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:
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 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 \( \tau_1 \) and \( \tau_2 \) communicate through a shared register \( \text{register}_1 \) where \( \tau_1 \) writes input data to be used by \( \tau_2 \) instances.

\[ \text{Figure 1: Consistency violation case} \]

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 \( \tau_1 \) produces an output data which is immediately consumed by the first instance of \( \tau_2 \) at \( t = 1 \). At \( t = 2 \), the second instance of \( \tau_1 \) is released and it preempts the first instance of \( \tau_2 \) due to the high priority of \( \tau_1 \). At \( t = 3 \) a new data is written into \( \text{register}_1 \). At this
moment $t = 3$, the first instance of $\tau_2$ 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 $\tau_2$ 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 as 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 $\tau$ of $n$ periodic tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$ scheduled preemptively on one processor according to fixed-priority scheduling algorithm. Each task $\tau_i$ 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 $\tau_i$. 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 $\tau_i$ has a higher priority than $\tau_j$. Each task $\tau_i$ generates an infinite number of successive jobs $\tau_{i,j} | j = 1, \ldots, \infty$. However, for simplicity, in the reminder of this paper, we do not focus on specific jobs. Therefore, $\tau_i$ 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 $H = \text{lcm} \{T_i\} | i = 1, \ldots, n$. 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 $\tau_3$ is both a producer and a consumer. Task $\tau_3$ is reading from the buffer $\text{Cb}_1$ the data that the task $\tau_1$ has produced. The buffer model is described in Section 2.2.

![Figure 2: System tasks model](image)

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 $\{\tau_1, \ldots, \tau_n\}$ and $E$ is the set of edges where $(\tau_i, \tau_j) \in E$ if $\tau_j$ consumes data produced by $\tau_i$. The graph $G$ may have several components. A task that has no predecessor is a source task and such tasks are often corresponding to actuators. We denote by $\text{pred}(\tau_i)$ the set of predecessors of a task $\tau_i$ and by $\text{succ}(\tau_i)$ the set of its successors. The former and the latter return, respectively, a list of predecessors and successors to $\tau_i$ if they exist. For a source task $\tau_i$ the $\text{pred}(\tau_i)$ has one element by default $\tau_0$ as $\tau_i$ has no predecessors. For a sink task $\tau_f$ the set $\text{succ}(\tau_f)$ contains $\tau_0$ as $\tau_f$ 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 $\tau_i$ is a source task and $\tau_j$ 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 $\tau_i$ has a fork relation with $\tau_{i,j} \cdots \tau_{i,k}$ if $(\tau_i, \tau_j) \in E$, where $j \in \{1, \cdots, k\}$ with $k$ the number of successors for the task $\tau_i$. 

\[ \text{By definition, a fork relation is activated by a sink task.} \]
A join relation implies that a single producer is writing data for several consumers. An example of this relation is \((τ_1, τ_2, τ_3, τ_4) \in E\) 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 \(τ_{j_1}, \ldots, τ_{j_k}\) if \((τ_{j_j}, τ_j) \in E\), where \(i \in \{1, \ldots, k\}\) 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 \((τ_3, τ_2, τ_4, τ_2)\), 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 \((τ_2, τ_5) \in E\), where \(τ_5\) has only one predecessor the task \(τ_2\). 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.

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.

What from precedes, we consider a set of \(l\) buffers \(\{β_1, \ldots, β_l\}\), where each buffer \(β_l\) is characterized by its cardinality \(|β_l|\); 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, \(∀ (τ_i, τ_j) \in E\) where \(j \in \{1, \ldots, k\}\) with \(k\) the number of successors for the task \(τ_j\), with \(β_l\) a buffer that \(τ_j\) shares with all \(τ_i\) and whose size is \(|β_l|\), we aim to compute the optimal size of \(|β_l|\) guarantying against any data consistency violation without any arbitration mechanism or buffer copy making.

**Definition 3.1 (Optimal size).** \(|β_l|\) is optimal if it is guaranteed that \(∀(τ_i, τ_j) \in E\) \(|τ_j| = 1, \ldots, k\) an input data value read from a given slot of \(β_l\) by \(τ_j\) 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 \((τ_j, τ_j) \in E\) \(|τ_j| = 1, \ldots, k\) with \(k\) the number of tasks that produce input data for \(τ_j\), where \(τ_j\) has a join relation with \(τ_j\), 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 \(|β_l|\), is equal to the number of \(τ_j\) instances that can be released and complete their executions before an instance of \(τ_j\) completes its largest execution. Hence,

\[
|β_l| = \begin{cases} 
1, & \text{if } T_j \geq T_j, \\
R_j & \text{if } T_j < T_j.
\end{cases}
\]

where \(|β_l|\) is the cardinality of \(β_l\), \(T_j\) is the period of \(τ_j\), and \(T_j\) and \(R_j\) are, respectively, the period and the worst case response time of \(τ_j\).

**Proof:** Consider \((τ_j, τ_j) \in E\) where \(i\) is one of the \(k\) tasks that produce input data for \(τ_j\),
if \( T_{ij} = T_{ij} \), it means that \( \tau_{ij} \) and \( \tau_{ij} \) instances are always released at a same time instant. Hence, a buffer of size one is sufficient.

- if \( T_{ij} > T_{ij} \), it implies that a data sample produced by an instance of \( \tau_{ij} \) can be read by several instances of \( \tau_{ij} \). Additionally, no instance of \( \tau_{ij} \) can execute before an executing instance of \( \tau_{ij} \) completes. Therefore, a buffer of size one is also sufficient.

- Finally, if \( T_{ij} < T_{ij} \), they may be new productions of \( \tau_{ij} \) between the activation and the completion of an instance of \( \tau_{ij} \). The largest time an instance of \( \tau_{ij} \) can execute is equal to its worst case response time denoted \( R_{ij} \). Within \( R_{ij} \) time units \( \left[ \frac{R_{ij}}{T_{ij}} \right] \) instances of \( \tau_{ij} \) 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 \( \tau_{ij} \) instance, a buffer of \( \left[ \frac{R_{ij}}{T_{ij}} \right] \) size is sufficient.

Theorem 3.3. We consider a fork relation such that \((\tau_i, \tau_j) \in E[j = 1, \cdots, k]\) where \( k \) is the number of tasks whose instances consume data samples produced by \( \tau_i \) instances. The optimal size of \( \beta_i \) is equal to the number of \( \tau_i \) instances that can be released and complete their executions before the completion of an instance of the slowest among all \( \tau_i \). Hence,

\[
|\beta_i| = \left\lceil \frac{\max \{ R_{ij} \}}{T_{ij}} \right\rceil.
\]

where \( |\beta_i| \) is the cardinality of \( \beta_i \); the buffer where \( \tau_i \) instances write the data needed by \( \tau_{ij} \) instances, \( T_i \) is the period of \( \tau_i \) and \( R_{ij} \) the worst case response time of \( \tau_{ij} \).

Proof: Within a fork relation, a single task is producing data for several consumer tasks which may have different sampling periods. So, \( \forall (\tau_i, \tau_{ij}) \in E[j = 1, \cdots, k] \) where \( k \) is the number of tasks whose instances consume data produced by \( \tau_i \) instances. \( |\beta_i| \) is computed by focusing on the \( \tau_{ij} \) task whose instances read the input data slowly. In other words, they are the one having the largest worst case response time among all \( \tau_{ij} \). Intuitively, if the accessed slot cannot be overwritten before the completion of the instance of the slowest among all \( \tau_{ij} \) tasks, then this can be the case for all \( \tau_{ij} \). 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 by \( p \) 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 \( (\tau_i, \tau_{ij}) \in E \), the writing point of the \( p^{th} \) data sample by \( \tau_i \), denoted \( w_{i,p} \), corresponds to the completion time of the \( p^{th} \) instance of \( \tau_i \), where \( p \) in the subscript and \( p \) in superscript refer respectively to the \( p^{th} \) instance of \( \tau_i \) and the \( p^{th} \) data sample already produced by \( \tau_i \) instances. Formally,

\[
w_{i,p} = p \star T_i + C_i + I_k
\]

with

\[
I_k = \sum_{p \in \mathbb{P}(t_i)} w_{i,p} \cdot \frac{w_{i,p}}{T_k} + C_k
\]

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

The same way, the writing time of the data sample by the next instance of \( \tau_i \); that is \( p + 1 \) is denoted \( w_{i,p+1} \) 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 \( \tau_i \) is not an actuator task and \( p \) the \( p^{th} \) data sample produced by \( \tau_i \). At a time instant \( t \), if we have

\[
w_{i,p} \leq t < w_{i,p+1}
\]

then the \( p \) is temporal valid.

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

Hence, \( \forall (\tau_i, \tau_{ij}) \in E \), the data produced by \( \tau_i \) instances are meant to be consumed by \( \tau_{ij} \) instances. Let \( t_{ij} \) be the reading time of the \( p^{th} \) data sample by the \( q^{th} \) instance of \( \tau_{ij} \). From the algorithms:

**Algorithm 1: Algorithm calculating the buffer's optimal size**

1. **Require:** \( t_i | \{ 1, \cdots, n \} \), \( \omega_j | \{ 1, \cdots, m \}, \beta_k[k = 1, \cdots, l] \)
2. **Initialize:** \( \text{pred}(\tau_i) \leftarrow -1 \), \( \text{succ}(\tau_i) \leftarrow -1 \)
3. **for each** \( \omega_j \) **do**
   4. **for each** \( \tau_j \in \omega_j \) **do**
      5. **return** \( \text{pred}(\tau_j), \text{succ}(\tau_j) \)
8. **if** \( \text{size}(\text{succ}(\tau_j)) \geq 1 \land \text{succ}(\tau_j) \neq -1 \) **then**
10. **else**
12. **for each** \( \tau \in \text{succ}(\tau_j) \) **do**
14. **Return** \( \max \left\{ \frac{R_{\text{succ}(\tau_i)}}{T_i} \right\} \)
16. **end**

**References:**

1. C. B. A. and E. Ntaryamira et al.
tasks execution order point of view, the propagation of the \( p^{th} \) data sample up to \( t_i \) is delayed by all tasks of higher priority than \( t_i \) released in the time interval bounded by \( w_{i,p} \) and \( q_{i,p} \). The reachability property is always verified for all the cases where \( T_i \geq T_j \), since each data produced by \( t_j \) will be used by at least one instance of \( t_i \). However, if \( T_i < T_j \), some data will be overwritten before being used and an instance of \( t_j \) 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.

\[
q = \begin{cases} 
\frac{p \cdot T_i}{T_{ij}}, & \text{if } T_{ij} \mod T_i = 0 \\
\left(\frac{(p+1) \cdot T_i}{T_{ij}}\right) + 1, & \text{Otherwise}
\end{cases}
\]  

(6)

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

\[
q \cdot T_{ij} - (p + 1) \cdot T_i \leq T_i
\]  

(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 \( p \cdot T_i \) time instant and \((p, p + 1, \cdots, p + k)\) instances are also released before the activation of the \( q^{th} \) instance of \( \tau_i \), then the \((p, p + 1, \cdots, p + k)\) data samples may have been overwritten or are still queued waiting to be probably read by this \( q^{th} \) instance of \( \tau_i \). 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 \( \tau_i \). In this context, only \((p + k)\) data sample is consumed by \( q^{th} \) instance of \( \tau_i \).

### 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 \( \sigma \).

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

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 \( \sigma \) is computed online based on the following principle:

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

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

<table>
<thead>
<tr>
<th>( \tau_1 )</th>
<th>( D_i )</th>
<th>( T_i )</th>
<th>( R_i )</th>
<th>( \beta ) name</th>
<th>( \beta ) size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>(cb_1)</td>
<td>(</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>(cb_4)</td>
<td>(</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>(cb_2)</td>
<td>(</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>16</td>
<td>16</td>
<td>(cb_3)</td>
<td>(</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>24</td>
<td>24</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

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:

(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 \( \tau_2 \) 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 1cdfd8 (cb_2) and 0 from 1cdf9e (cb_3). The above holds for all tasks within \( \tau \).

(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 1cd5c4 and 1cd5c8. This buffer is used by both task \( \tau_3 \) and \( \tau_4 \) for their respective input data. Let us focus on the task \( \tau_4 \) as for instance. The first instance of \( \tau_4 \) read input data at \( t = 3 \), where it read value 58 from 1cd5c4. This instance completed its execution at \( t = 8 \). We need to prove that 1cd5c4 will not be accessed by (the instances of \( \tau_1 \) activated at \( t = 3 \)) before the completion time of that instance of \( \tau_4 \). Indeed, the next writing time of an instance of \( \tau_1 \) being between \( t = 3 \) and \( t = 8 \) happened at \( t = 5 \) and wrote value 5 in the next slot 1cd5e8 and the second that overwrote 1cd5c4 content happened at \( t = 9 \), which is right, since the instance of \( \tau_4 \) that used data from 1cd5c4 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 \( \tau_2 \) that read data produced by \( \tau_2 \). The first instance of \( \tau_5 \) activated at \( t = 10 \) consumed the last data produced by an instance of \( \tau_4 \) that completed at \( t = 7 \) while the second activated at \( t = 27 \) consumed the last data produced by an instance of \( \tau_2 \) 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
