Data consistency and temporal validity under the circular buffer communication paradigm

Evariste Ntaryamira
INRIA Paris
Paris, France
evariste.ntaryamira@inria.fr

Cristian Maxim
IRT-SystemX
Paris, France
cristian.maxim@irt-systemx.fr

Liliana Cucu-Grosjean
INRIA Paris
Paris, France
liliana.cucu@inria.fr

ABSTRACT

Technologies within embedded real-time systems are continuously evolving making such systems intelligent. This evolution has increased the interest in the data utilization while real-time constraints are considered. In this paper, we consider real-time constraints for programs communicating using a circular buffer communication paradigm. We propose a first result optimizing the buffer size. Our second contribution consists in providing an analytical characterization of the temporal validity and reachability properties of the data propagating along a functional chain. Last but not least, we propose a scheduling policy ensuring the consistency and the temporal validity of the used data.

CCS CONCEPTS

• **Computer systems organization → Real-time system specification**; **Embedded software**; • **Information systems → Process control systems**;

KEYWORDS

Real-time systems , Data management , Circular buffer inter-task communication model , Data consistency and data temporal validity

ACM Reference Format:
Evariste Ntaryamira, Cristian Maxim, and Liliana Cucu-Grosjean. 2019. Data consistency and temporal validity under the circular buffer communication paradigm. In *Proceedings of International Conference on Research in Adaptive and Convergent Systems, Chongqing, China, September 24–27, 2019 (RACS '19),* 6 pages.
https://doi.org/10.1145/3338840.3355682

1 INTRODUCTION AND RELATED WORK

The software embedded in a real-time system is composed of a large number of applications communicating through shared variables. Technologies within these systems are continuously evolving making them intelligent in the sense that at some point they can achieve targeted functions autonomously. For instance, the autonomous vehicles are extended with capability of sensing the surrounding environment and navigating on their own by making driving decisions. Hence, the correctness of a decision highly depends on the quality of the used input data. Thus, the data management within such systems must fulfill some properties in order to guarantee a their correct functioning.

In this paper, we consider that fulfilling the *temporal validity* and the *consistency* of the input data is mandatory for a correct functioning of the system. Given that a sensed input data provides the current status of the corresponding entity in the system environment [13], an input data is said *temporally valid* if, at a time instant t following its production date, no other input related to that entity is yet produced.

The data *consistency* property consists in protecting the data from shared resources against any corruption or state change when this data is still under use by some applications. This property is verified at the implementation stage of the system development life cycle. Assuming that the input data structure may be complex (i.e data table, frames), it is obvious that, if an application is preempted before it has finished to *read* the input data, at the next activation, it may resume the reading using the updated data value. For the convenience purpose, in the reminder of the paper we use the word "task" to represent a program or application.

In order to illustrate a situation of a data consistency violation, we consider the example in the Figure 1. Herein, two periodic tasks $\tau_1(1,4)$ and $\tau_2(4,8)$ where a task is defined by a pair (C_i, T_i). We understand by C_i the worst-case execution time and by T_i the period of the task. In this paper, we consider implicit deadline tasks, i.e., the period is equal to the deadline. Two tasks τ_1 and τ_2 communicate through a shared register *register_1* where τ_1 writes input data to be used by τ_2 instances.

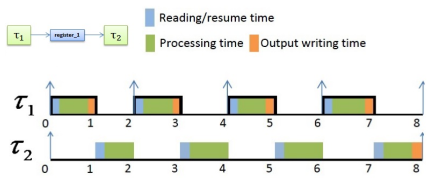


Figure 1: Consistency violation case

According to the register-based inter-task communication paradigm, a register of size one is shared with communicating tasks.

Each time a new data is produced, the new data overwrites the old one. So, reading task always accesses the most recent value. From this point of view, the scheduling result presented on the Figure 1 shows that at the time instant $t = 1$, the first instance of τ_1 produces an output data which is immediately consumed by the first instance of τ_2 at $t = 1$. At $t = 2$, the second instance of τ_1 is released and it preempts the first instance of τ_2 due to the high priority of τ_1. At $t = 3$ a new data is written into $register_1$. At this moment $t = 3$, the first instance of τ_2 resumes its execution using the second data given that the first one is overwritten at $t = 3$. This situation repeats at $t = 5$ and $t = 7$, where the first instance of τ_2 resumed its execution using each time a new value. However, this combination of old and new data might lead to erroneous results or system performance degradation.

In order to cope with similar violation situations, solutions ensuring the data consistency have been proposed during the last decades. Most of the existing solutions require the implementation of various arbitration mechanisms such the use of semaphores and synchronization protocols which may lead to an unpredictable behavior of the system [2, 12]. To a certain extent, the use of the arbitration mechanisms may provoke the priority inversion problems and possible deadlock formations [9–11]. Solutions based on the utilization of variables local copies exist but induce the uncertainty in the data management for the reason that these copies are allocated and destroyed dynamically. We cite, for instance, the *implicit communication* considered in [1, 7]. In order to increase the data management predictability while avoiding the drawbacks induced by the above mechanisms, we propose to calibrate the buffer such that the *data consistency* is ensured without implementing any arbitration mechanism, nor making local copies.

Contribution In this paper we consider the inter-task communication model based the circular buffer, which eases the data consistency between tasks. Formal method calculating the optimal size for each of the buffers is given. Afterwards, we provide an analytical characterization of the temporal validity and reachability properties of the data flowing in between communicating tasks. Finally, a scheduling policy ensuring the data consistency and temporal validity is proposed.

Paper structure. We present the context of our work in Section 1 as well as existing results on the data consistency. In Section 2 we present the system model and the associated notations. In Section 3.1 we introduce our first contribution; an algorithm calculating the optimal size of different buffers while in Section 3.2 an analytical characterization of the temporal validity and reachability properties of the data propagating along the chain is given. In Section 3.3 we present an algorithm ensuring the data temporal validity under buffer fixed sizes. Numerical results related to these algorithms are presented in Section 4. We conclude our paper in Section 5, where we provide also hints for future work.

2 MODELS AND ANNOTATIONS

In this section, we introduce the system and the communication models as well as the notion of functional chains.

2.1 System Model

We consider a system τ of n periodic tasks $\{\tau_1, \tau_2, \cdots, \tau_n\}$ scheduled preemptively on one processor according to fixed-priority scheduling algorithm. Each task τ_i is described by the tuple (C_i, D_i, T_i), where C_i is the worst-case execution time, D_i the deadline, and T_i is the minimum inter-arrival time (release period) of the task τ_i. We assume that all tasks are released simultaneously and they have implicit deadlines; that is, $\forall \tau_i \in \tau, T_i = D_i$. Without any loss of generality, we consider that the tasks are ordered from the highest to the lowest priority. Hence, if $i < j$, then τ_i has a higher priority than τ_j. Each task τ_i generates an infinite number of successive jobs $\tau_{i,j} | j = 1, \cdots, \infty$. However, for simplicity, in the reminder of this paper, we do not focus on specific jobs. Therefore, τ_i has the meaning of any instance $\tau_{i,j}$. We define the hyper-period as the least common multiple of the periods of all tasks and we denote it by $H = lcm\,\{T_i\}\,|i = 1, \cdots, n$. Given that we consider synchronous released implicit deadlines tasks, then the interval $(0, H]$ is a feasibility interval for the task system [5]. By feasibility interval we understand the smallest time interval such that if all deadlines are met within this interval, then all deadlines are met. We consider that tasks share data through buffers and a task may belong to two different classes: producer or consumer. For instance, in Figure 2 the task τ_3 is both a producer and a consumer. Task τ_3 is reading from the buffer Cb_1 the data that the task τ_1 has produced. The buffer model is described in Section 2.2.

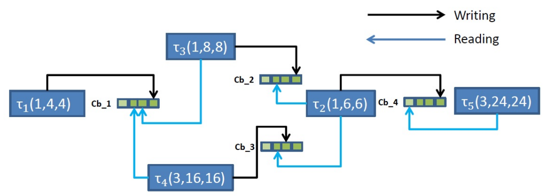


Figure 2: System tasks model

The data propagation order between tasks does not impose an execution order between those tasks. We describe the data dependencies between the tasks by a graph. We denote by $G = (V, E)$ such graph, where V is the set of tasks $\{\tau_1, \cdots, \tau_n\}$ and E is the set of edges where $(\tau_i, \tau_j) \in E$ if τ_j consumes data produced by τ_i. The graph G may have several components. A task that has no predecessor is a source task and such tasks are often corresponding to tasks dealing with data from sensors. A task that has no successor is a sink task and such tasks correspond to tasks dealing with data from actuators. We denote by *pred*(τ_i) the set of predecessors of a task τ_i and by *succ*(τ_i) the set of its successors. The former and the latter return, respectively, a list of predecessors and successors to τ_i if they exist. For a source task τ_i the *pred*(τ_i) has one element by default -1 as τ_i has no predecessors. For a sink task τ_j the set *succ*(τ_j) contains -1 as τ_j has no successors.

*Definition 2.1 (**Functional chain**).* A *functional chain* is the data propagation path composed of tasks starting from a source task and ending by a sink task. We denote such propagation path by $\{\tau_i, \cdots \tau_j\}$, where τ_i is a source task and τ_j a sink task.

A functional chain determines the data propagation order while the scheduling algorithm defines the tasks execution order based on their priority. Given that several functions may co-exist within the same task system, a *functional chain* corresponds to one function. A functional chain may include three possible relations: linear, join and fork that we define as follows.

*Definition 2.2 (**Fork relation**).* A task τ_i has a *fork* relation with $\tau_{i_1}, \cdots, \tau_{i_k}$ if $(\tau_i, \tau_{i_j}) \in E$, where $j \in \{1, \cdots, k\}$ with k the number of successors for the task τ_i.

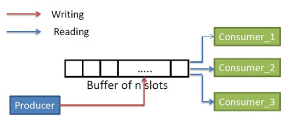


Figure 3: A fork relation

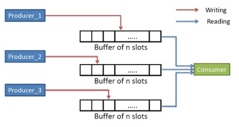


Figure 4: A join relation

A fork relation implies that a single producer is writing data for several consumers. An example of this relation is $(\tau_1, \tau_3), (\tau_1, \tau_4) \in E$ thus τ_1 has 2 successors, namely, τ_3 and τ_4. This means that τ_1 produces input data for both tasks τ_3 and τ_4.

*Definition 2.3 (**Join relation**).* A task τ_j has a *join* relation with $\tau_{j_1}, \cdots, \tau_{j_k}$ if $(\tau_{j_i}, \tau_j) \in E$, where $i \in \{1, \cdots, k\}$ with k the number of predecessors for the task τ_j.

A join relation refers to a case where a single consumer task has many inputs coming from different sources or sub-systems. An example of this relation is $(\tau_3, \tau_2), (\tau_4, \tau_2)$, where both τ_3 and τ_4 produce input data required by τ_2. A particular case of join relation is when the number of predecessors $k = 1$ and we name this case as the ***linear relation***. An example of a linear relation is $(\tau_2, \tau_5) \in E$, where τ_5 has only one predecessor the task τ_5. Moreover, a functional chain may be composed of tasks producing /consuming data at different rates (related to their periods), which may provoke over- or under sampling situations. Inevitably, some data samples will never be consumed while others are used several times.

2.2 Communication model

The communication model used in this paper is the circular buffer.

2.2.1 ***Definition and organization.*** A circular buffer is a *FIFO data structure* that considers memory to be managed circularly; that is, the *read/write* indices loop back to 0 after it reaches the buffer length [3]. It has a fixed size allocated once at the system run-time. It uses the *tail* and *head* pointers to indicate where to read or write, respectively, input and output data.

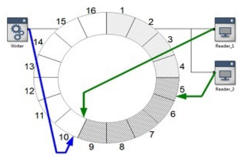


Figure 5: The circular buffer [6]

Accordingly, each time a new data sample is added (write) into the buffer, the *head* pointer is incremented and likewise, when the data is being read the *tail* pointer is incremented. At the beginning *tail* and *head* are both at index 0. The modulo operation is performed to reset head or tail to zero, every time the maximum index is reached. The utilization of such buffer offers many benefits, such as:

- A high degree of the communication predictability: Since the address and the size of the buffer structure never change, we always know for producer/consumer tasks where to write/read by the help of *tail* and *head* pointers.
- No dynamic memory allocation: Making local copies during the execution is a resource consuming operation which increases the degree of uncertainty regarding the data management in real-time. Finally, the circular buffer is easy to implement.

2.2.2 ***Data management within the circular buffer.*** Within a functional chain, each buffer is dedicated to store data samples related to a unique data variable (label) and may be shared between multiple tasks. Any number of consumer tasks can read from the shared buffer simultaneously, but only one producer can write to the shared buffer. Even though the buffer may consist of several slots, when a producer is writing data into a given buffer slot, no other tasks can access this slot. Otherwise, the producer may put the data being read into the inconsistent state.

From what precedes, we consider a set of l buffers $\{\beta_1, \cdots, \beta_l\}$, where each buffer β_l is characterized by its cardinality $|\beta_l|$; that is, its size. A consumer task reads data from the buffers where its consecutive predecessors (producers) write their output data.

3 DATA CONSISTENCY MAINTENANCE.

This section provides the formal methods used along the paper to ensure the data consistency property based on the communication model proposed in section 2.2. We recall that in this paper the shared buffers are accessed asynchronously in a non-blocking fashion, where a same buffer can be used by a unique producer to write new data, and multiple consumers may use it for reading their inputs data. The buffer content can be modified only by the producer and the buffer size depends on the sampling rate of the producer task.

3.1 Computing the buffer optimal size

In this section, we propose formal methods to compute the minimum size of the buffer guaranteeing against data consistency violation. In order ensure the data consistency, it is required that if a buffer slot is accessed for reading the input data, this slot will never be used by the producer task to write new data before the execution completion of the instances of all consumers that are currently reading from this slot.

Hence, $\forall\ (\tau_i, \tau_{i_j}) \in E$ where $j \in \{1, \cdots, k\}$ with k the number of successors for the task τ_i, with β_i a buffer that τ_i shares with all τ_{i_j} and whose size is $|\beta_i|$, we aim to compute the *optimal size* of $|\beta_i|$ guarantying against any data consistency violation without any arbitration mechanism or buffer copy making.

*Definition 3.1 (**Optimal size**).* $|\beta_i|$ is optimal if it is guaranteed that $\forall(\tau_i, \tau_{i_j}) \in E | j = 1, \cdots, k$ an input data value read from a given slot of β_i by τ_{i_j} instances, will be accessed for writing new data value (overwriting) only if all k instances that had accessed this slot for reading have completed their executions.

Theorem 3.2. *We consider $(\tau_{j_i}, \tau_j) \in E | i = 1, \cdots, k$ with k the number of tasks that produce input data for τ_j, where τ_j has a **join relation** with τ_{j_i} if $k > 1$ or a **linear relation** if $k = 1$. Given that the τ_j instances consume input data from buffers populated each by a single producer, the optimal size of β_{j_i}, is equal to the number of τ_{j_i} instances that can be released and complete their executions before an instance of τ_j completes its largest execution. Hence,*

$$|\beta_{j_i}| = \begin{cases} 1, & \text{if } T_{j_i} >= T_j. \\ \left\lceil \frac{R_j}{T_{j_i}} \right\rceil, & \text{if } T_{j_i} < T_j. \end{cases} \tag{1}$$

where $|\beta_{j_i}|$ is the cardinality of β_{j_i}, T_{j_i} is the period of τ_{j_i}, and T_j and R_j are, respectively, the period and the worst case response time of τ_j.

Proof: Consider $(\tau_{j_i}, \tau_j) \in E$ where i is one of the k tasks that produce input data for τ_j,

- if $T_{j_i} = T_j$, it means that τ_{j_i} and τ_j instances are always released at a same time instant. Hence, a buffer of size one is sufficient.
- if $T_{j_i} > T_j$, it implies that a data sample produced by an instance of τ_{j_i} can be read by several instances of τ_j. Additionally, no instance of τ_{j_i} can execute before an executing instance of τ_j completes. Therefore, a buffer of size one is also sufficient.
- Finally, if $T_{j_i} < T_j$, they may be new productions of τ_{j_i} between the activation and the completion of an instance of τ_j. The largest time an instance of τ_j can execute is equal to its worst case response time denoted R_j. Within R_j time units $\left\lceil \frac{R_j}{T_{j_i}} \right\rceil$ instances of τ_{j_i} can be released and complete their executions. Therefore, in order to guarantee that the buffer slot accessed for reading will never be overwritten before the completion of the slowest τ_{j_i} instance, a buffer of $\left\lceil \frac{R_j}{T_{j_i}} \right\rceil$ size is sufficient.

Theorem 3.3. *We consider a* ***fork relation*** *such that $(\tau_i, \tau_{i_j}) \in E | j = 1, \cdots, k$ where k is the number of tasks whose instances consume data samples produced by τ_i instances. The optimal size of β_i is equal to the number of τ_i instances that can be released and complete their executions before the completion of an instance of the slowest among all τ_{i_j}. Hence,*

$$|\beta_i| = \left\lceil \frac{max\left\{R_{i_j}\right\}}{T_i} \right\rceil. \tag{2}$$

where $|\beta_i|$ is the cardinality of β_i; the buffer where τ_i instances write the data needed by τ_{i_j} instances, T_i is the period of τ_i and R_{i_j} the worst case response time of τ_{i_j}.

Proof: Within a fork relation, a single task is producing data for several consumer tasks which may have different sampling periods .So, $\forall (\tau_i, \tau_{i_j}) \in E | j = 1, \cdots, k$ where k is the number of tasks whose instances consume data produced by τ_{i_j} instances. $|\beta_i|$ is computed by focusing on the τ_{i_j} task whose instances read the input data slowly. In other words, they are the one having the largest worst case response time among all τ_{i_j}. Intuitively, if the accessed slot cannot be overwritten before the completion of the instance of the slowest among all τ_{i_j} tasks, then this can be the case for all τ_{i_j}. Hence, the Equation 2 is correct.

3.2 Analytical characterization of the temporal validity and reachability properties

In this section we introduce the temporal validity and reachability properties of the propagating data under buffer size constraint. We also provide an analytical characterization of each of them. These two properties are characterized by considering both the tasks execution and the data propagation orders.

In this paper we assume, a task instance reads all its inputs data at its activation time and writes back the output data at the completion time where this data becomes immediately available for consumption. Given that they may be several data samples available in the buffer, we say that a data sample is *fresh* or *temporal valid* if, since the time instant it is produced, its producer has not completed another execution.

We denote p the number of data samples already written by a producer task at a time instant t. Therefore, for a system of n periodic tasks $\{\tau_1, \cdots, \tau_n\}$ and a pair of tasks $(\tau_i, \tau_{i_j}) \in E$, the writing point of the p^{th} data sample by τ_i, denoted $w_{i,p}^{p}$, corresponds to the

Algorithm 1: Algorithm calculating the buffer's optimal size

```
1  Require: $\tau_i|_{i=1,\cdots,n}, \omega_j|_{j=1,\cdots,m}, \beta_k|k = 1,\cdots,l$
2  Initialize: $pred(\tau_i) \longleftarrow -1$ $succ(\tau_i) \longleftarrow -1$
3  for each $\omega_j$ do
4      for each $\tau_i \in \omega_j$ do
5          return $pred(\tau_i)$, $succ(\tau_i)$
6          if sizeof($succ(\tau_i)$)$\geq 1 \wedge succ(\tau_i) \neq -1$ then
7              return $\beta_k$
8              if $sizeof(succ(\tau_i))$=1 then
9                  if $T_i \geq T_{succ(\tau_i)}$ then
10                     $|\beta_k|= 1$
11                 else
12                     $|\beta_k|= \left\lceil \frac{R_{succ(\tau_i)}}{T_i} \right\rceil$
13             else
14                 for each $\tau \in succ(\tau_i)$ do
15                     Return $\max\{R_{succ(\tau_i)}\}$
16                 $|\beta_k|= \left\lceil \frac{max\{R_{succ(\tau_i)}\}}{T_i} \right\rceil$
```

completion time of the p^{th} instance of τ_i, where p in the subscript and p in superscript refer respectively to the p^{th} instance of τ_i and the p^{th} data sample already produced by τ_i instances. Formally,

$$w_{i,p}^{p} = p * T_i + C_i + I_k \tag{3}$$

with

$$I_k = \sum_{k \in hp(i)}^{w_{i,p}^{p}}{}_{p*T_i} \left\lceil \frac{w_{i,p}^{p}}{T_k} \right\rceil * C_k \tag{4}$$

where $hp(i)$ is the set of higher priority tasks than τ_i that were executed within a time interval bounded by the release time of the p^{th} instance of τ_i which is given by $p * T_i$ and $w_{i,p}^{p}$ its completion time. I_k is the interference induced by k higher priority tasks than τ_i and is computed recursively.

The same way, the writing time of the data sample by the next instance of τ_i; that is $p+1$ is denoted $w_{i,p+1}^{p+1}$ and is analogically calculated using the Equation 3 substituting p by $p+1$.

*Definition 3.4 (**Data temporal validity**).* We consider a task $\tau_i \in \tau$ such that τ_i is not an actuator task and p the p^{th} data sample produced by τ_i. At a time instant t, if we have

$$w_{i,p}^{p} \leq t < w_{i,p+1}^{p+1} \tag{5}$$

then the p is *temporal valid.*

On the other hand, for each $(\tau_i, \tau_{i_j}) \in E$, the *reachability property* determines if the p^{th} output data produced by an instance of τ_i at $w_{i,p}^{p}$ time instant will be consumed by at least one of the τ_{i_j} instances before being overwritten.

Hence, $\forall (\tau_i, \tau_{i_j}) \in E$, where the data produced by τ_i instances are meant to be consumed by τ_{i_j} instances. Let $r_{j,q}^{p}$ be the reading time of the p^{th} data sample by the q^{th} instance of τ_{i_j}. From the tasks execution order point of view, the propagation of the p^{th} data sample up to τ_{i_j} is delayed by all tasks of higher priority than τ_{i_j} released in the time interval bounded by $w^p_{i,p}$ and $r^p_{j,q}$. The reachability property is always verified for all the cases where $T_i \geq T_{i_j}$ since each data produced by τ_i will be used by at least one instance of τ_{i_j}. However, if $T_i < T_{i_j}$, some data will be overwritten before being used and an instance of τ_{i_j} will consume utmost one of the produced data. The Equation 6 allows us to know for each data sample p the instance of the consumer task that may read it.

$$q = \begin{cases} \left\lceil \frac{p*T_i}{T_{i_j}} \right\rceil, & \text{if } T_{i_j} \mod T_i = 0 \\ \left\lfloor \frac{(p+1)*T_i}{T_{i_j}} \right\rfloor + 1, & Otherwise \end{cases} \tag{6}$$

Lemma 3.5. *We consider a system of n periodic tasks $\{\tau_1, \cdots, \tau_n\}$ and a pair of tasks (τ_i, τ_{i_j}) such that $(\tau_i, \tau_{i_j}) \in E$. If we have*

$$q * T_{i_j} - (p+1) * T_i \leq T_i \tag{7}$$

then p is reachable.

Proof: A data sample is reachable if it is consumed before being overwritten. So, when the producer writes data quicker than the reader consumes them, it is obvious that there may be several data written between two consecutive activations of the consumer which may still be available depending on the buffer size. So, if the p^{th} data sample is produced at the completion of the instance released at $p * T_i$ time instant and $(p+1, \cdots, p+k)$ instances are also released before the activation of the q^{th} instance of τ_{i_j}, then the $(p, p+1, \cdots, p+k)^{th}$ data samples may have been overwritten or are still queued waiting to be probably read by this q^{th} instance of τ_{i_j}. Unfortunately, only the data sample which is *fresh* or *temporally valid* will be read; that is, the one that has last for no longer than T_i time units at the activation the q^{th} instance of τ_{i_j}. In this context, only $(p+k)^{th}$ data sample is consumed by q^{th} instance of τ_{i_j}.

3.3 Data temporal validity

Given that we use buffers whose size may be larger than one, it is obvious that the consumer task will not implicitly know which data is *temporally valid*. In order to use the data that reflects the current status of the system environment (valid data), we introduce a novel parameter; the ***sub-sampling rate***, denoted σ.

*Definition 3.6 (**Sub-sampling rate**).* We consider a pair of tasks $(\tau_i, \tau_{i_j}) \in E$ and a buffer β_i where τ_i instances write data to be consumed by τ_{i_j} instances. The sub-sampling rate $\sigma_{\tau_{i_j},\beta_i}$ is the number of data samples that τ_i instances have written into β_i within a time delay bounded by the previous and the current activation times of τ_{i_j}.

This parameter was first utilized in [4, 8] where authors consider the under-sampling pattern; that is, the data are produced quicker than they are consumed. In this paper, we consider all possible sampling patterns (under-, over- and same sampling patterns). The value of σ is computed online based on the following principle:

$\forall(\tau_i, \tau_{i_j}) \in E$ and a buffer β_i where τ_i instances write data to be consumed by τ_{i_j} instances, each time an instance of τ_i completes its execution it inserts a new data sample into β_i. In this case $\sigma_{\tau_{i_j},\beta_i}$ is incremented by 1. Subsequently, when an instance of τ_{i_j} is activated, it reads from β_i the data being at the position given by the preview value of $tail_{\tau_{i_j},\beta_i}$ augmented with the current value of $\sigma_{\tau_{i_j},\beta_i}$ and afterwards, $\sigma_{\tau_{i_j},\beta_i}$ is initialized to 0.

Algorithm 2: Algorithm to ensure the data temporal validity

```
1  Require: $\tau_i$, $\beta_k|\{i = 1, \cdots, n\} \wedge \{k = 1, \cdots, l\}$
   Output: $\sigma_{pr,\beta}, tail_{pr,\beta}, head_{pr,\beta}$
2  Initialize: $t = 0, \sigma_{\tau_i,\beta} = 0, q_{\tau_i} \longleftarrow 0$
3  while $t < lcm$ do
4      return $\tau_{pr}$
5      if $q_{pr} = 0$ then
6          if $pred(\tau_{pr}) \neq -1$ then
7              for each $\beta_k$ do
8                  if $(tail_{pr,\beta} = |\beta_k|-1)$ then
9                      $tail_{pr,\beta} \longleftarrow \sigma_{pr,\beta_k}$
10                 else
11                     $tail_{pr,\beta} \longleftarrow ((tail_{pr,\beta} + 1) + \sigma_{pr,\beta_k})$
                         mod $|\beta_k|$
12             $\sigma_{pr,\beta_k} \longleftarrow 0$
13     $q_{pr} \longleftarrow q_{pr} + 1$
14     if $q_{pr} = C_{pr}$ then
15         if $succ(\tau_{pr}) \neq -1$ then
16             for each $\beta_k$ do
17                 if $head_{pr,\beta_k} = |\beta_k|-1$ then
18                     $head_{pr,\beta_k} \longleftarrow 0$
19                 else
20                     $head_{pr,\beta_k} \longleftarrow head_{pr,\beta_k} + 1$
21                 $\sigma_{pr,\beta_k} \longleftarrow \sigma_{pr,\beta_k} + 1$
22         $q_{pr} \longleftarrow 0$
23     $t \longleftarrow t + 1$
```

By doing so, we guarantee that the reader task instances will always retrieve valid data from the buffer. In the Algorithm 2, τ_{pr}, q_{pr} and $head_{pr,\beta}$ stand, respectively, for the higher priority task, the budget already consumed by τ_{pr} and the pointer to the slot of β where τ_{pr} is going to write the next data sample.

The Algorithm 2 works as follows: Since the input data are read at the task instance activation time and the output result are written back at the completion time, at the line 5 and 14, this algorithms checks these two states. If the consumed budget is equal to zero and the prior task is not a sensor task then it stats reading input data from all connected buffers. These inputs are read at the positions given by $tail_{pr,\beta_k}$. Similarly, at $q_{pr} = C_{pr}$; that is, at the completion time. In this case, if the prior task is not an actuator task then its running instance writes the data in the corresponding buffer at the positions given by $head_{pr,\beta_k}$.

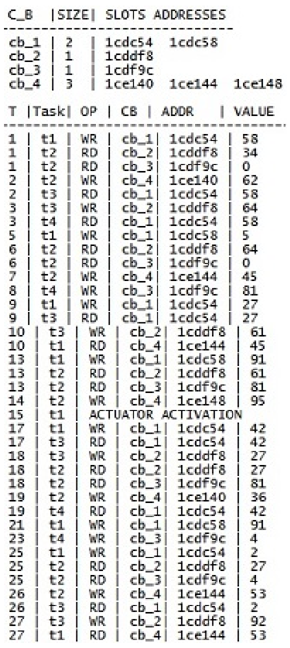
4 EVALUATION AND NUMERICAL RESULTS

In this section we present our experimental results regarding algorithms proposed within Section 3. The evaluation is performed considering the task system presented on the Figure 2 and may be implemented to for more complex systems. The Table 1 contains evaluation results of the Algorithm 1 where the size of *cb*_1, *cb*_2, *cb*_3 and *cb*_4 are, respectively, 2, 1, 1 and 3 buffer slots.

	C_i	D_i	T_i	R_i	β name	β size
τ_1	1	4	4	1	cb_1	$\lvert cb\_1\rvert = \left\lceil \frac{max\{R_3, R_4\}}{T_1} \right\rceil = \left\lceil \frac{8}{4} \right\rceil = 2$
τ_2	1	6	6	2	cb_4	$\lvert cb\_2\rvert = \left\lceil \frac{R_5}{T_2} \right\rceil = \left\lceil \frac{15}{6} \right\rceil = 3$
τ_3	1	8	8	3	cb_2	$\lvert cb\_3\rvert = \left\lceil \frac{R_2}{T_3} \right\rceil = \left\lceil \frac{2}{8} \right\rceil = 1$
τ_4	3	16	16	8	cb_3	$\lvert cb\_4\rvert = \left\lceil \frac{R_2}{T_4} \right\rceil = \left\lceil \frac{2}{16} \right\rceil = 1$
τ_5	3	24	24	15		

Table 1: Evaluation of the Algorithm 1

Figure 6 presents the scheduling results taking into account the sizes of the different buffers. In order to prove the correctness of the Algorithm 2, we need to verify the following statements:



```
C_B  |SIZE| SLOTS ADDRESSES
-----------------------------------
cb_1 | 2  | 1cdc54  1cdc58
cb_2 | 1  | 1cddf8
cb_3 | 1  | 1cdf9c
cb_4 | 3  | 1ce140  1ce144  1ce148

T |Task| OP | CB | ADDR   | VALUE
-------------------------------------
1 | t1 | WR | cb_1| 1cdc54 | 58
1 | t2 | RD | cb_2| 1cddf8 | 34
1 | t2 | RD | cb_3| 1cdf9c | 0
2 | t2 | WR | cb_4| 1ce140 | 62
2 | t3 | RD | cb_1| 1cdc54 | 58
3 | t3 | WR | cb_2| 1cddf8 | 64
3 | t4 | RD | cb_1| 1cdc54 | 58
5 | t1 | WR | cb_1| 1cdc58 | 5
6 | t2 | RD | cb_2| 1cddf8 | 64
6 | t2 | RD | cb_3| 1cdf9c | 0
7 | t2 | WR | cb_4| 1ce144 | 45
8 | t4 | WR | cb_3| 1cdf9c | 81
9 | t1 | WR | cb_1| 1cdc54 | 27
9 | t3 | RD | cb_1| 1cdc54 | 27
10  | t3 | WR | cb_2| 1cddf8 | 61
10  | t1 | RD | cb_4| 1ce144 | 45
13  | t1 | WR | cb_1| 1cdc58 | 91
13  | t2 | RD | cb_2| 1cddf8 | 61
13  | t2 | RD | cb_3| 1cdf9c | 81
14  | t2 | WR | cb_4| 1ce148 | 95
15  | t1 | ACTUATOR ACTIVATION
17  | t1 | WR | cb_1| 1cdc54 | 42
17  | t3 | RD | cb_1| 1cdc54 | 42
18  | t3 | WR | cb_2| 1cddf8 | 27
18  | t2 | RD | cb_2| 1cddf8 | 27
18  | t2 | RD | cb_3| 1cdf9c | 81
19  | t2 | WR | cb_4| 1ce140 | 36
19  | t4 | RD | cb_1| 1cdc54 | 42
21  | t1 | WR | cb_1| 1cdc58 | 91
23  | t4 | WR | cb_3| 1cdf9c | 4
25  | t1 | WR | cb_1| 1cdc54 | 2
25  | t2 | RD | cb_2| 1cddf8 | 27
25  | t2 | RD | cb_3| 1cdf9c | 4
26  | t2 | WR | cb_4| 1ce144 | 53
26  | t3 | RD | cb_1| 1cdc54 | 2
27  | t3 | WR | cb_2| 1cddf8 | 92
27  | t1 | RD | cb_4| 1ce144 | 53
```

Figure 6: Algorithm 2 evaluation

(1): Producer and consumer tasks must write/read data in/ from the right slot of the buffer thanks to tail and head pointers. We consider for instance the task τ_2 that writes in **cb**_4, which size is of 3 data samples. The composing slots occupy following addresses: **1ce140**, **1ce144** and **1ce148**. With the help of *head* pointer, at time instant $t = 2$, it wrote value 62 in **1ce140**, at $t = 7$ it wrote value 45 in **1ce144**, at $t = 14$ it wrote value 95 in **1ce148**, at $t = 19$ it replaced value 45 by 36 in **1ce140**, and so on and so forth. On the other hand, based on the *tail* position, it always read input from the appropriate buffers, namely **cb**_2 and cb_3. For instance, at the time instant $t = 1$ it read 34 from **1cddf8** (**cb**_2) and 0 from **1cdf9c** (**cb**_3). The above holds for all tasks within τ.

(2): The content of a slot being accessed for reading must not be accessed by the writer before the completion of all readers currently reading from it. We consider the buffer cb_1 which occupies **1cdc54** and **1cdc58**. This buffer is used by both task τ_3 and τ_4 for their respective input data. Let us focus on the task τ_4 as for instance. The first instance of τ_4 read input data at $t = 3$, where it read value 58 from **1cdc54**. This instance completed its execution at $t = 8$. We need to prove that **1cdc54** will not be accessed (by the instances of τ_1 activated after $t = 3$) before the completion time of that instance of τ_4. Indeed, the next writing time of an instance of τ_1 being between $t = 3$ and $t = 8$ happened at $t = 5$ and wrote value 5 in the next slot **1cdc58** and the second that overwrote **1cdc54** content happened at $t = 9$, which is right, since the instance of τ_4 that used data from **1cdc54** had finished the execution at $t = 8$.

(3): The consumer task instances always read the data which are temporally valid given the size of different buffers. This is easily observed on the Figure 6. For instance, we consider task τ_5 that read data produced by τ_2. The first instance of τ_5 activated at $t = 10$ consumed the last data produced by an instance of τ_4 that completed at $t = 7$ while the second activated at $t = 27$ consumed the last data produced by an instance of τ_2 that completed at $t = 26$. This is the case for all the tasks.

5 CONCLUSION AND FUTURE WORKS

In this paper we introduced communication model based on the circular buffer. In the Algorithm 1 we showed how to formally reduce the buffers size in such a manner that the data consistency is deterministically ensured.

Compared to the communication model proposed in [1, 7], the utilization of the circular buffer is more predictable and does not induce extra resource consumption since the buffer optimal size is allocated once at the system run-time.

An analytical characterization of the data reachability and temporal validity properties of is provided. Further, an algorithm ensuring the utilization of valid data, taking into account the buffers size, is given and proved.

In the future works, we aim to extend these results to multi-processor platform. Further, we intend to implement the proposed communication model on a real automotive use case.

REFERENCES

[1] AUTOSAR. 2016. Spec. of Timing Extensions. *AUTOSAR Std. 4.3* (2016).

[2] P. Buonocunto, A. Biondi, M. Pagani, M. Marinoni, and G. Buttazzo. 2016. ARTE: Arduino Real-time Extension for Programming Multitasking Applications. In *the 31st Annual ACM Symposium on Applied Computing (SAC).*

[3] EmbedJournal. [n. d.]. Implementing Circular Buffer in C. https://embedjournal.com/implementing-circular-buffer-embedded-c/. ([n. d.]).

[4] R. Gerber, S. Hong, and M. Saksena. 1994. Guaranteeing End-to-End Timing Constraints by Calibrating Intermediate Processes. In *the 15th IEEE Real-Time Systems Symposium (RTSS).* 192–203.

[5] J. Goossens, E. Grolleau, and L. Cucu-Grosjean. 2016. Periodicity of real-time schedules for dependent periodic tasks on identical multiprocessor platforms. *Real-Time Systems* 52, 6 (2016), 808–832.

[6] INTEMPORA. 2019. RTMaps application. https://intempora.com/products/rtmaps.html. (2019).

[7] L. Michel, T. Flaemig, D. Claraz, and R. Mader. 2016. Shared SW development in multi-core automotive context. In *the 8th European Congress on Embedded Real Time Software and Systems (ERTS).*

[8] E. Ntaryamira, C. Maxim, and L. Cucu-Grosjean. 2018. Ensuring data freshness for periodic real-time tasks. In *the 12th Junior Researcher Workshop on Real-Time Computing.*

[9] J. Schlatow and R. Ernst. 2016. Response-Time Analysis for Task Chains in Communicating Threads. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).* 245–254.

[10] L. Sha, R. Rajkumar, and J. P. Lehoczky. 1990. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. Computers* (1990).

[11] T.Kloda, A. Bertout, and Y. Sorel. 2018. Latency analysis for data chains of real-time periodic tasks. In *23rd IEEE International Conference on Emerging Technologies and Factory Automation, ETFA.* 360–367.

[12] J. Wu. 2017. A survey of energy-efficient task synchronization for real-time embedded systems. In *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA).*

[13] M. Xiong, S. Han, K.-Y. Lam, and D. Chen. 2008. Deferrable Scheduling for Maintaining Real-Time Data Freshness: Algorithms, Analysis, and Results. *IEEE Trans. Computers* 57, 7 (2008), 952–964.