An efficient FIFO buffer management to ensure task level and effect-chain level data properties

Evariste Ntaryamira, Cristian Maxim, Therence Niyonsaba, Liliana Cucu-Grosjean

An efficient FIFO buffer management to ensure task level and effect-chain level data properties

Evariste Ntaryamira*†, Cristian Maxim‡, Therence Niyonsaba† and Liliana Cucu-Grosjean*
*INRIA Paris, France, firstname.lastname@inria.fr
†University of Burundi, Burundi, firstname.lastname@ub.edu.bi
‡IRT-SystemX, France, firstname.lastname@irt-systemx.fr

***Abstract*—Real-time embedded systems (automotive, avionics,drones autopilots, etc.) are composed of numerous functional components that are continuously interacting via a variety of communication models where they intensively share data. For the overall functional correctness these systems must verify not only real-time scheduling requirements but they also must guarantee that the data being used are qualitatively correct. The quality of the data reflects the preservation of data related properties: temporal properties (i.e freshness, end-to-end latency, etc.) and integrity related properties (i.e data consistency). The proposed protocols to ensure such properties highly depend on the considered communication model (shared registers or large buffers) and the data access policy (directly or via local copies). In order to overcome this limitation, we provide in this paper, the means for managing the FIFO buffers to guarantee these data properties in a way that communication dependencies do not impact the tasks system scheduling order. We do so while considering the communication model presented in [14]. We provide methods computing a sufficient size for the considered buffers. Last but not least, an algorithm implementing the "last reader tags mechanism" together with the data temporal matching is provided and explained.**

I. Introduction

One of the specificity of the real-time systems is that they must react timely to the events in the environment. From the real-time scheduling point of view, these systems must be carefully verified and validated provided that functional failure may result in significant consequences. For that purposes,

Real-time embedded systems are composed of large number of applications that continuously communicate. One of their specificity is that they must react timely to the events in the environment. Part of the applications are made to produce the data to be utilized by the others. The producing and consuming applications are referred to as producers and consumers, respectively. These applications may have different sampling rates (periods) and , accordingly, some output data may never be used while others may be read several times. Additionally, such application may be triggered by different clocks which may be synchronized or not or for which clocks synchronization is a challenging task. Additionally to that, communicating applications may be sampling at different rates (periods). For the correct functioning of the system, all these particularities must be taken into account when determining the scheduling policies. Inter-application communications are often ensured through shared variables where the output of one application is used as input for another application and so on and so forth. A sequence of such applications, involved in the definition of a given function, is referred to as *functional chain (i.e [13]*) or as *cause-effect chain (i.e [11]).*

With the evolution of technologies, real-time embedded systems are getting more and more intelligent in the sense that, at some point, they acquire the capability to achieve targeted functions autonomously. For instance, the autonomous vehicles or drones are extended with the capability of sensing the surrounding environment and navigating on their own by making driving decisions. The correctness of these driving decisions depends not only on the system schedulability but also, and importantly, on the quality of the data being used. By the quality of the data we mean the ***temporal properties*** of the data such as the *freshness*, *end-to-end latency*, and the ***integrity related properties*** such as *the data consistency* which consists in protecting the data from the shared resources against any corruption or modification when this data is still being utilized by some currently executing applications. Part of such properties are required to be ensured on the application level (i.e freshness, consistency) while others are required on the cause-effect chain level (i.e end-to-end latency, data age). Accordingly, in the remainder of this paper we clearly classify such properties into *task level* and *effect-chain level* properties, respectively.

Regarding the task level properties, the data freshness is implicitly ensured for communication channels having the storing capacity of only one data sample. This is the case for the register-based model or the buffers of size one. The reason is that only one data sample can be available at a given time instant and when a new data is produced, the old one is overwritten. This being, for such models, maintaining the data consistency can't be achieved implicitly. This is usually taken care of by the operating system which relies on certain arbitration mechanisms (i.e semaphores, mutexes, different synchronization protocols). However, these arbitration mechanisms may have a negative impact on the system schedulability with prospect of leading to an unpredictable system behavior as they may provoke priority inversion problems and possible deadlock formations [4]–[6]. Mechanisms guaranteeing the data consistency are of three categories (*lock-based, lock-free* and *wait-free*) as detailed in Section II. Such mechanisms are meant to support generally very basic communication models that need to be extended to properly maintain different data properties (at different levels, preferably with zero or less

impact on the system schedulability) in more complex systems implementing complex communication semantics such as *publish/subscribe* mechanism using the μORB (The Micro Object Request Broker). The latter is implemented in the PX4 autopilot [14] where it operates in the form of *active polling*; that is, the execution is triggered by the production of new data sample and, an application ready to execute, blocks until a new data sample is produced.

Functional requirements in the autonomous systems impose new communication requirements in such a manner that the utilization of sample shared variables does not meet them. In the spirit of meeting such requirements, the communication within the autopilot [14] is ensured in terms of semantic messages, where a semantic message is a set of several variables considered together to form a data structure with an intuitive name with respect to the physical parameter it represents (i.e $position, altitude, etc.$). Accordingly, ensuring application (task) level and effect-chain level data properties while taking into account such complex communication semantics requires specific and precise approaches guaranteeing the data management predictability (in addition to the system scheduling predictability).

Obviously, the structure of this buffer may be complex (i.e) in such a way that the use of registers as communication channel may not handle it properly. Instead, the data should be stored into re-sizable communication channels such as memory buffer which size may be bigger than one.

However, in order to cope with this challenge, one might prefer to implement direct data access mechanisms with arbitration mechanisms which may negatively impact the system scheduling as state previously or the direct data access without arbitration mechanisms to the price of using global variable local copies (Implicit communication). It may also be preferred to use larger buffers with an asynchronous direct access also called the (*wait-free mechanism*), which we consider in the scope of this paper.

Let us consider, for instance, the model presented on the Figure **??**. The tasks τ_1, τ_2 and τ_3 access concurrently the communication channel var where on the Figure **??** (A) var is a register of size one accessed directly, on the Figure **??**(B) var is a global variable accessed through local copies var_2, var_3 and var_4 (a copy for each reader) and finally on the Figure **??** (C) var is a circular buffer accessed directly based on the task priory, its storing capacity or size is equal to 2 which is computed considering the tasks timing parameters.

Most of the time the communication between applications is ensured through shared registers of size one or larger memory buffers. The shared variable may be a simple variable as it may be in the form of semantic message where the latter is seen a set of one or several variables considered together to form a new variable to which an intuitive name is assigned with respect to the physical parameter it represents

for instance, to represent an object in the space we need to specify 3 variables: x, y and z, which normally provides the *position* of this object. Similarly,from the system considered in [**?**], the vehicle *position* is defined the following way: $position\{uint64\ timestamp;\ float64\ lat;\ float64\ lon; \cdots\}$. In result, the semantic messages are of complex structures [14]. In the rest of this paper we'll be simply using the term *message* to designate *semantic message*.

In addition to these properties, in this paper, we introduce an other chain level temporal property that we referrer to as the *data matching*.

This property is important A single application may belong to different propagating chains. These chains may have different timing characteristics, so information which transits on these chains takes various time to reach the final application. However, depending on the targeted function (object recognition for instance), data propagating from different chains may need to adequately associated, so that the final application uses inputs which all result from the same execution step of the initial application. We call this property the *data matching*.

In addition to that, they must fulfill heterogeneous functions such as *classical functions* (control-command, data logging, display,$\cdots$) and *intelligent functions* (recognition, data fusion,$\cdots$). We should note that each of the category of functions presents its own requirements.

for instance, let us consider the following situation: *The consumer application is preempted before reading the entire input data (due to the complex data structure), and in between the preemption and resume times, the producer writes a certain number of data samples. Further, imagine that the communication channel is a register of size one. Obviously, by the time the previously preempted application is re-activated, it goes on reading and fetches the updated data value*. Not only this provokes the *data consistency* violation but also combining old and new data in the same execution process might lead to erroneous results or performance degradation.

A single application may belong to different propagating chains. These chains may have different timing characteristics, so information which transits on these chains takes various time to reach the final application. However, depending on the targeted function (object recognition for instance), data propagating from different chains may need to adequately associated, so that the final application uses inputs which all result from the same execution step of the initial application. We call this property the *data matching*.

`Paper contribution`: The first contribution of this paper consists in showing how the FIFO communication buffers can be exploited efficiently to increase the data management determinism while not considering any kind of arbitration mechanism. To that end, we adopt a fully asynchronous wait-free data access policy exploiting the system scheduling policy and the applications' timing parameters. Further, while taking advantage of the manageability of the FIFO buffers, a general framework easing to ensure the *task level data property* and the *effect-chain level data property* is provided. *Freshness*, *consistency* and *matching* are the three data properties to be ensured in this paper. *Data matching* is required when there exists in the system an application managing (associating) data from different chains and making decisions on the basis of these data samples. In this paper we assume that the

data to be associated originally result from the same source application, *single-source data matching*. Accordingly, only data samples resulting from the same execution step of the source application must be associated. An example of meeting this requirement is the FADE system presented in [22]. The latter is a vehicle detection and tracking system composed by a set of image processing components in charge of detecting the characteristics (detection of shadows, headlights, etc.) related to the presence of a vehicle (or other moving objects) in the neighborhood. Herein, an image captured by the camera sensor is propagated across two different chains (each chain treating a particular part of the image data, i.e central part or image periphery) and the outputs of each chain are associated together by an application in charge of inferring the identity of the object. Obviously, only outputs resulting from the same original data image must be associated considering that they describe a same image. Handling this issue is a challenging task given that the propagation chains may have different propagation delays and communicating applications may have different periods.

II. Related works

Regarding the data consistency maintenance, Zeng and al. proposed in [7] a survey on existing protocols for supporting communication over shared memory or, in general, protecting shared resources . An experimental evaluation for all theses mechanisms is also provided. Therein, these mechanisms are classified into three mains categories:

- *Lock-based*: A lock is utilized to arbitrate the access to the shared memory. Precisely, when a task needs to access the shared memory while another task holds the lock, it blocks.
- *Lock-free*: Each reader accesses the communication data without blocking. At the end of the operation, it performs a check. If the reader realizes that there was a possible concurrent operation by the writer and it has read an inconsistent value, it repeats the operation.
- *Wait-free*: Readers and writer are protected against concurrent access by replicating the communication buffers and by leveraging information about the time instants when they access the buffer or other information that constrains the access (such as priorities or other scheduling related information). For more details on this mechanism the reader should refer to [7] for details.

Our proposed mechanism is different from existing work as follows. We consider a fully asynchronous method without any kind of arbitration mechanisms which is not the case for the existing mechanisms. There is no dynamic allocation of local copies. Instead, we use buffers for which sizes are computed based on the tasks timing parameters and priorities. The necessary memory buffers are allocated once at the system start-up and the occupied memory addresses will always be the same until the system stops. The issue related to the maintenance of the data matching is addressed by Pontesso and al. in [12]. However, authors assume that the FIFO buffers are correctly managed, while in this paper we propose means to manage efficiently FIFO buffers based on the system scheduling parameters. Additionally, authors consider the *logical execution model* while in this paper we consider that each application reads all the input data at its release time and writes back the results at the completion time. The written data sample becomes immediately available for reading.

III. The systems modeling

A. The tasks system model

We consider a periodic time-triggered system $\mathcal{T}$ composed of n tasks $\{\tau_1, \cdots, \tau_n\}$ executing upon a uni-processor or partitioned multiprocessor platform. The tasks are independent and scheduled preemptively based on a fixed-priority scheduling algorithm such as Rate Monotonic [1] or Deadline Monotonic [9]. Each task τ_i is described by the tuple $(c_i, C_i, T_i, \Gamma_i, \mathcal{W}_i)$, where c_i is the best-case execution time, C_i is the worst-case execution time, T_i is the period, Γ_i the task execution state and $\mathcal{W}_i$ the task publication (writing) status. All tasks are released simultaneously and they have implicit deadlines; $\forall \tau_i,\ T_i = D_i$. A task τ_i may be in two different execution states (Γ_i): `initial state,` $\Gamma_i = -1$ which corresponds to the case when τ_i has never been released or when its current job has completed. This parameter is used to control if the ready job is just starting its execution or is resuming it. The task publication parameter ($\mathcal{W}_i$) is used to control the task writing moments. The $\mathcal{W}_i$ parameter guarantees that the data publication happens at predictable instants (and not anywhere in the program; i.e., during the interrupts as it is the case for [14]).

At the release time, each task reads the required inputs, performs some computations using these data and output the results at the completion instant. Hence, C_i is the sum of the time spent on the reading of the input data, the processing time and writing output results. Any job of the reader released after the execution completion of the writing task can read the produced data.

Tasks are released periodically and their priorities are assigned according to Rate Monotonic. Each task τ_i generates an infinite number of jobs and, for the sake of the simplicity, in the reminder of this paper, we do not focus on specific jobs. Therefore, τ_i has the meaning of any of its jobs. We define the hyper-period as the least common multiple of the periods of all tasks and we denote it by $\mathcal{H} = lcm\{T_i\} | i = 1, \cdots, n$. The interval $(0, H)$ is a feasibility interval for the task system $\mathcal{T}$ [19] since all the tasks are released simultaneously and have implicit deadlines. By feasibility interval we understand the smallest time interval such that if all deadlines are met within this interval, then all deadlines are met for the entire system.

B. The communication model

The tasks communicate through the bounded FIFO circular buffer known to be a *FIFO data structure* that considers memory to be managed circularly. Such buffer has in its structure the *tail* and *head* pointers that loop back to 0 after their values reach the size of the buffer. The size is fixed and allocated once at the system run-time. The memory addresses

occupied by a given shared buffer will never change during the system execution which has the advantage of avoiding dynamic memory allocation. For such an asynchronous data access, bounding the sizes requires to have knowledge of the communicating tasks timing characteristics and the scheduling policy such that the real-time system is schedulable, as it is assumed in the scope of this paper.

The access to the shared buffer is based on the *single writer, many readers principle* and is asynchronous and non-blocking. Precisely, the tasks accessing the buffer for writing (producer) and reading input data (consumers) are able to access the shared buffer independently of the execution state of each other without blocking, using only temporal characteristics guaranteed by the real-time scheduling and a sufficiently large circular buffer to manage concurrency.

1) ***`Message formalization`****:* Let $\mathcal{M}$ be the set of messages that are being exchanged between the tasks of $\mathcal{T}$. A single task may produce/consume a certain number of messages. Subsequently, we denote by $\mathcal{M}_i$ the set of input and output messages of the task τ_i , where $\mathcal{M}_i \subset \mathcal{M}$. Input and out messages of τ_i are denoted by $\mathcal{M}_i^{In}$ and $\mathcal{M}_i^{Out}$, respectively.

2) ***`Data sample characterization`****:* Each buffer has the capability of storing a finite number of data samples related to a given message. The set of data samples related to the message m is denoted by $\mathcal{D}^m = \{\partial_1^m, \cdots, \partial_\omega^m\}$ for all $m \in \mathcal{M}$ where ω is the total number of data samples that have been produced. $\partial_i^m = (t^m_{stamp_i}, \Upsilon_i^m, t^m_{issue_i}),\ \forall i \in [1, \omega)$ where i is the i^{th} data sample, $t^m_{stamp_i}$ its timestamp, $t^m_{issue_i}$ its time of issue and Υ_i^m is the set of values affected to each of the buffer structure members for the i^{th} data sample such that $\Upsilon_i^m = \{v^m_{i,1}, \cdots, v^m_{i,n}\}$ with n the number of the buffer members. The data timestamp is the intrinsic date of the data sample. It should be as close as possible to the date of occurrence of the data sample. It is supplied by the task that originally produces the message owning this data sample and remains unmodified until all the data samples resulting from this data sample are overwritten and disappeared from the system. Additionally to the timestamp, at the execution completion, each job adds this completion time to the data sample in terms of time of issue. The time of issue changes from task to task.

C. The communication graph

Definition 1 (`The communication graph`): The communication graph is a bipartite graph $\mathcal{G} = (\mathcal{T}, \mathcal{M}, \mathcal{E}^{\mathcal{M}})$ such that $\mathcal{T}$ is the set of tasks, $\mathcal{M}$ is the set of messages and $E^{\mathcal{M}}$ is the set of linking edges between the tasks of $\mathcal{T}$, where messages of $\mathcal{M}$ represent bridges on these edges; n and m the cardinality of $\mathcal{T}$ and $\mathcal{M}$, respectively.

Two tasks $(\tau_i, \tau_j) \in \mathcal{T}$ don't communicate directly; there must be a message of $\mathcal{M}$ to bridge this communication where one of these tasks is the `message producer` and the other is the `message consumer`. For instance, the notation $\mathcal{E}^{msg} = \{\tau_i | \tau_j\}$ is used to show that τ_i and τ_j communicate through the message msg where τ_i is the producer and τ_j the consumer, for all $msg \in \mathcal{M} \wedge (\tau_i, \tau_j) \in \mathcal{T}$. In other words, two tasks τ_i and τ_j communicate though the message msg iff $(\tau_i, \tau_j) \in \mathcal{E}^{msg}$.

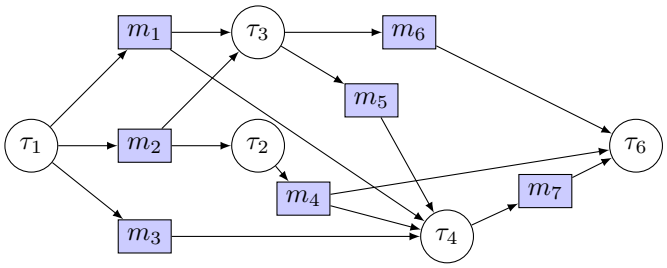

Figure 1: Example of the communication graph

For instance, considering the communication graph from the Figure 1, $\mathcal{E}^{m_1} = \{\tau_1 | \tau_3, \tau_4\}, \mathcal{E}^{m_2} = \{\tau_1 | \tau_2, \tau_3\}, \cdots, \mathcal{E}^{m_7} = \{\tau_4 | \tau_6\}$.

IV. Task level data property maintenance

In the scope of this paper, the task level properties (data consistency and data freshness) are ensured while considering the following:

- The *single producer, many consumers data access principle*: Each buffer can be accessed by a single producer for writing and one or many consumer(s) for reading.
- The access to the buffer (for writing/reading) is asynchronous and non-blocking.

A. Data consistency

Preserving of the *data consistency* property consists in ensuring that no data sample will be overwritten as long as there exists at-least a single job of one of the readers still processing, considering that the tasks access the data samples asynchronously and with no arbitration mechanisms.

B. Data freshness

A given data sample is *fresh* if and only if, from the time instant this data was written into the buffer, no other job of the producer hasn't completed yet.

C. Relation to existing work

The issue related to the preservation of the data freshness and consistency has previously been addressed in [18]. Herein, authors propose a method computing the minimal (optimal) size of each of the buffers. An algorithm ensuring these properties while considering the found buffer size is proposed and proved. The buffer sizes are found based on the *lifetime bound method [15], [16]* which is based on the computation of an upper bound of the number of times the writer can produce new data samples while a given data sample is being used by at least one reader. Accordingly, since only one task can write into a given buffer, the size of each buffer is computed as the ceiling of the reader worst-case response time over the sampling period of the producer task if only one task reads from this buffer. In the case when several tasks read from this buffer, the size of the buffer is computed considering the largest of the readers worst-case response time as shown in

the Equations 1 and 2. Formally, for a buffer β^m where a task τ_i writes the data samples related to the message m and the tasks $\{\tau_{i_1}, \cdots, \tau_{i_k}\}$ such that $\mathcal{E}^m = \{\tau_i | \tau_{i_1}, \cdots, \tau_{i_k}\}$ where k is the number of tasks reading from β^m, when $k = 1$ we have

$$|\beta^m| = \left\lceil \frac{R_{i_k}}{T_i} \right\rceil \tag{1}$$

while for the case where $k \geq 2$ we have

$$|\beta^m| = \max_{1<j\leq k} \left\{ \left\lceil \frac{R_{i_j}}{T_i} \right\rceil \right\} \tag{2}$$

where R_i, $\forall \tau_i \in \mathcal{T}$, is the worst-case response time [3] which corresponds to the maximum time delay it can take for the first job of τ_i to complete its execution when released simultaneously with all the higher priority tasks denoted here by $hp(i)$.

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil * C_j \tag{3}$$

V. Effect-chain level property maintenance: Data matching

With respect to the single source data matching category, the data matching property consists in associating data samples resulting from the same execution step of the source task. The set of cause-effect chains, involved in the propagation of the data samples from the source task until the destination task, form a `spindle` that we define as follows:

Definition 2 (`Spindle`): A spindle, denoted by $\mathcal{S}_q^m(\tau_{src}, \tau_{trm})$, is the set of q chains propagating the data samples related to a message m such that all these chains have in common the *spindle source task* τ_{src} and the *spindle terminus task* τ_{trm} task, with $q \geq 2$.

Each chain of $\mathcal{S}_q^m$ is denoted by $\mathcal{C}_{c:q}^m$ with c the index of a specific chain, $\forall 1 \leq c \leq q$. The chains composing the spindle can be linear or branched.

Definition 3 (`Linear chain`): A chain $\mathcal{C}_{c:q}^m \in \mathcal{S}_q^m$ is said `linear` iff $\forall c', c' \in [1, q] \wedge c' \neq c$ we have $\mathcal{C}_{c:q}^m \cap \mathcal{C}_{c':q}^m = \varnothing$.

Definition 4 (`Branched chain`): A chain $\mathcal{C}_{c:q}^m \in \mathcal{S}_q^m$ is said `branched` iff $\exists \mathcal{C}_{c':q}^m \in \mathcal{S}_q^m$ such that $\mathcal{C}_{c:q}^m \cap \mathcal{C}_{c':q}^m \neq \varnothing$, $c, c' \in [1, q] \wedge c' \neq c$.

In the presence of one or many branched chains, the number of initial chains may differ from the one of chains reaching the end of the spindle. Accordingly, we denote by p the number of initial chains while q keeps the meaning of the number of the chains reaching the spindle terminus task. For instance, regarding the spindle on the Figure 4, $p = 2$ while $q = 3$. On this basis, we distinguish the following classes of spindles: *pure*, α-*pure* and β spindles.

Definition 5 (`Pure spindle`): $\mathcal{S}_q^m$ is a pure spindle iff $p = q$ and all q chains are linear.

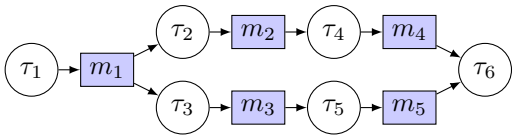

Figure 2: A pure spindle example.

Definition 6 (α-`pure spindle`): $\mathcal{S}_q^m$ is a pure spindle iff $p = q$ and there is a number α of embedded pure spindles in its composition.

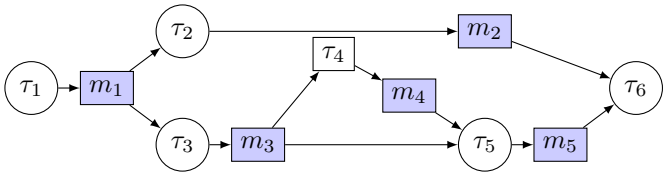

Figure 3: An α-*pure spindle* example.

Definition 7 (β-`spindle`): $\mathcal{S}_q^m$ is a β-spindle iff $p < q$.

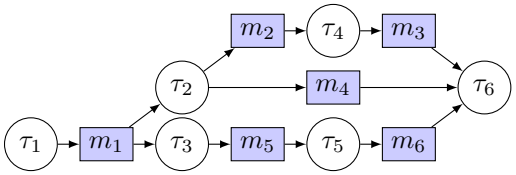

Figure 4: A β-spindle example.

In the scope of this paper we focus solely on the `pure spindle`.

The value of p has also the meaning of the number of tasks reading from the *spindle source buffer* that we define as follows:

Definition 8 (`Spindle source buffer`): Let β^m be the buffer storing the data samples related to the message m. β^m is the spindle source buffer iff $\exists \mathcal{S}_q^m \wedge \{\tau_{src}, \tau_{src_1}, \cdots, \tau_{src_p}\} \in \mathcal{T}$ such that $\mathcal{E}^m = \{\tau_{src} | \tau_{src_1}, \cdots, \tau_{src_p}\}, \forall 2 \leq p \leq q$.
where τ_{src} is the producer of the message m and $\tau_{src_1}, \cdots, \tau_{src_p}$ are the tasks reading from the buffer β^m.

Further, the spindle terminus task reads the input data from q different buffers where each of these buffers is written by the last task belonging to each of the q reaching chains (except the spindle terminus task). We analogically name each of these buffers as the *spindle chain terminus buffer*.

So, let β^{m_c} be the output buffer of the $\mathcal{C}_{c:q}^{m_c}$ such that c is the index of a specific chain and $1 \leq c \leq q$ with $q \geq 2$:

Definition 9 (`Spindle chain terminus buffer`): β^{m_c} is a spindle chain terminus buffer iff $\exists (\tau_{trm_c}, \tau_{trm}) \in \mathcal{T}$ such that τ_{trm} is the spindle terminus task and $\mathcal{E}^{m_{trm_c}} = \{\tau_{trm_c} | \tau_{trm}\}$, $\forall 1 \leq c \leq q$.
where τ_{trm_c} is the last task belonging to the chain of index c. This task is the producer of the message m_{trm_c} being one of the spindle terminus task input message.

A. The solution overview

Addressing the *single-source data matching* problem requires that the spindle terminus task associate only the data samples resulting from the same execution step of the spindle source task. Precisely, by the time a job of the spindle terminus task is released there should be, in each of the *spindle chain terminus buffers*, the data samples resulting from the same execution step of the spindle source task. However, for a

multi-rate system the propagation of the data may be broken due the gap between the sampling periods of the message producers and consumers which may lead to the loss of some data before reaching the terminus task. On the other hand, different chains may have different data propagation delays. Subsequently, the propagation of the data samples produced during the same execution step of the spindle source may not reach the spindle terminus task. In order to overcome the aforementioned situation, in the following we propose the steps leading to the solution:

***Step** 1:* Forcing all the consumers reading from the spindle source buffer to propagate the same data samples. For this purpose, we propose the *last reader tags mechanism* that we present in the Section V-B1.

***Step** 2:* Computing the optimal (minimal) size of spindle source buffer in such a manner that the *data consistency* of the read data is always guaranteed. This is ensured in the Section V-B2.

***Step** 3:* Computing the propagation delay for each propagation chains and the optimal size of each of the *chain terminus buffer* such that there will be in each of these buffers at-least one data sample resulting from the same execution step of the spindle source task. The solution is proposed along the Section V-B3.

***Step** 4:* Implement the *temporal alignment and matching algorithm* to ensure that the spindle terminus task reads the right data samples from each of the chain terminus buffers.

B. The data matching maintenance

*1) **The last reader tags mechanism**:* The main idea behind this mechanism resides in modifying the rules under which the access to the spindle source buffer is done. This is organized in a way that all the readers involved in the single source data matching process, can always propagate the same data samples. In other words, this means that all the tasks reading from this buffer should always have the same value of the *tail pointer*, as opposite to the classical organization of the circular buffer where each reader follows its own tail pointer. This mechanism relies on the following rule: *if a data sample is consumed by one of the readers then all the readers will consume it until the execution completion of the job of the slowest among these readers. Only at this instant, another sample can be chosen*. The implementation of this mechanism is performed at the price of the larger memory required to guarantee the consistency of this data which must not be overwritten as long as there is at-least one task still processing it. Accordingly, let us consider the message domain $\mathcal{E}^m = \{\tau_{src}|\tau_{src_1,\cdots,\tau_{src_p}}\}$ and the buffer β^m where p is the number of readers. Let hpp and lwp be, respectively, the higher and the lower priority tasks among $\{\tau_{src_1}, \cdots, \tau_{src_p}\}$. From the fact that all the tasks are released simultaneously, we formulate Lemma 1 regarding exclusively the first data sample produced by τ_{src}.

***Lemma** 1:* Setting $tail(\tau_{src_c}) \leftarrow tail(hpp)$ forces the first job of any task τ_{src_c} to read the same data sample from β^m, and the job of lwp is the one processing this data lastly with c the index of a given chain, $\forall c \in [1, p]$.

***Proof** 1:* When all the tasks are released simultaneously, the execution of the first job lasts for the worst-case response time and the lower is its task priority the latter this happens. Accordingly, the first job of lwp is the one reading the data lastly while the one of hpp reads the data firstly. After the job of hpp had read this data, all jobs released after its completion will have to inherit this tail value until the completion of the job lwp.

For the correctness of the last reader tags mechanism, all the jobs released after the completion of job of the lwp will start reading a different data sample. To that end, we formulate the Lemma 2, which is generalized to all the data samples written into β^m.

***Lemma** 2:* Setting $tail(\tau_{src_c}) \leftarrow tail(lwp)$ forces each task $\tau_{src_c,\forall c\in[1,p]}$ to read the same data from β^m.

***Proof** 2:* Analogically to the Lemma 1, once the job of the lower priority among the readers lwp completes, a new data sample is tagged (marked) and can start being used by any job released after this instant and that until the completion of the next job of lwp. At the completion of the job of lwp the new data sample to be read is the one being into the slot number given by

$$(head - 1 + |\beta^m|) \mod |\beta^m|.$$

The current value of $head$ points to the slot where to write at the next completion. That is why we take the data recently written $(head - 1)$ to which we add $|\beta^m|$ to avoid negative values.

*2) **Computing the spindle source buffer size**:* On the basis of the lifetime bound method we need to compute the largest amount of time the tagged data can still be in use by at-least one of the consumers with respect to the last reader tags mechanism. We call this delay the *spindle source buffer data consistent interval* which we denote by **SCI** and compute on the basis of the Lemma 3.

***Lemma** 3:* We consider $\mathcal{E}^m = \{\tau_{src}|\tau_{src_1,\cdots,\tau_{src_p}}\}$ and two consecutive jobs of lwp that we denote by lwp_j and lwp_{j+1}. The value of **SCI** is found when the data tagged at the completion of lwp_j was produced as early as possible (by τ_{src}) and the execution of lwp_{j+1} lasts for the lwp worst-case response time.

***Proof** 3:* The delay separating two consecutive tagged data directly depends of the time instant the tagged data was written into the buffer and the time instant when the next data is going to be tagged. If the tagged data was written as earlier as possible and the next data is tagged as late as possible (which is when the tagging task executes for its worst-case response time), the delay separating these data is the maximum possible. Hence,

$$SCI = T_{lwp} - c_{src} + R_{lwp} \tag{4}$$

where c_{src} is the best-case execution time of the task τ_{src}, T_{lwp} and R_{lwp} are, respectively, the period and the worst-case response time of lwp.

With respect to the life time bound method, the size of the spindle source buffer is given by the maximal number of data samples that may be produced between two consecutive data tagging instants. The Theorem 1 provides its formal computation.

***Theorem** 1:* We consider β^m the spindle source buffer related to the message m.

$$|\beta^m| = \begin{cases} \left\lceil \frac{SCI}{T_{src}} \right\rceil, & \text{if } T_{src} \leq T_{lwp}. \\ 1, & Otherwise \end{cases} \quad (5)$$

***Proof** 4:* From the Equation 4, the largest delay that can separate two consecutive data tagging instants is equal to the SCI value. So, the computation result of the Theorem 1 guarantees that within SCI time interval, there may not be produced more than $|\beta^m|$ data samples, which would lead to overwriting some data. Subsequently, any data sample read from the spindle source buffer will never be overwritten before the completion of all jobs having read it.
For instance, considering the scheduling results presented on the Figure 5, we have

$$|\beta^m| = \left\lceil \frac{T_3 + R_3 - c_1}{T_1} \right\rceil = \left\lceil \frac{25}{6} \right\rceil = 5 \; slots \quad (6)$$

3) ***Computing the spindle chains propagation delays**:* In this section we compute the minimum and the maximum delays it takes for a data sample to propagate from the spindle source until the spindle terminus tasks for each of the chains involved into this task.

We start by decomposing the c^{th} chain belonging to the spindle $\mathcal{S}_q^m$ as follows:

$$\mathcal{C}_{c:q}^m = \underbrace{\tau_{src} \rightarrow}_{\text{Delay 1}} \underbrace{\tau_{src_c} \rightarrow \tau_i \rightarrow \tau_{i+1} \cdots \tau_n \rightarrow}_{\text{Delay 2}} \underbrace{\tau_{trm_c} \rightarrow \tau_{trm}}_{\text{Delay 3}}$$

where

- τ_{src} and τ_{trm} are the spindle source task and the spindle terminus tasks, respectively.
- τ_{src_c} and τ_{trm_c} are, respectively, the task reading from the spindle source buffer and the task writing into the spindle chain terminus buffer; both belonging to the chain of index $c, \forall c \in [1, p]$.
- And finally, $\tau_i \rightarrow \tau_{i+1} \cdots \tau_n$ is the chain segment comprised between τ_{src_c} and τ_{trm_c}.

The minimum and the maximum propagation delay of the c^{th} chain that we respectively denote by $\mathcal{D}_{c:q}^{min}$ and $\mathcal{D}_{c:q}^{max}$, are computed as the sum of the following delays:

***Delay** 1:* The amount of time that that can separate the tagging instant and the consumption of a given data by each task τ_{src_c}. This delay concerns the segment

$$\tau_{src} \rightarrow \tau_{src_c}, \forall c \in [1, p]$$

and we call it the *spindle source data waiting time* and denote it by swt_c. Accordingly, the minimum and the maximum of this parameter are denoted by swt_c^{min} and swt_c^{max}, respectively.

***Delay** 2:* The propagation delay by the tasks belonging to the segment

$$\tau_{src_c} \rightarrow \tau_i \rightarrow \tau_{i+1} \cdots \tau_n$$

***Delay** 3:* The execution completion of τ_{trm_c}, which is the time an output data is written into the spindle chain terminus buffer belonging to the c^{th} propagation chain. This is the segment

$$\tau_{trm_c} \rightarrow \tau_{trm}, \forall c \in [1, q]$$

The maximum delay is found by summing the maximum delays while the minimum delay is found by summing the minimum delays for each segment, with respect to the c^{th} chain.

a) ***Computing the** Delay 1:* The swt_c^{min} can be not less than two processor cycles. This corresponds to the case where the time separating the completion of the job of τ_{src} and the tagging moment by the job of the lwp task is only one cycle. The other cycle is spent between the completion of the lwp task and the reading by a job of τ_{src_c}. Since this value is very short, we round it to zero.

The swt_c^{max}, for its part, is the time that can be used by $n = \lfloor \frac{SCI}{T_{src_c}} \rfloor$ jobs of τ_{src_c} that can be released and complete their executions within the time interval bounded by the **SCI** plus the largest interference that can be induced by the higher priority tasks than τ_{src_c}. This interference corresponds to the time at which the $(n+1)^{th}$ job of τ_{src_c} (the last job) can start reading.

b) ***Computing the** Delay 2:* Within the concerned propagation chain, the maximum delay is found if two consecutive execution completions of the producer task must happen as early as possible and as late as possible, respectively. On the contrarily, the minimum value is found when these completions happen as late as possible and as early as possible, respectively. So, for all two adjacent tasks τ_i and τ_j such that τ_i produces input for τ_j, the maximum delay a data sample produced by a job of τ_i can be available for the the job of τ_j is utmost $2T_i - C_i$ [10] while the shortest time this data can be available is equal to c_i. On the other hand, the number of jobs of τ_j that can be released between two consecutive releases of τ_i is denoted by ω_i^j and computed as

$$\omega_i^j = \begin{cases} \lceil \frac{T_j}{T_i} \rceil, & if \; T_i \leq T_j \\ \lfloor \frac{T_j}{T_i} \rfloor + 1, & if \; T_i > T_j \end{cases}$$

So for the segment $\tau_{src_c} \rightarrow \tau_i \rightarrow \tau_{i+1} \cdots \tau_n$, the maximum delay is computed as

$$\omega_{src_c}^i (2T_{src_c} - C_{src_c}) + \cdots + \omega_{n-1}^n (2T_{n-1} - C_{n-1}) \quad (7)$$

and the minimum propagation delays are

$$\omega_{src_c}^i c_{src_c} + \cdots + \omega_{n-1}^n c_{n-1} \quad (8)$$

*c) **Computing the** Delay 3:* The minimum completion time is found when the task executes with no interference; which is equal to its worst-case execution time, while the late moment of completion can happen at the end of the task period. In result,

$$\begin{cases} \mathcal{D}^{max}_{c:q} = swt^{max}_{c} + \max\limits_{1\leq c\leq q}(Delay\ 2) + T_{trm_c} \\ \mathcal{D}^{min}_{c:q} = swt^{min}_{c} + \min\limits_{1\leq c\leq q}(Delay\ 2) + c_{trm_c} \end{cases} \quad (9)$$

*4) **Computing the spindle chain terminus buffers size**:* Knowing the smallest and the largest propagation delays allows us to compute the size of each of the spindle terminus buffers. To that end, we compute the ceil of the largest time it can take for a data to propagate from the spindle source task until the spindle terminus task over the smallest propagation delay of the c^{th} propagation chain, $\forall c \in [1, n]$.

***Theorem** 2:* Let $\beta^{m_c}_{c:q}$ be the spindle source terminus buffer belonging to the chain of index c,

$$|\beta^{m_c}_{c:q}| = \left\lceil \frac{\max\limits_{1\leq c\leq q} \mathcal{D}^{max}_{c:q}}{\mathcal{D}^{min}_{c:q}} \right\rceil \quad (10)$$

***Proof** 5:* The idea here is to compute the sizes ensuring that by the time a job of the spindle terminus task is released it must find, in each of the spindle chain terminus buffers, at-least a single data sample resulting from the execution step of the spindle chain source task and that, without any kind of synchronization mechanism. The Equality 10 has the meaning of the maximum number of times the data samples propagating along the chain of index c can reach the spindle terminus task while the slowest chain (the one having the largest propagation delay) still hasn't output at-least once. Accordingly, the size we consider for each of the spindle terminus buffers is the one willing to store (without overwriting) all the data samples produced by τ_{trm_c} within a time interval bounded by $\max\limits_{1\leq c\leq q} \mathcal{D}^{max}_{c:q}$.

*5) **The temporal alignment and matching algorithm**:* This algorithm implements the proposed spindle-single source data matching solution. All the notations used are previously defined in the paper except $prior, \mathcal{B}^{In,min}_{prior}$ and β^{src} which stand for the index of the prior task, the set of input buffers for $prior$ and the spindle source buffer, respectively. Scheduling starts at the $Statement\ 4$ (further ***Stt.***). During the scheduling time, use the method `get_Prior()` which returns the prior task (***Stt. 5***). At the release of $prior$, it checks all its input buffers (***Stt. 6***). For all of these buffers, $prior$ needs to retrieve the $tail$ value which indicates from which slot of the buffer to read. If the current buffer is a spindle source one, $tail$ is the one of lwp; the lower priority tasks among readers (***Stt. 7-8***).

If $prior$ is a spindle terminus task, then it has to read from each of the spindle chain terminus buffers. To that end, for a correct matching of the data, the method `get_Timestamp` is used to return the timestamp of the data being into the smallest (the one belonging to the chain with the largest propagation delay; its size is one). Further, it retrieves from each of the remained buffers, the data sample whose timestamp is equal to the one returned by `get_Timestamp()` at ***Stt. 10-12***. Finally, if the current buffer is not a spindle source one and $prior$ is not the spindle terminus task, then, we use the classical approach where each task owns a $tail$ pointer (***Stt. 14***). Once the correct value of $tail$ pointer is found, $prior$ reads from the slot of β pointed by the current value of $tail$ by calling the method `readFrom()` (***Stt. 17***).

Algorithm 1 The temporal alignment and matching algorithm

```
1:  Require G = (T, M, E^M), H, τ_src, lwp, τ_trm, prior
2:  Require β^src, B^{In,min}_prior
3:  int tail = head = 0, t_stamp
4:  while schedule(T, H) == true do
5:      prior=get_Prior()
6:      for each β ∈ B^{In}_prior do
7:          if β ≡ β^src then
8:              tail ← tail(lwp, β)
9:          else
10:             if prior ≡ τ_trm then
11:                 t_stamp ← get_Timestamp(B^{In,min}_prior)
12:                 tail ← get_Tail(β, t_stamp)
13:             else
14:                 tail ← tail(prior, β)
15:             end if
16:         end if
17:         readFrom(prior, β, tail)
18:     end for
19: end while
```

VI. Conclusions

In this paper we propose a full framework ensuring the data freshness, consistency and temporal matching in the context of the single-source data matching problem. Our solution is achieved asynchronously without any kind of arbitration mechanism (blocking semaphores and protocols). Considering that predictability is a mandatory property for real-time systems, our solution responds perfectly to this requirement unlike using arbitration mechanisms which, while trying to protect the shared resource, induce temporal uncertainty with possibility of leading to undesirable situations like priority inversion problem or dead-lock formations. Further, we formally show how to bound the size of all the categories of buffers considered herein (spindle source, spindle chain terminus buffer and inner buffers). The proposed results will serve as the internal communication model within the the PX4-RT drone autopilot being prepared within the KOPERNIC research team at INRIA Paris. In the future works, we aim to address the issues regarding the data matching for the α- and β- spindles.

References

[1] C. L. Liu and James W. Layland, "Scheduling Algorithms for Multprirorogramming in a Hard-Real-Time Environment" Journal ACM, 1973.

[2] J. Lehoczky, L. Sha, Y. Ding, "The rate monotonic scheduling algorithm: Extact characterization and average case behavior", Real-Time Systems Symposium (1989) 166–171.

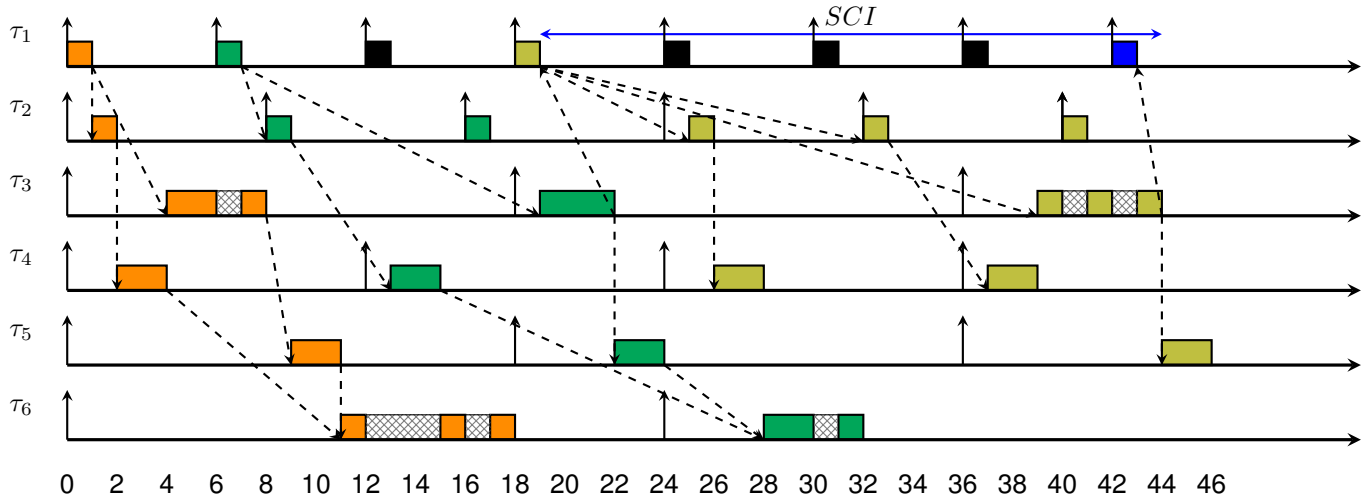

Figure 5: Implementation results while considering the model presented on the Figure 2 where $\tau_1(1,6), \tau_2(1,8), \tau_3(3,18), \tau_4(2,12), \tau_5(2,18)$ and $\tau_6(3,24)$. We assume the best- and the worst-case execution times are equal.

[3] M. Joseph, P. Pandya, Finding response times in a realtime system, BCS Computer Journal 29 (5) (1986) 390–395.

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

[5] T.Kloda,A. Bertout and Y. Sorel, "Latency analysis for data chains of real-time periodic tasks.", ETFA, 2018.

[6] J. Schlatow and R. Ernst, "Response-Time Analysis for Task Chains in Communicating Threads.", 2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), ETFA, 2016.

[7] H. Zeng and Marco Di Natale, "Mechanisms for guaranteeing data consistency and flow preservation in AUTOSAR software on multi-core platforms.",SIES 2011.

[8] Bruno Steux, "RTMAPS, un environnement logiciel dédié à la conception d'applications embarqués tems-réel. Utilisation pour la détection automatique de véhicules par fusion radar/Vision.", PhD paper, Ecole des mines de Paris, France,2001.

[9] Joseph Y.-T. Leung and Jennifer Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks" Journal Perform. Evaluation, 1982.

[10] M. Becker,D. Dasari, S. Mubeena, M. Behnam and Thomas Nolte, "Synthesizing Job-Level Dependencies for Automotive Multi-Rate Effect Chains.", RTCSA, At Daegu, South Korea, 2016.

[11] Alix Munier Kordon and Ning Tang, "Evaluation of the Age Latency of a Real-Time Communicating System Using the LET Paradigm.", ECRTS, 2020.

[12] N. Pontisso and P. Quéinnec and G. Padiou, "Analysis of distributed multi-periodic systems to achieve consistent data matching.", 2013.

[13] J. Forget, F. Boniol and C. Pagetti, "Verifying end-to-end real-time constraints on multi-periodic models.", ETFA 2017, Limassol, Cyprus, 2017.

[14] L. Meier, D. Honegger, and M. Pollefeys, "PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms", ICRA 2015, Seattle, WA, USA, 26-30 May, 2015.

[15] J. Chen and A. Burns, "Loop-free asynchronous data sharing in multpriorocessor real-time systems based on timing properties.", RTCSA, 1999.

[16] H. Kopetz and J. Reisinger, "The non-blocking write protocol NBW: A solution to a real-time synchronization problem.",RTSS, 1993.

[17] N. Feiertag, K. Richter, J. Nordlander and J. Jonsson, "A Compositional Frameworfor End-to-End chain Delay Calculation of Automotive Systems under Different chain Semantics.", journal CRTS, 2008.

[18] E. Ntaryamira, C. Maxim and Liliana Cucu-Grosjean, "*data consistency* and temporal validity under the circular buffer communication paradigm.", RACS 2019, Chongqing, China, September 24-27, 2019.

[19] Liliana Cucu-Grosjean and Joël Goossens, "Exact schedulability tests for realtime scheduling of periodic tasks on unrelated multpriorocessor platforms", Advances in Real-Time Systems, 2012.

[20] C. M. Kirsch and A. Sokolova, "The Logical Execution Time Paradigm" IEEE International Conference on Robotics and Automation, ICRA 2015, Seattle, WA, USA, 26-30 May, 2015.

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

[22] B. Steux and C. Laurgeau and L. Salesse and D. Wautier, "Fade: a vehicle detection and tracking system featuring monocular color vision and radar data fusion", Intelligent Vehicle Symposium, 2002. IEEE.