma^p_L$ corresponding to that synchronization point is released in all processors. Consider first the interference generated by the children tasks of $\Gamma^p_j$. Following Definition 4, a child task can be treated as a task with a given offset and no release jitter. Note that this corresponds to a particular instance of the transactional task model in [28]. As a consequence, condition (1) of theorem directly follows from Theorem 2 in [28].

Now, consider the interference generated by LET communications. Following following Secs. III-D and IV-A, note that the activations of the LET tasks $\Gamma^p_j$, and the work they perform in favor of $\Gamma^p_j$, within a time interval $[0, t]$ is directly implied by the release pattern of the child tasks of $\Gamma^p_j$ in $[0, t]$, and no other pattern exists. Furthermore, due to the inter-core synchronization discussed in Sec. IV-A, the release of all LET tasks $\Gamma^p_L$ is synchronized. The workload of the LET task $\Gamma^p_L$ on behalf of each child task of $\Gamma^p_j$ consists in a set of communications to be periodically performed (see Section IV-B). This is true also for the children tasks of the other cores $P_q \neq P_p$. Since these communications happen periodically, there exists a job of $\Gamma^p_j$ for which all communications required by all children tasks must be updated. Therefore, the interference contribution due to LET communications can be studied as the one generated by a classical periodic task, whose worst case corresponds to the case of synchronous release at time 0. This implies that the first LET communication at the beginning of the analysis interval is not skipped, which corresponds to condition (2) of the theorem. Hence the theorem follows. $\square$

B. Worst-case response time for children tasks

An upper bound on the worst-case response time of each child task $\tau_{i,k}$ can be computed by leveraging Theorem 1. Indeed, the theorem allows bounding the interference generated by a group of container tasks to a child task under analysis: hence, by summing up the interference contribution of each group of container tasks, it is possible to obtain a bound on the overall interference suffered by $\tau_{i,k}^p$. Since the LET tasks run at highest priority, also the LET communication related to lower-priority container tasks may generate interference to $\tau_{i,k}^p$.

Hence, the interference bound implied by Theorem 1 must consider all the container tasks (except the one to which $\tau_{i,k}^p$ belongs) independently of their priorities.

First of all, following the scheduling scenario of Theorem 1, a child task $\tau_{i,k}^p$ of $\Gamma^p_j$ is synchronously released with $\tau_{i,k}^j$, as in the example of Figure 4. We define $\Phi_{j,q,s}$ as the offset of an
arbitrary child task $\tau^p_{j,k}$ of $\Gamma^p_j$ with respect to the critical instant in which $\tau^p_{j,k}$ is released. Formally, $\Phi_{j,q,s}$ can be computed as follows:

$$\Phi_{j,q,s} = (\phi_{j,q} - \phi_{j,s} + T_j) \mod T_j.$$  

(4)

To address the analysis problem we distinguish between execution interference and LET interference: the former is generated by the execution of a high-priority container task when it preempta a lower-priority task, while the latter is generated by the LET task (running at the highest priority). Note that high-priority tasks contribute with both the types of interference.

1) Execution interference: A function $I^p_{j,s}(t)$ is introduced to denote the worst-case execution interference generated by $\Gamma^p_j$ in $[0,t]$ when $\tau^p_{j,s}$ is the child task released at the critical instant (time 0), as indicated by Theorem 1. $I^p_{j,s}(t)$ is meaningful only when $\Gamma^p_j$ has higher priority than the child-task under analysis $\tau^p_{i,k}$ and is mapped to the same processor $P_p$. The execution interference can be computed as in [28], i.e.,

$$I^p_{j,s}(t) = \sum_{q=1}^{N^p} \left[ \frac{\Delta_{j,q,s}(t)}{T_j} \right] C_{j,q}.$$  

(5)

where

$$\Delta_{j,q,s}(t) = \begin{cases} t - \Phi_{j,q,s} & \text{if } t - \Phi_{j,q,s} > 0, \\ 0 & \text{otherwise.} \end{cases}$$  

(6)

2) LET interference: The analysis of the LET interference is an original contribution, since it has not been addressed in any previous work. In the scheduling scenario defined by Theorem 1, consider the time interval $[0,t]$, and a child task $\tau^p_{j,s}$ released on core $P_p$ at time 0. The LET interference $L^p_{j,s}(t)$ generated by the container tasks $\Gamma^q_j$ ($q = 1, \ldots, N_P$) when $\tau^p_{j,s}$ is the child task of $\Gamma^q_j$ released at time 0, consists of the following terms:

- the LET interference $L^{p}_{j,s}(t)$ generated by all the label writes performed by the LET task $\Gamma^p_j$ on behalf of the runnables in $\Gamma^q_j$, within the interval $[0,t]$;
- the LET interference generated by the busy-waiting of $\Gamma^p_j$, which depends on the interference generated by the LET tasks in execution on the remote cores $P_q$ ($q \neq p$).

To simplify the presentation, we introduce the subsets $\mathcal{L}^W_L(r) \subseteq \mathcal{L}^W_W(r)$ and $\mathcal{L}^R_L(r) \subseteq \mathcal{L}^R_W(r)$ of labels that are written and read by an arbitrary runnable $r$ using LET communication. Note that each pair $(r, \ell)$, with $r \in \mathcal{R}(\tau^p_{j,s})$ and $\ell \in \mathcal{L}^R_L(r)$, generates a contribution to the LET interference, which can be analyzed as the one generated by a sporadic task with minimum inter-arrival time $\sigma^{R}_r(t) \cdot T_j$ (according to the analysis of Section IV-B). The WCET of the equivalent sporadic task can be computed by considering the set of memory operations that are required for a LET label read: (i) one access to the global memory (to read the shared copy); and (ii) one access to a local memory (to write the local copy that is read by the runnable). According to the platform model of Section III-C, the WCET of this copy (both for reading and writing) issued by processor $P_p$ is $c^p_L = \lambda_{pr,G} + \lambda_p$. This latter term can be used to bound the LET interference in the scheduling scenario of Theorem 1.

**Lemma 1.** Consider the scheduling scenario in $[0,t]$ of Theorem 1 and let $\tau^p_{j,s}$ be the child task of $\Gamma^p_j$ that is released at the critical instant $t = 0$. The total LET interference generated by the reads performed by the LET task on core $P_p$ on behalf of $\Gamma^p_j$ is given by

$$L^p_{j,s}(t) = \sum_{q=1}^{N^p} \sum_{r \in \mathcal{R}(\tau^p_{j,s})} \sum_{\ell \in \mathcal{L}^R_L(r)} \left[ \frac{\Delta_{j,q,s}(t)}{\sigma^{R}_r(t) \cdot T_j} \right] c^p_L.$$  

(7)

**Proof.** Consider a child task $\tau^p_{j,q}$ and let $\tau^p_{j,s}$ be the child task of $\Gamma^p_j$ released at time 0. Each pair $(r, \ell)$, with $r \in \mathcal{R}(\tau^p_{j,q})$ and $\ell \in \mathcal{L}^R_L(r)$, requires one LET read with cost $c^p_L$ for each activation of $\tau^p_{j,q}$ in $[0,t]$. Leveraging the formulation of Equation (5) and the oversampling factor for $(r, \ell)$ (Section IV-B), the number of activations in the interval is bounded by $[\Delta_{j,q,s}(t)/(\sigma^{R}_r(t) \cdot T_j)]$. The total contribution of $\Gamma^p_j$ can be computed using the same formulation for each $\tau^p_{j,q}$ with $q = 1, \ldots, N^p$, and iterating through all the label reads $(r, \ell)$ performed using LET of each $\tau^p_{j,s}$. Thus, the lemma follows. □

A similar reasoning applies to the case of producer runnables. Indeed, the contribution of each pair $(r, \ell)$, with $r \in \mathcal{R}(\tau^p_{j,q})$ and $\ell \in \mathcal{L}^W(r)$, to the LET interference is equivalent to the one a sporadic task with minimum inter-arrival time $\sigma^{W}_r(t) \cdot T_j$. The only difference with respect to Lemma 1 is that each synchronization point in the window of interest $[0,t]$ corresponds to the writes of labels produced by the runnables allocated in the LET interval that precedes the synchronization point. For this reason, $\mathcal{R}(\mathcal{pr}(\tau^p_{j,q}))$ is introduced to denote the set of runnables of the child task immediately preceding $\tau^p_{j,q}$, i.e., $\mathcal{R}(\mathcal{pr}(\tau^p_{j,q})) = \mathcal{R}(\tau^p_{j,q-1})$ for $q > 1$.
The worst-case response time for the child task equation as:

\[ R_i(t) = \Gamma_i \text{ as in Equation (7) and (8):} \]

\[ \sum_{h=1}^{N_p} \sum_{j=1}^{N_p} \sum_{s} \left[ \frac{\Delta_{j,q,s}(t)}{\sigma_{i,j}(r,t)} \cdot T_j \right] P_{i,j}^p \cdot L_r^h \cdot L_{w}^h. \]  

(8)

We now account for interference related to the busy-waiting originated by other container tasks \( \Gamma_j^p \) mapped in remote cores \( P_h \) (\( h \neq p \)). Due to the baton-passing protocol described in Section IV-A, the busy-waiting is composed of (i) the writing phases of all remote cores plus (ii) the reading phases of remote cores \( P_h \) executed before the reads in \( P_p \) (\( h < p \)). Note that the resulting interference can be computed by Equations (7) and (8) applied to processor \( P_q \), i.e., \( \sum_{h=1}^{N_p} \sum_{j \neq h} \sum_{s} \left[ \frac{\Delta_{j,q,s}(t)}{\sigma_{i,j}(r,t)} \cdot T_j \right] P_{i,j}^p \cdot L_r^h \cdot L_{w}^h. \) By adding to the last equation the LET interference generated on the core \( P_h \) under analysis, it is finally possible to get the total LET interference:

\[ L_y^h(t) = \sum_{h=1}^{N_p} \sum_{j \neq h} \sum_{s} \left[ \frac{\Delta_{j,q,s}(t)}{\sigma_{i,j}(r,t)} \cdot T_j \right] P_{i,j}^p \cdot L_r^h \cdot L_{w}^h. \]  

(9)

3) Total interference (execution and LET) on children tasks:

The worst-case interference experienced by \( \tau_i^p \) in an interval \([0, t]\), from both higher- and lower-priority tasks is denoted as \( IW_i^p(t) \) and computed as follows.

**Lemma 2.** The interference incurred by a child task \( \tau_i^p \) on core \( P_p \) due to high- and low-priority tasks in an interval \([0, t]\) is bounded by

\[ IW_i^p(t) = \sum_{j<i} \max_s \left( I_{i,j}^p + L_y^h + L_r^h \right) \sum_{j>i} \max_s L_y^h. \]  

(10)

**Proof.** The container tasks \( \Gamma_j^p \) with higher priority than \( \tau_i^p \) \( (j < i) \) can generate both execution and LET interference. The term \( I_{i,j}^p(t) + L_y^h + L_r^h \) bounds the execution and LET interference generated by \( \Gamma_j^p \) when the child task \( \tau_i^p \) is synchronously released with \( \tau_i^p \). Since only one child task \( \tau_i^p \) can be synchronously activated with \( \tau_i^p \), maximizing over all the possible children tasks \( \tau_i^p \) yields a safe bound on the maximum interference generated by \( \Gamma_j^p \). The same holds for container tasks \( \Gamma_j^p \) with lower priority than \( \tau_i^p \) \( (j > i) \), which can only generate LET interference (the second term only includes \( L_y^h \)). Hence, the lemma follows. \( \square \)

As the very last step, the interference \( I_{i,k}^p \) generated by the LET communication of the child task \( \tau_i^p \) itself, follows in a similar way as in Equations (7) and (8):

\[ I_{i,k}^p = \sum_{q=1}^{N_p} \sum_{r \in \mathcal{R}(\tau_i^p)} \sum_{t \in \mathcal{C}_L^q(r)} \sum_{i,k}^p \sum_{r \in \mathcal{C}_L^q(r)} \sum_{i,k}^p. \]  

(11)

The worst-case response time for the standard fixed-point equation as:

\[ R_i^p = \min_{t>0} \left\{ t > 0 \mid t = C_i^p + I_{i,k}^p + IW_i^p(t) \right\}. \]  

(12)

A multicore deployment solution is feasible if \( R_i^p \leq D_i \) for each child task \( \tau_i^p \) in the system.

VI. MILP FORMULATION OF THE OPTIMIZATION PROBLEM

In this section, we present a mixed-integer linear programming (MILP) formulation of the design problem faced in this work. The proposed formulation leverages the constraints and the analysis defined in the previous sections, with the objective of computing a mapping that minimizes the response time of each child task. The MILP formulation requires that the response-time analysis is converted in a set of linear constraints, with an objective function to be minimized. In the following, the main assumptions and (conservative) approximations are presented. The resulting formulation has a sufficiently low computational complexity to allow application to quite large systems, while being slightly more pessimistic than the one proposed in the previous section. An application to a realistic case study is then presented in Section VII.

A. Linearizing the response-time analysis

The response-time analysis in Equation (12) requires the knowledge of the subset of runnables, labels, and communications associated to each LET interval. Moreover, it includes some non-linearities, like the ceiling terms used to compute the number of activations of each child-task in the interval under analysis. An approximate linearized analysis must be derived in order to implement it as MILP.

1) Equally-spaced synchronization points: As a first assumption, in our MILP formulation the number of synchronization points \( N_i^S \) for each container task \( \Gamma_i \) is a known constant parameter. These points are chosen such that each container task of \( \Gamma_i \) in every processor is divided in \( N_i^S \) LET intervals of equal size \( T_i/N_i^S \). With this choice, each child task \( \tau_i^p \) belonging to \( \Gamma_i \) has the same deadline, computed as \( D_i = T_i/N_i^S \), while its offset can be formulated as \( \phi_{i,k} = (k-1)T_i/N_i^S \).

2) Limiting the schedulability test to a small number of checkpoints: Encoding the response-time analysis in Eq. (12) as a recursive equation in a MILP is extremely inefficient, due to the need of multiple integer variables to model the ceiling terms. This is especially true for large-scale applications. An alternative approach, which is sufficient-only, yet extremely accurate, is proposed here, building upon the results of [29]. Starting from a formulation of the problem as a schedulability test for \( \tau_i^p \), defined as:

\[ \exists t \in \mathcal{T}_i \mid R_i^p = C_i^p + I_{i,k}^p + IW_i^p(t) \leq t, \]  

(13)

we find a small set \( \mathcal{T}_i \) of checkpoints, which allows to obtain a sufficient-only test with accuracy extremely close to the exact one. For each pair \((i, j)\), with \( j < i \), we compute a time point in \( \mathcal{T}_i \) as \( t_{i,j} = \lfloor D_i/D_j \rfloor D_j \) (a time instant that is a candidate for the computation of the response time and the feasibility test). \( t_{i,j} \) corresponds to the activation of the last interfering child-task belonging to an arbitrary higher priority \( \Gamma_j^p \), before the deadline \( D_{i,k} \) of \( \tau_i^p \), when considering the critical instant activations of Theorem 1 for the worst-case interference conditions. The set of checkpoints used for
the schedulability test of $\tau^p_{i,k}$ (including also $D_{i,k} = \overline{D}_i$) is computed as $T_i = \bigcup_{j \in \mathcal{T}_i} \{t_{i,j}\} \cup \{\overline{D}_i\}$. Please refer to [29] for further details on this analysis approximation.

Note that, since all synchronization points are equally spaced, the checkpoints $t_{i,j} \in T_i$ are independent of the child task under analysis, and the child task $\tau^p_{j,s}$ released synchronously with $\tau^p_{i,k}$; therefore, they do not depend on indexes $s$ and $k$.

3) Objective function: The optimization problem requires finding the mapping that minimizes the maximum response time of any child task $R^p_{i,k}$, relative to its deadline $D_{i,k}$, which is referred to as $R/D$ ratio. Consider the formulation in Eq. (12). The interference term $IW^p_i(t)$ is independent of the index $k$ of the child task $\tau^p_{i,k}$ under analysis. Thus, since $D_{i,k} = \overline{D}_i$, $\forall k$, the child task with the largest value for $(C^p_{i,k} + IL^p_{i,k})$ will have the greatest $R/D$ ratio among all those in $\Gamma^p_i$. The response-time analysis can then be formulated as follows:

$$ R^p_{i,k} \leq \overline{R}^p_i = \max_{i,j \in \mathcal{T}_i} \left\{ C^p_{i,k} + IL^p_{i,k} \right\} + IW^p_i(t^*) \quad (14) $$

1) Labels involved in LET communication: The variable LETCa introduced above to define if $\ell_a$ requires LET communication is easily determined in the case of inter-task messages, i.e., LETCa = 1. Conversely, if $\ell_a$ is involved in an intra-task message, it requires LET communication only if its producer and consumer runnables are mapped on different cores.

Constraint 1. $\forall m^l \in \{r_i, r_j, \ell_a\}$, \( \text{LETCa} \geq |RC_{i,p} - RC_{j,p}| \)

Proof. If $r_i$ and $r_j$ are on the same core, they do not need LET communication and $\text{RC}_{i,p} = \text{RC}_{j,p}$, $\forall P_p$, thus their difference will be always equal to zero; otherwise, the difference $\text{RC}_{i,p} - \text{RC}_{j,p}$ is equal to 1 for the core on which $r_i$ is executed and -1 for the core on which $r_j$ is mapped, thus LETCa = 1.

2) Runnable mapping rules: An integer variable $\text{RBin}_{j,p} \geq 0$ is introduced $\forall r_j$, representing the index of the LET interval where $r_j$ is mapped (regardless of the core). This variable is related to the boolean $\text{RI}_{j,p,k}$ with the following constraint:

Constraint 2. $\forall r_j, \sum_{p} \sum_{a=1}^{N} \left(\text{RI}_{j,p,k} \cdot k\right) = \text{RBin}_{j,p}$

Proof. $\text{RI}_{j,p,k}$ is equal to 1 only for the $k$-th LET interval of the core $P_p$ where $r_j$ is mapped. For this reason, if $\text{RI}_{j,p,k} = 1$ then $\text{RI}_{j,p,k} \cdot k = k$ and all other addends are null.

The rules of Section IV-C are then simply expressed in our MILP formulation as follows:

Constraint 3. (Rules R1, R2:) $\forall m^l \in \{r_p, r_c, \ell_a\}$ with $r_p \prec r_c$, $\text{RBin}_p \leq \text{RBin}_c - \text{LETCa}$

Constraint 4. (Rules R3, R4:) $\forall m^l \in \{r_p, r_c, \ell_a\}$ with $r_c \prec plus r_p$, $\text{RBin}_c \leq \text{RBin}_p$

3) Assigning labels to memories: We define $\text{LMA}_{a,p} \in \mathbb{B}$ and $\text{LGA}_a \in \mathbb{B}$ to represent the mapping of label $\ell_a$ in memory $M_p$, or in $M_G$, respectively. Each label is assigned to one and only one memory, thus trivially \( \forall \ell_a, \sum \text{LMA}_{a,p} + \text{LGA}_a = 1 \). The mapping rules for labels are here expressed as MILP constraints.

Constraint 5. (Rules L1, L2, L3:) $\forall r_j$ and $\forall \ell_a \in \mathcal{L}(r_j)$ where $\ell_a$ is read-only, write-only, or involved a loop message, $\forall P_p$, \( \text{LMA}_{a,p} = \text{RC}_{j,p} \).

Constraint 6. (Rules L4, L5:) $\forall \ell_a$ involved in a message, $\text{LGA}_a = \text{LETCa}$.

4) Response time computation: The response-time analysis presented in the previous section is transformed using inequalities as lower bounds for WCETs, interference, and response time, instead of equalities. This is particularly effective for reducing the number of constraints in the MILP formulation (where equalities need both $\leq$ and $\geq$): indeed, since we are minimizing the response time, the solver will naturally move towards the minimum values that satisfy the lower bounds.

The WCET of child task $\tau^p_{i,k}$, presented in Equation (3), is lower-bounded as follows:
**Constraint 7.** From Equation (3), \( \forall \Gamma_i, \forall P_p, \forall k: \)
\[
C_{i,k}^p \geq \sum_{r_j \in \mathcal{R}(\Gamma_i)} R_{i,j,p,k} \left( t_j + \sum_{\ell_a \in \mathcal{L}(r_j)} A_{j,a} \lambda_p \right). \tag{16}
\]

Following the same rationale, all the LET interference functions can be formulated accordingly. As a representative example, the constraint used to compute \( IL_{i,k}^p \) is presented.

**Constraint 8.** From Equation (11), \( \forall \Gamma_i, \forall P_p, \forall k: \)
\[
IL_{i,k}^p \geq \sum_{q=1}^{N_p} \sum_{r_j \in \mathcal{R}(\Gamma_i)} \sum_{\ell_a \in \mathcal{L}^W(r_j)} L_{i,j,q,k-1,a} \cdot c_L^q + \sum_{p=1}^{P} \sum_{r_j \in \mathcal{R}(\Gamma_i)} \sum_{\ell_a \in \mathcal{L}(r_j)} L_{i,j,q,k,a} \cdot c_L \tag{17}
\]

Finally, from Equation (14), the response time \( \hat{R}_i^p \) is chosen from the set of response time candidates computed as \( RT_{i,j,p} \geq \max [C_{i,k}^p + IL_{i,k}^p] + IW_{i,j}^p(t_{i,j}) \), using the following constraints:

**Constraint 9.** \( \forall P_i^p \) not empty,
- \( \sum_{i,j,p} a_{i,j,p} \geq 1 \)
- \( \hat{R}_i^p \geq RT_{i,j,p} - (1 - a_{i,j,p}) \cdot \text{bigM} \)
- \( \hat{R}_i^p \leq t_{i,j} + (1 - a_{i,j,p}) \cdot \text{bigM} \)

where \( \text{bigM} \) is a sufficiently-large positive constant value to represent infinity, and \( a_{i,j,p} \in \mathbb{B} \) is an auxiliary variable.

**Proof.** An auxiliary variable \( a_{i,j,p} \) is associated to each checkpoint \( t_{i,j} \) such that \( a_{i,j,p} = 1 \) if the schedulability condition of Eq. (13) holds for \( t_{i,j} \). In order to guarantee the schedulability of all the children tasks in \( \Gamma_i^p \), the inequality of Eq. (13) must be verified for at least one checkpoint \( t_{i,j} \). This is enforced by the first inequality in the constraint. If \( a_{i,j,p} = 1 \), the second and third inequalities of the constraint become equivalent to \( RT_{i,j,p} \leq \hat{R}_i^p \leq t_{i,j} \), hence correctly enforcing the schedulability condition of Eq. (13). In all other cases where \( a_{i,j,p} = 0 \), the last two inequalities of the constraint become equivalent to \( -\infty \leq \hat{R}_i^p \leq \infty \), and hence have no effect.

**C. Design parameters and possible limitations of the approach**

A more accurate selection of the number and position of the synchronization points requires solving a design problem with trade-offs. For instance, with equally-spaced synchronization points, the higher the number of synchronization points, the more freedom in parallelizing the code; however, the corresponding LET intervals become shorter and the deadlines become tighter, hence possibly penalizing the system schedulability. From a computational point of view, the number of synchronization points also influences the complexity of the optimization problem by increasing the number of variables involved, thus directly affecting its runtime.

Furthermore, the communication overhead due to the copy process of LET labels may become significant. In case of systems with a high communication load (e.g., image-based recognition using camera sensors), if the number of synchronization points is reduced to limit the overhead, then the solver may be unable to find a feasible solution. In these cases, mixed approaches might be used to parallelize (some of) the memory transfers, e.g., using DMA controllers [30], [31].

Overall, the selection of the size and number of LET interval is similar to other common design decisions faced in automotive software, such as the granularity of runnables or the definition of the system tick size. We aim at exploring such aspects in future work.

**VII. CASE STUDY**

In this section, the MILP formulation is evaluated by applying it to a model of a realistic (and large-scale) engine control application, provided in the WATERS 2017 challenge [23].

### A. System model of the WATERS challenge

The target task set provided in [23] is composed of 10 periodic tasks and 11 interrupt service requests (ISRs), with given priorities and periods (or minimum inter-arrival times, respectively), mapped on a multicore platform with 4 identical cores. Globally, the application is composed of a total number of 1250 runnables. The model specifies the WCET of runnables, the set of labels accessed (both read and write) by each runnable, and the number of those accesses. The system includes 10000 labels, half of which are shared. Globally, the system requires 2235 inter-task messages and 2947 intra-task messages. Table I summarizes the main parameters of the periodic tasks of the application.

In the original allocation provided with the application, tasks are mapped to the cores as a whole (no splitting). To match the requirements of our analysis, when using this initial setting, all ISRs are considered as mapped in core \( P_1 \).

### B. Testing the MILP formulation

The MILP-based formulation presented in Sec. VI has been coded in C++ using the IBM CPLEX API, and was tested on the target task set of the WATERS challenge. Due to the high number of runnables and labels involved in the system, the optimization problem is particularly complex. One of the parameters that drastically impacts the complexity of the problem is the number of synchronization points \( N_i^p \). Considering as an example a selection of values for \( N_i^p \) as the one presented in Table II, the resulting MILP formulation

---

**TABLE I**

<table>
<thead>
<tr>
<th>Task</th>
<th>( T_i (\mu s) )</th>
<th># Runnab.</th>
<th># Accessed ( t )</th>
<th>WCET</th>
<th>Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_1 )</td>
<td>1000</td>
<td>42</td>
<td>293</td>
<td>764 ( \mu s )</td>
<td># 2</td>
</tr>
<tr>
<td>( T_2 )</td>
<td>6660</td>
<td>147</td>
<td>2055</td>
<td>3805 ( \mu s )</td>
<td># 2</td>
</tr>
<tr>
<td>( T_3 )</td>
<td>2000</td>
<td>137</td>
<td>1046</td>
<td>404 ( \mu s )</td>
<td># 3</td>
</tr>
<tr>
<td>( T_4 )</td>
<td>5000</td>
<td>23</td>
<td>122</td>
<td>931 ( \mu s )</td>
<td># 3</td>
</tr>
<tr>
<td>( T_5 )</td>
<td>10000</td>
<td>304</td>
<td>4686</td>
<td>11712 ( \mu s )</td>
<td># 4</td>
</tr>
<tr>
<td>( T_6 )</td>
<td>10000</td>
<td>307</td>
<td>2894</td>
<td>10468 ( \mu s )</td>
<td># 3</td>
</tr>
<tr>
<td>( T_7 )</td>
<td>50000</td>
<td>46</td>
<td>571</td>
<td>3084 ( \mu s )</td>
<td># 3</td>
</tr>
<tr>
<td>( T_8 )</td>
<td>( 2 \cdot 10^5 )</td>
<td>247</td>
<td>3001</td>
<td>9418 ( \mu s )</td>
<td># 3</td>
</tr>
<tr>
<td>( T_9 )</td>
<td>( 10^6 )</td>
<td>15</td>
<td>418</td>
<td>138 ( \mu s )</td>
<td># 3</td>
</tr>
<tr>
<td>( T_{10} )</td>
<td>( 10^6 )</td>
<td>44</td>
<td>631</td>
<td>137 ( \mu s )</td>
<td># 3</td>
</tr>
</tbody>
</table>
requires more than $10^5$ variables and nearly $2 \cdot 10^5$ constraints. All tests have been performed on a machine with 128GB of memory, 2x Intel Xeon(R) CPU E5-2640 v4 @ 2.40GHz, with 40 cores. The MILP solver is automatically executed in parallel by CPLEX (which often uses all the available cores). In our tests, using the selection of $N_i^\Gamma$ of Table II, the first solution satisfying all the constraints is found on average in approximately 10 minutes from the start of the execution. After that, the solver continues exploring the search tree, looking for better solutions. A timeout of 24 hours was set, after which the mapping related to the best R/D ratio is provided as solution.

The mapping produced by the MILP optimization is compared with a plain genetic algorithm that implements the analysis of Sec. V as fitness function. Each chromosome represents a possible mapping of runnables and labels, and is randomly initialized such that it satisfies all the given mapping rules. All new generations created through crossover and mutation functions must also respect all the mapping rules. A starting population of 200 chromosomes is used. Additionally, the results of the MILP optimization are also compared against the original task set mapping of Table I (provided with the challenge), by adding a LET communication task to each core and using the LET design of [25].

The model provided in the WATERS challenge is overloaded, i.e., the worst-case response times do not satisfy all the deadline constraints with the original mapping. For this reason, we will use a scaling factor $\gamma$ applied to the WCET of all the runnables. Here, we chose $\gamma$ in the set \{0.65, 0.7, 0.75, 0.8\}. Figure 5 shows a comparison of the results, obtained with a selection of values for $N_i^\Gamma$ for the 10 periodic tasks, as \{$N_i^\Gamma$\} = \{2, 3, 2, 2, 2, 2, 4, 4, 4, 4, 4\}. This configuration has been empirically selected such that a schedulable solution is guaranteed for all the tested scaling factors $\gamma$. The maximum R/D obtained by the MILP optimization is always lower than the one obtained using the original formulation, thus guaranteeing schedulability for higher values of $\gamma$. The results are also always better than the ones of the genetic algorithm.

In order to test the accuracy of our optimizer, we used the response-time analysis of Section V, without the approximations presented in Section VI-A, to check the mapping obtained with the MILP formulation and compute the resulting (refined) R/D value. The results are presented in Figure 6. The small difference between the two values, computed for each $\gamma$, denotes that our approximate linear formulation used for the MILP has an accuracy extremely close to the original one.

Finally, some detail of the mapping produced by the MILP-based optimization with $\gamma = 0.75$ is presented as a representative example in Table II. The table shows the values of maximum R/D for each container task $T_i^\Gamma$ among the cores. Interestingly, the selected mapping requires that some container tasks are void (marked with an “X” in the table), especially for the tasks with a small number of runnables in the original deployment.

**TABLE II**

<table>
<thead>
<tr>
<th>Task</th>
<th>$T_i$</th>
<th>$N_i^\Gamma$</th>
<th>Core #2</th>
<th>Core #3</th>
<th>Core #4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma_1$</td>
<td>1000</td>
<td>2</td>
<td>0.167</td>
<td>0.477</td>
<td>0.517</td>
</tr>
<tr>
<td>$\Gamma_2$</td>
<td>6660</td>
<td>3</td>
<td>0.040</td>
<td>0.109</td>
<td>0.735</td>
</tr>
<tr>
<td>$\Gamma_3$</td>
<td>2000</td>
<td>2</td>
<td>0.115</td>
<td>0.368</td>
<td>X</td>
</tr>
<tr>
<td>$\Gamma_4$</td>
<td>5000</td>
<td>2</td>
<td>X</td>
<td>0.341</td>
<td>0.661</td>
</tr>
<tr>
<td>$\Gamma_5$</td>
<td>10000</td>
<td>2</td>
<td>0.831</td>
<td>X</td>
<td>0.794</td>
</tr>
<tr>
<td>$\Gamma_6$</td>
<td>20000</td>
<td>2</td>
<td>0.427</td>
<td>0.749</td>
<td>0.822</td>
</tr>
<tr>
<td>$\Gamma_7$</td>
<td>50000</td>
<td>2</td>
<td>X</td>
<td>0.389</td>
<td>0.338</td>
</tr>
<tr>
<td>$\Gamma_8$</td>
<td>$10^3$</td>
<td>4</td>
<td>0.376</td>
<td>0.786</td>
<td>X</td>
</tr>
<tr>
<td>$\Gamma_9$</td>
<td>$2 \cdot 10^3$</td>
<td>4</td>
<td>X</td>
<td>0.393</td>
<td>X</td>
</tr>
<tr>
<td>$\Gamma_{10}$</td>
<td>$10^6$</td>
<td>4</td>
<td>X</td>
<td>0.078</td>
<td>X</td>
</tr>
</tbody>
</table>

**Maximum R/D values using the mapping obtained with the MILP formulation ($\gamma = 0.75$). “X” identifies empty container tasks.

**VIII. Conclusion**

This paper presented a functional partitioning of a real-time application, described accordingly to the AUTOSAR standard, to a multicore platform. The proposed design leverages the LET paradigm and multiple synchronization points to enforce causality and determinism in the final system. This design is able to handle also constraints coming from causal relations between runnables that communicate over shared labels. A matching response-time analysis was presented, considering both interference due to execution of tasks in the same core, and interference due to LET communication coming from parallel executions of all the cores. A MILP formulation of the proposed design was also presented, adapting the response-time analysis with few conservative approximations. The resulting MILP-based optimizer was finally applied to a realistic case study of industry size showing very good performance.
REFERENCES


