Abstract—Modern multiprocessor streaming systems have hard real-time constraints that must be always met to ensure correct functionality. At the same time, these streaming systems must be designed to use the minimum required amount of resources (such as processors and memory). In order to meet such constraints, using scheduling algorithms from the classical real-time scheduling theory represents an attractive solution approach. These algorithms enable: (1) providing timing guarantees to the applications running on the system, and (2) deriving analytically the minimum number of processors required to schedule the applications. So far, designers in the embedded systems community have focused on global and partitioned scheduling algorithms. However, recently, a new hybrid class of scheduling algorithms has been proposed. In this work, we investigate the applicability of a sub-class of these hybrid algorithms, called semi-partitioned algorithms, to applications modeled as Cyclo-Static Dataflow (CSDF) graphs. The contribution of this paper is two fold. First, we devise an approach that enables semi-partitioned scheduling algorithms, even soft real-time ones, to be applied to CSDF graphs while providing hard real-time guarantees at the input/output interfaces with the external environment. Second, we focus on an existing soft real-time semi-partitioned approach, for which we propose an allocation heuristic, called FFD-SP. The proposed heuristic reduces the minimum number of processors required to schedule the applications compared to a pure partitioned scheduling algorithm, while trying to minimize the buffer size and latency increases incurred by the soft real-time approach.

I. INTRODUCTION

The design of modern embedded systems is a difficult task due to the complex functionalities that have to be implemented and the strict constraints that must be met. In order to implement these complex functionalities, Multiprocessor Systems-on-Chip (MPSoCs) are nowadays the standard choice for system designers. Realizing such complex functionalities on MPSoCs entails two challenges. The first challenge is expressing the parallelism in the applications in order to efficiently exploit the multiple processors found in MPSoCs. The common practice for expressing the parallelism in an application is to use Models of Computation (MoCs) [1]. Several MoCs have been proposed such as Synchronous Dataflow (SDF) [2] and its generalization, Cyclo-static Dataflow (CSDF) [3].

The second challenge is how to allocate and schedule the applications’ tasks on the system in such a way that all the timing constraints are guaranteed to be met. Many algorithms have been proposed in the classical real-time scheduling theory to deal with the problem of scheduling applications on multiprocessor systems [4]. These algorithms assume that the application tests satisfy a certain real-time task model which allows them to analytically reason about the timing behavior of the tasks. Several real-time task models have been proposed such as the periodic task model and the sporadic task model. Recent works (such as [5], [6]) have shown that applications modeled as CSDF graphs can be scheduled as a set of real-time periodic tasks. This provides two main benefits. The first benefit is that the designer can apply hard real-time (HRT) scheduling algorithms [4] to streaming applications in order to guarantee timing constraints and temporal isolation among different applications. The second benefit is that applications can be loaded at run-time on the system, provided that the appropriate schedulability tests are satisfied. The benefits mentioned above are advantages of [5], [6] over conventional static scheduling.

So far, the approaches proposed in [5], [6] consider only hard real-time global or partitioned scheduling algorithms. Under global scheduling algorithms, all the tasks can migrate among all the processors. Such algorithms can be optimal for multiprocessor systems, which means that they can fully exploit the available computational resources (see for instance [7]). However, this comes at the cost of high scheduling overheads due to excessive task preemptions and migrations. Moreover, modern MPSoCs systems that ensure predictability of execution are commonly implemented as distributed memory systems in order to avoid the unpredictability of accessing shared resources. Therefore, implementing a global scheduler on such distributed memory systems implies that the code of each task has to be replicated1 on all the processors, incurring a large memory overhead. Partitioned scheduling algorithms, in contrast, incur neither migration overhead nor memory overhead because each task is statically allocated to a single processor. However, these algorithms are affected by bin-packing issues [9]. If no limit on the maximum task utilization of a task set is imposed, partitioned algorithms may require almost twice as many processors compared to an optimal scheduler [10].

Recently, a third class of algorithms, called hybrid scheduling algorithms, has been proposed. Among the hybrid scheduling algorithms, semi-partitioned algorithms (e.g., [11]) seem to be a good match for distributed memory systems. Under semi-partitioned algorithms, most of the tasks are statically allocated to processors, and only a subset of the tasks is allowed to migrate among different processors. Migrating tasks follow a migration pattern derived at design-time. Thus, semi-partitioned approaches represent a “middle ground” between partitioned and global scheduling algorithms. In fact, semi-partitioned scheduling algorithms can overcome (or mitigate) the bin-packing effects that affect partitioned approaches. As a result, semi-partitioned algorithms require less processors than partitioned algorithms to schedule certain task sets. At the same time, these algorithms do not incur large memory overheads and task migration/preemption overheads like global algorithms.

Several semi-partitioned scheduling algorithms have been proposed [11]–[15]. These algorithms can be classified based on when migrations are allowed to occur. In restricted-migration approaches [11]–[13] migrations can happen at job boundaries only. In unrestricted-migration (or partitioned) approaches, migrations can happen at any time during a job execution. We argue that the restricted-migration class of semi-partitioned schedulers is the most suitable for distributed memory MPSoCs. This is because migrating at job boundaries reduces the amount of data (state) to be transferred from one processor to the next. Moreover, if the task

1We assume task migration using code replication, as in [8], because of its low overhead in distributed memory MPSoC systems.
does not keep an internal state between two successive jobs\(^2\), no state migration is needed. Within the class of restricted-migration semi-partitioned approaches, EDF-fm [11] is particularly suited to distributed memory systems because in that approach a migrating task is allowed to migrate only between two processors (contrary to [12], [13]), in which migrating tasks may span among all the processors). This property reduces substantially the overhead caused by replicating the task code. In addition, EDF-fm uses a fast utilization-based schedulability test (contrary to [12], [13]), that can be easily executed at run-time for incoming applications. For the reasons explained above we employ EDF-fm in our work.

Although EDF-fm can have great benefits for distributed memory MPSoCs, it provides hard real-time guarantees only for migrating tasks and soft real-time (SRT) guarantees for fixed tasks. This means that fixed tasks can miss their deadlines by a bounded value called tardiness. As a consequence, the proposed scheduling frameworks in [5], [6] can not be used directly with such an algorithm, since these frameworks assume that all task deadlines are met. Therefore, in this paper, we extend the framework proposed in [5] to support soft real-time scheduling algorithms while providing hard real-time guarantees on the input/output interfaces of the application with the external environment. This is illustrated in Fig. 1 which shows the difference between using HRT and SRT schedulers. Given an application, we want to provide hard real-time guarantees to I and O, which denote the external provider/consumer of the data streams. I and O are assumed to execute periodically. I delivers the data into an input buffer \(b_{in}\), from which the application reads the data and processes it. After that, the application writes the processed data into an output buffer \(b_{out}\), which is read by the external data consumer O. The framework in [5] shows that the application can be scheduled under a partitioned HRT scheduling algorithm as a set of real-time periodic tasks by computing sufficient minimum buffer sizes between application tasks. In this paper, we show that the same can be done under semi-partitioned SRT schedulers, achieving the same throughput and using less processors, albeit requiring larger buffers and increased application latency. This increase in the buffering requirements is visualized in Fig. 1(b) using red color. Therefore, under both schemes (i.e., Fig. 1(a) and 1(b)), the external data consumer O will always find enough data in \(b_{out}\) (and I enough space in \(b_{in}\)), regardless of the fact that \(v_{sink}\) and \(v_{sink}\) might miss deadlines under the SRT scheduler.

A. Paper Contributions

Given an implicit-deadline periodic task set representation of a CSDF graph and any soft real-time scheduling algorithm that ensures a bounded tardiness to each task, we derive the earliest task start times and minimum buffer sizes that guarantee the existence of a valid schedule of the given application. Valid schedule means that, even in presence of task tardiness, tasks can be released periodically and neither buffer underflow nor overflow can occur. We call this approach tardiness-aware periodic scheduling. Moreover, we guarantee that the interfaces of the environment with the application (see I and O in Fig. 1(b)) can execute in a strictly periodic way with neither underflow nor overflow on input and output buffers (see \(b_{in}\) and \(b_{out}\) in Fig. 1(b)), i.e., we provide HRT guarantees for the input/output interfaces.

Then, using the above result, we focus on a specific restricted-migration semi-partitioned scheduling algorithm, namely EDF-fm [11], with the goal of reducing the number of required processors compared to partitioned approaches. We propose a novel heuristic, called FFD-SP, to assign tasks to processors while taking data dependencies into account. This heuristic replaces the ones proposed in [11] that are intended for independent task sets. Our proposed heuristic is aimed at reducing the number of required processors while keeping a lower buffer size and latency overhead when the EDF-fm algorithm is used.

Finally, we show on a set of real-life benchmarks that our approach can lead to significant benefits by reducing the number of processors required to schedule a given application, compared to a partitioned approach, while maintaining the same throughput. Among the used benchmarks, we show that this is true for all the applications that suffer from the bin-packing effects of partitioned algorithms. Moreover, our experiments show that the increase in memory requirements and application latency introduced by our approach is acceptable (on average +24% and +29%, respectively), especially for systems in which the throughput constraint is more important than memory or latency constraints.

II. RELATED WORK

To the best of our knowledge, real-time semi-partitioned scheduling algorithms have never been studied when mapping streaming applications with inter-task data dependencies to MPSoCs. In fact, existing semi-partitioned solutions [11]–[15] only consider sets of independent tasks. In the real-time community, however, techniques different from pure partitioning to assign data-dependent application tasks to a multiprocessor platform have already been devised. Existing approaches which are close to our work are [16] and [17] by Liu and Anderson. These approaches use a global scheduler which, similar to our case, satisfies soft real-time requirements. In particular, [16] describes a way to guarantee bounded tardiness of an application specified as a pipeline of tasks under a SRT global scheduler. A strong limitation in [16] is that only simple pipeline application topologies are handled, contrary to our approach that can handle more complex topologies like CSDF graphs. In [17], they extend their analysis to guarantee bounded task tardiness in more complex application graph topologies, such as Processing Graph Method (PGM) graphs. However, PGM graphs are still less expressive than CSDF graphs, which are supported in our approach. Moreover, the work in [17] does not address the calculation of minimum buffer sizes, which is an important metric to evaluate the practicability of the approach. In contrast, the calculation of buffer sizes is supported by our approach.

III. BACKGROUND

In Sec. III-A we introduce the system model considered in our work. Then, we summarize two techniques instrumental to our approach: strictly periodic scheduling of acyclic CSDF graphs (Sec. III-B) and the EDF-fm semi-partitioned scheduling algorithm (Sec. III-C).

A. System Model

We consider a system composed of a set \(\Pi = \{\pi_1, \pi_2, \ldots, \pi_m\}\) of \(m\) homogeneous processors. The processors execute a task set \(\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}\) of \(n\) periodic tasks, which can be preempted at any time. A periodic task \(\tau_i \in \Gamma\) is defined by a 4-tuple \(\tau_i = (C_i, T_i, S_i, \Delta_i)\), where \(C_i\) is the worst-case execution time of the task, \(T_i\) is the task period, \(S_i\) is the start time of the task, and \(\Delta_i\) represents the task tardiness bound, as defined in Definition 1 below. In this paper we consider only implicit-deadline tasks, which have relative deadline \(D_i\) equal to their period \(T_i\). The utilization of a task \(\tau_i\) is given by \(u(\tau_i) = C_i/T_i\) (also denoted

\(^2\)Tasks that possess this property are called stateless and are common in streaming applications.
as \( u_i \). Given a set \( \gamma \) of tasks mapped onto processor \( \pi_k \), the total assigned utilization on \( \pi_k \) is denoted with \( U(\pi_k) = \sum_{\tau_i \in \gamma} u(\tau_i) \).

The \( k \)th job of task \( \tau_i \) is denoted with \( \tau_{i,k} \). Job \( \tau_{i,k} \) of \( \tau_i \) for all \( k \in \mathbb{N}_0 \), arrives in the system at the time instant \( r_{i,k} = S_i + kT \). The absolute deadline of job \( \tau_{i,k} \) is \( d_{i,k} = S_i + (k + 1)T \), which is coincident with the arrival of job \( \tau_{i,k+1} \). We denote the actual completion time of \( \tau_{i,k} \) with \( f_{i,k} \). In SRT systems, tasks are allowed to miss their deadline by a certain bounded value, as defined below.

**Definition 1** (Tardiness bound). A task \( \tau_i \) is said to have a tardiness bound \( \Delta_i \) if \( f_{i,k} \leq (d_{i,k} + \Delta_i) \), \( \forall k \in \mathbb{N}_0 \).

**B. Strictly Periodic Scheduling of CSDF Graphs**

A CSDF graph [3] is composed of a set of actors \( V \) and a set of edges \( E \), through which actors communicate. The authors in [5] show that the actors in an acyclic CSDF graph can be scheduled as a set of real-time periodic tasks. An example of this is given in Fig. 2. Based on the properties of the graph, they derive the minimum period of each actor using the following expression:

\[
T_i = \frac{Q}{\eta} \left( \frac{r}{\eta} \right) + 1
\]

where \( q_i \) is the number of repetitions of actor \( \tau_i \) per graph iteration, \( C_i \) is the worst-case execution time of \( \tau_i \), \( \eta = \max_{v \in V} |C_i| \) and \( Q = \text{lcm}(q_1, q_2, \ldots, q_n) \). Consider the CSDF graph shown in Fig. 2(a). Its repetition vector is \( \eta = \{3, 2, 3\} \). Its worst-case execution time vector is \( \overrightarrow{C} = \{1, 2, 2\} \). Then, it follows that \( \eta = 6 \) and \( \overrightarrow{T} = \{2, 3, 2\} \). In general, the derived period vector \( \overrightarrow{T} \) satisfies the condition:

\[
q_1T_1 = q_2T_2 = \cdots = q_nT_n = \alpha
\]

where \( \alpha \) is defined as iteration period, and represents the duration needed by the graph to complete one iteration.

Then, the authors in [5] define the following functions:

**Definition 2** (Cumulative Production Function). The cumulative production function of actor \( \tau_i \) producing into channel \( e_u \) during a time interval \( [t_s, t_e] \), denoted by \( \text{prd}_{([t_s, t_e])}(\tau_i, e_u) \), is the sum of the number of tokens produced by \( \tau_i \) into \( e_u \) during the interval \( [t_s, t_e] \).

**Definition 3** (Cumulative Consumption Function). The cumulative consumption function of actor \( \tau_i \) consuming from channel \( e_u \) over a time interval \( [t_s, t_e] \), denoted by \( \text{cns}_{([t_s, t_e])}(\tau_i, e_u) \), is the sum of the number of tokens consumed by \( \tau_i \) from \( e_u \) during the interval \( [t_s, t_e] \).

Note that the time interval in Definitions 2 and 3 can be either open or closed from the right. Using the functions defined in Definitions 2 and 3, the authors derive the earliest start times of actors and minimum buffer sizes of the channels, as described in the following.

1) **Earliest start times**: The earliest start time of actor \( \tau_j \), denoted with \( S_j \), is derived using the following expression (according to Lemma 3 in [5] and assuming no task tardiness):

\[
S_j = \begin{cases} 
0 & \text{if } \text{prec}(\tau_j) = \emptyset \\
\max_{v \in \text{prec}(\tau_j)} (S_i + \tau_j) & \text{if } \text{prec}(\tau_j) \neq \emptyset
\end{cases}
\]

where

\[
S_i \rightarrow j = \min_{t \in [0, S_i + \alpha]} \left\{ t : \frac{\text{prd}_{[S_i, \text{max}(S_i, t) + k]}(\tau_i, e_u)}{\text{cns}_{[S_i, \text{max}(S_i, t) + k]}(\tau_j, e_u)} \geq \frac{\text{max}}{\text{min}} \right\}
\]

where \( \alpha \) is the iteration period being defined by Equation (2), and \( S_i \) is the start time of the predecessor actor \( \tau_i \). Equation (4) considers the dependency between predecessor actor \( \tau_i \) and successor actor \( \tau_j \), over channel \( e_u \). It computes the earliest start time \( S_i \rightarrow j \) such that \( \tau_j \), in its periodic execution, is never blocked on read. This is ensured by checking that at each time instant the cumulative number of tokens produced by \( \tau_i \) over \( e_u \) is greater than or equal to the number of tokens consumed by \( \tau_j \) from the same channel. For instance, actor \( \tau_2 \) in Fig. 2(b) must start no earlier than time \( t = 3 \) to satisfy its data dependency from actor \( \tau_1 \).

Start times \( S_i \rightarrow j \) are computed for each actor in the predecessor set of \( \tau_j \) (denoted with \( \text{prec}(\tau_j) \)). Then, when actor \( \tau_j \) has several predecessors, the start time \( S_j \) has to be set to the maximum of the start times \( S_i \rightarrow j \) considering each predecessor in isolation, as captured by Equation (3).

2) **Minimum buffer sizes**: Once the start time of actors have been calculated, the authors derive the following expression for the minimum size \( b_u \) of communication channel \( e_u \) connecting actors \( \tau_i \) and \( \tau_j \) (see Lemma 4 in [5]):

\[
b_u(\tau_i, \tau_j) = \max_{k \in [0, 1, \ldots, \alpha]} \left\{ \text{max} \left( \frac{\text{prd}_{[S_i, \text{max}(S_i, t) + k]}(\tau_i, e_u)}{(t_i, e_u)} \right) \right\}
\]

In other terms, Equation (5) evaluates the maximum number of unconsumed tokens in \( e_u \) during one iteration of \( \tau_i \) and \( \tau_j \).

**C. EDF-fm Semi-partitioned Scheduling**

Anderson et al. in [11] describe the EDF-fm semi-partitioned scheduling algorithm for sporadic tasks sets in the context of soft real-time multiprocessor systems. Under EDF-fm tasks can be either fixed or migrating. Migrating tasks are allowed to migrate between only two processors, with the restriction that migration can only happen at job boundaries.

The EDF-fm approach consists of two phases: the assignment phase and the execution phase, which are summarized in the following sections.

1) **Assignment phase**: Consider the following definitions:

**Definition 4** (Task share). A task \( \tau_i \) is said to have a share \( s_{i,k} \) on \( \pi_k \) when a fraction \( s_{i,k} \) of \( \pi_k \)'s available utilization is allocated to \( \tau_i \).

**Definition 5** (Task fraction). Given a share \( s_{i,k} \), \( \pi_k \) executes a fraction \( \varphi_{i,k} = s_{i,k} \cdot \alpha \) of \( \tau_i \)'s total execution requirement.

In the assignment phase each task is assigned to either one processor (fixed task) or two processors (migrating task). In particular, the assignment phase assigns tasks in sequence to processors. Tasks are assigned to a processor \( \pi_k \) until its capacity is exhausted. If a task \( \tau \) cannot entirely fit onto processor \( \pi_k \), then a share \( s_{i,k} = 1 - U(\tau|\pi_k) \) of its utilization is assigned to \( \pi_k \) and the remaining utilization \( s_{i,k+1} = (s_i - s_{i,k}) \) is assigned to the next processor, \( \pi_{k+1} \). The assignment phase of EDF-fm ensures that at most two migrating tasks are assigned to any processor.

**Example 1**. Given the task set \( \{\tau_1 = (C_1 = 3, T_1 = 10), \tau_2 = (2, 5), \tau_3 = (2, 5), \tau_4 = (1, 2), \tau_5 = (1, 2), \tau_6 = (2, 5), \tau_7 = (1, 2)\} \), the EDF-fm algorithm derives the task assignment shown in Fig. 3. For instance, task \( \tau_3 \) in Fig. 3 cannot entirely fit onto
π₁, thus its utilization is split between π₁ and π₂ with shares s₁₁ = 3/10 and s₁₂ = 1/10, respectively.

2) Execution phase: The execution phase employs a simple online scheduling algorithm that is derived from EDF and ensures bounded tardiness with a minimal overhead compared to a canonical EDF scheduler. Jobs belonging to a task τᵢ, which migrates between processors πₖ and πₖ₊₁, are assigned such that in the long run the fraction of τᵢ’s workload executed on πₖ is close to ϕᵢ,k. For instance, in the scenario depicted in Fig. 3, on average 1 out of 4 jobs of τ₃ are assigned to π₂ and the remaining 3 jobs are assigned to π₁. When two migrating tasks, τᵢ and τⱼ, are assigned to πₖ, the tardiness bound for a fixed task τₐ assigned to the same processor is given by:

\[ \Delta(\tauₐ) = \frac{C_j \cdot (\phi_j,k + 1) + C_i \cdot (\phi_i,k + 1) - T_u \cdot (1 - U(\pi_k))}{1 - s_i,k - s_j,k} \]  

(6)

where Cᵢ and Cⱼ are the worst-case execution times of τᵢ and τⱼ (as defined in Sec. III-A), T_u is the period of task τₐ, and U(πₖ) is the sum of fixed tasks’ utilization and migrating tasks’ shares allocated to πₖ. By contrast, migrating tasks do not miss any deadline, therefore their tardiness bound is zero.

On a processor running two migrating tasks, τᵢ and τⱼ, EDF-fm requires that the sum of their utilization does not exceed one:

\[ U_{\text{max}}(\pi_k) = u_i + u_j \leq 1 \]  

(7)

This condition is automatically satisfied if the maximum utilization of any task is limited to 1/2, given the fact that at most two migrating tasks can be assigned to a single processor. However, tasks that exceed this utilization limit can still be considered, provided that condition (7) is valid for all the processors.

IV. TARDINESS-AWARE PERIODIC SCHEDULING

In this section, we present the first main contribution of this paper. We show how a CSDF graph, represented by a set of periodic tasks derived using the approach in [5] (see Sec. III-B), can be scheduled using soft real-time schedulers, i.e., schedulers that may introduce a bounded tardiness on the completion of tasks. The SRT scheduler considered in this paper is the EDF-fm algorithm, whose per-task tardiness bound is given by Equation (6). Note, however, that the results obtained in this section are valid for any SRT scheduler which provides bounded task tardiness. Our solution extends the framework in [5] by deriving earliest start times for each task (see Sec. IV-A) and minimum buffer sizes (see Sec. IV-B) that can handle task tardiness and still allow a periodic release of each task.

A. Earliest Start Times in Presence of Tardiness

Based on Definitions 1, 2, and 3 introduced in Sec. III, we give the following Lemma:

Lemma 1. In presence of task tardiness, bounded by Δᵢ for source actor vᵢ and by Δⱼ for destination actor vⱼ, the earliest start time Sᵢ→ⱼ of actor vⱼ due to its dependency from vᵢ, under a valid schedule, is given by:

\[ Sᵢ→ⱼ = \min_{t \in [0,Sᵢ+Δᵢ+\alpha]} \{ \text{t : prd :} \} \]

\[ \{ vᵢ, eᵢ \} \geq \text{cns} \{ vⱼ, eⱼ \} \]

\[ \forall k = 0, 1, \ldots, \alpha \} \]  

(8)

Proof: If actors vᵢ and vⱼ may be affected by tardiness, Equation (4) cannot be applied in its original form. For instance, if job v₁₂ in Fig. 2(b) completes later than its deadline, job v₂₁ (that depends on the completion of v₁₂) cannot be released at time t = 6. It follows that the start time of actor v₂ has to be changed.

We can define a worst-case scenario of the execution of vᵢ and vⱼ to derive the earliest start time Sᵢ→ⱼ in case of tardiness. This worst-case scenario to determine task start times occurs when: (i) all the invocations of the source actor vᵢ are completed with the maximum tardiness; (ii) none of the invocations of the destination actor vⱼ are affected by the tardiness (i.e., vⱼ meets all its deadlines). In the worst-case execution of vᵢ, the completion of all its invocations are delayed by the maximum considered tardiness Δᵢ, such that fᵢ,k = dᵢ,k + Δᵢ, for all k. We can associate these worst-case delayed invocations to a fictitious task ˜vᵢ. Task ˜vᵢ has start time Sᵢ = Sᵢ + Δᵢ, constant period Tᵢ = Tᵢ, and no tardiness. Notice that invocations ˜vᵢ,k of ˜vᵢ are strictly periodic. Fig. 4 shows the schedule of actor vᵢ and its worst-case execution scenario, represented by ˜vᵢ.

By contrast, in the worst-case scenario to determine Sᵢ→ⱼ, vⱼ is executed as early as possible, so we assume that all invocations vⱼ,k of vⱼ are not affected by tardiness. Then, the earliest start time that guarantees the absence of blocking of vⱼ in its execution, even for the worst-case production and consumption patterns of vᵢ and vⱼ, is found by evaluating Equation (4) with vᵢ as source actor and vⱼ as destination actor. This scenario is captured by Equation (8). Note that any completion of an invocation vᵢ,k of vᵢ earlier than its corresponding worst-case ˜vᵢ,k results in an earlier production of tokens, such that the inequality in Equation (8) still holds for all k ∈ [0,1,⋯,α]. Similarly, if any of the invocations of vⱼ is affected by tardiness, the token consumption is executed later and Equation (8) guarantees that enough tokens will be available to be read.

Note also that the start time Sᵢ→ⱼ of actor vⱼ due to its dependency from vᵢ is only affected by the tardiness bound Δᵢ of the source actor. In addition, when actor vⱼ has several predecessors, the start time Sⱼ has to be set to the maximum of the start times Sᵢ→ⱼ given by Equation (8) considering each predecessor in isolation, as captured by Equation (3).

B. Minimum Buffer Sizes in Presence of Tardiness

Based on Definitions 1, 2, and 3 introduced in Sec. III, and given the actor start times calculated leveraging Lemma 1, the following lemma provides a way to derive minimum buffer sizes in case of task tardiness:

Lemma 2. In presence of task tardiness, bounded by Δᵢ for source actor vᵢ and by Δⱼ for destination actor vⱼ, the minimum buffer size bᵢ of a communication channel eᵢ connecting vᵢ and vⱼ, under a valid schedule, is given by:

\[ bᵢ(vᵢ,vⱼ) = \max_{k \in [0,1,\ldots,\alpha]} \{ \text{prd :} \} \]

\[ \{ Sᵢ + \Deltaᵢ, \text{max}(Sᵢ + \Deltaᵢ,t) + k \} \]

\[ \{ vᵢ, eᵢ \} \geq \{ vⱼ, eⱼ \} \]

\[ \forall k = 0, 1, \ldots, \alpha \} \]  

(9)

Proof: To get the minimum buffer size in presence of task tardiness, we consider the worst-case scenario that would result in the maximum buffer requirement for channel eᵢ. This worst-case scenario occurs when: (i) all the invocations vⱼ,k of the destination actor vⱼ complete with the maximum tardiness (such that fⱼ,k =
Algorithm 1: FFD-SP task assignment heuristic.

Input: The number of processors $m$, a task set $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$ of $n$ periodic tasks.

Result: True and an $m$-partition describing the task assignment onto $m$ processors if $\Gamma$ is schedulable, False otherwise.

1. Find $\Gamma_\alpha = \{\tau : \tau \in \Gamma \land \tau$ is stateful};
2. Assign tasks in $\Gamma_\alpha$ using FFD;
3. if $\Gamma_\alpha$ cannot be assigned then
   return False;
4. for $\tau \in (\Gamma - \Gamma_\alpha, \text{sorted in decreasing utilization})$ do
   5. Try to assign task $\tau$ using First-Fit heuristic;
   6. if First-Fit is successful then
      continue;
   7. for $\pi' \in (\Pi$ sorted in decre. available utilization) do
      8. $s_1(\tau) = 1 - U(\pi')$;
      9. if $sp\_assign(s_1(\tau), \pi') == True$ then
         10. $s_2(\tau) = u(\tau) - s_1(\tau)$;
         11. for $\pi'' \in (\Pi$ sorted in incre. available utilization) do
               12. if $sp\_assign(s_2(\tau), \pi'') == True$ then
                  13. Assigned $== True$;
                  14. break;
               15. if Assigned $== False$ then
                  16. Revise assignment of $s_1(\tau)$ to $\pi'$;
                  17. else
                     break;
               18. if Assigned $== False$ then
                  19. return False;
   20. Optimize the obtained partition;
   21. return True;

$\Delta_j,k = \Delta_j,k$. For all $k$); (ii) none of the invocations of the source actor are affected by tardiness.

We can then prove Lemma 2 with a procedure similar to the one used in the proof of Lemma 1. We associate the worst-case completion of all the invocations $v_{j,k}$ to a fictitious actor $v_j$. Actor $v_j$ is strictly periodic, with no tardiness, constant period $T_j$, and start time $S_j = S_j + \Delta_j$. The minimum buffer requirement of the communication channel $e_{u}$ is found by evaluating Equation (5) with $v_i$ as source actor and $v_j$ as destination actor. This scenario is captured in Equation (9).

Note that any earlier completion of any of the iterations of $v_j$ would not increase the buffer size requirement. This is because an earlier completion of $v_j$ would result in an earlier consumption of tokens from channel $e_u$. Similarly, any delayed completion of an iteration of $v_j$ would result in a delayed production of tokens to the considered channel. Thus, the derived value of $b_u$ is sufficient and minimum.

Note that Equations (8) and (9) can also be used to analyze the interfaces between the external data provider and consumer (I and O in Fig. 1(b)) and the source and sink nodes of the application ($v_{src}$, $v_{snk}$). Compared to the HRT approach shown in Fig. 1(a), in the SRT approach of Fig. 1(b) the $v_{src}$ and/or $v_{snk}$ may experience tardiness. In this case, Equation (8) and (9) derive delayed start time of the external consumer O and larger buffer sizes of $b_{in}$ and $b_{out}$ such that both I and O can execute strictly periodically with neither buffer overflow nor underflow occurring on $b_{in}$ and $b_{out}$.

V. FFD-SP TASK ASSIGNMENT HEURISTIC

In this section, we present the second main contribution of this paper. Considering the results (Lemma 1 and Lemma 2) shown in Sec. IV, we propose a task assignment heuristic that does not follow the sequential approach common to all the heuristics proposed in [11]. In fact, as explained in Sec. III-C, the heuristics in [11] assign tasks to processors in a sequential way, which means that in most cases processors are assigned migrating tasks. In turn, this makes most tasks in the system affected by tardiness. Actor tardiness imposes larger buffer sizes (according to Lemma 2) and postponed start times of successor actors (according to Lemma 1). Overall, this leads to larger memory requirements and increased application latency.

In contrast to the heuristics in [11], our proposed heuristic starts to consider semi-partitioning only when the First-fit Decreasing (FFD) heuristic [9] fails to assign a certain task in the system. The proposed allocation/assignment heuristic, called FFD-SP, is described in Algorithm 1. The algorithm accepts as input the number of processors $m$ onto which the task set $\Gamma$ has to be assigned. At the first execution of Algorithm 1, $m$ is set to $m_{opt}$, the number of processors required by an optimal scheduler. If the task set cannot be assigned to $m$ processors, $m$ is increased by one and Algorithm 1 is executed again until a successful assignment is found.

At first, Algorithm 1 builds $\Gamma_\alpha$, the set of stateful actors, which are then assigned using the FFD heuristic (lines 1-4). Stateful actors are considered first because this way they are fixed to a processor and there is no need to migrate their state. Then, considering task $\tau \in (\Gamma - \Gamma_\alpha)$, the algorithm tries to assign task $\tau$ to one of the processors using FFD (lines 6-8). If FFD does not succeed, the algorithm tries to divide the utilization of task $\tau$ in two shares, $s_1(\tau)$ and $s_2(\tau)$. Traversing the processor list in decreasing order of available utilization, a share $s_1(\tau) = 1 - U(\pi')$ is tried to be mapped on processor $\pi'$ (lines 9-12). If the assignment of $s_1(\tau)$ is successful, the algorithm attempts to map a share $s_2(\tau) = u(\tau) - s_1(\tau)$ by traversing the list of processors in increasing order of available utilization (lines 13-17).

The rationale behind this assignment heuristic is two-fold: (1) tasks are semi-partitioned only when the FFD assignment does not succeed; this way the number of processors with migrating tasks (and thus with tardiness) are likely to be less; (2) when a task has to be semi-partitioned, the algorithm tries to allocate the largest share possible $s_1(\tau)$ from the remaining utilization on processors; then, it tries to find the best fit for the remaining share $s_2(\tau)$, in order to leave larger “chunks” of processor available utilization to remaining (unallocated) tasks.

The algorithm makes use of the $sp\_assign$ function to try and assign task shares. As shown in Algorithm 2, this function checks two conditions. First, there must be enough available utilization on the processor to accommodate the share. Second, in case another migrating task has already been mapped on processor $\pi$, condition (7) in Sec. III-C must be satisfied.

When an $m$-partition has been successfully found, the heuristic tries to optimize it (line 24 in Algorithm 1). The optimization consists in re-assigning the migrating task shares, whenever possible, to processors to which less fixed tasks are assigned. This way, less actors are affected by tardiness, leading to lower application latency and buffer size requirements. Note that in Algorithm 1 the first share of a migrating task $s_1(\tau)$ is set to the largest possible value, given the current available utilization of processors. This in turns makes the second share of each migrating task $s_2(\tau)$ as small as possible, making the process of optimization of the partition more effective.

VI. EVALUATION

We evaluate our approach using the StreamIt benchmarks considered in [18], for which we employ the unfolding technique.
described in [18] to derive larger CSDF graphs with improved throughput. Among these benchmarks, seven applications require, under the partitioned FFD allocation scheme, more processors than an optimal scheduler. This set of applications is listed in Table I. In this section we compare the system metrics obtained with three different allocation/scheduling approaches: (i) Partitioned EDF with FFD heuristic; (ii) Semi-partitioned EDF-fm, with our proposed FFD-SP heuristic; (iii) Semi-partitioned EDF-fm, with the LUF heuristic proposed in [11]. These approaches are denoted in Table I with FFD, FFD-SP, and fm-LUF, respectively.

Note that all the approaches in Table I lead to the same application throughput. This is because the throughput of an application depends on the period of its sink actor, which is unchanged in our analysis even in presence of task tardiness. In addition, we choose to compare the results of the LUF heuristic with our FFD-SP heuristic because, among the heuristics proposed in [11], LUF achieves the smallest number of processors.

The $m_{OPT}$ column in Table I lists the number of processors required by an optimal scheduler (for instance [7]) to execute the considered applications.

Let us focus on the comparison between the partitioned approach (FFD) and our proposed semi-partitioned approach (FFD-SP). We note that the FFD approach results in a number of processors ($m_{FFD}$) which is on average 17.6% greater than the number required by an optimal scheduler (see column $m_{OPT}$). In contrast, our FFD-SP algorithm requires on average only 2.1% more processors (see column $m_{SP}$), while maintaining the same throughput. This means that our proposed approach can exploit the available processors more efficiently, getting significantly closer to the results obtained by optimal schedulers (see columns $m_{OPT}$). However, this comes at two costs. (1) The first cost is the increase of memory requirements. For each benchmark, column $M_{FFD}$ reports the memory required by the partitioned approach, expressed in bytes. Compared to FFD, in FFD-SP the memory requirements increase due to both the size of buffers, that have to be enlarged to handle task tardiness, and the code size overhead of task replicas, which are necessary in case of migrating tasks. In Table I this increase in memory requirements is expressed by the ratio $\frac{M_{FFD}}{M_{OPT}}$. On average, our proposed approach requires 24.3% more memory compared to FFD. We argue that this increase in memory requirements is acceptable, given the fact that even the most memory-demanding application requires 516 KB, which is way less than the memory available in modern MPSoC systems. (2) The second cost is the increase in applications’ latency, due to the postponement of task start times needed to handle task tardiness. Column $L_{FFD}$ shows the applications’ latency, expressed in clock cycles, under FFD. The latency increase of our FFD-SP over FFD is on average 29.2% (see column $\frac{L_{FFD}}{L_{OPT}}$). We consider these memory and latency overheads imposed by our approach reasonable, especially when the throughput constraint of an application is more important than memory or latency constraints.

Finally, to evaluate the effectiveness of our proposed FFD-SP heuristic, we compare its results with LUF. We can see from the last two columns of Table I that over the considered benchmarks the EDF-fm approach with LUF heuristic incurs a much larger memory overhead (on average +65.9%, see column $\frac{M_{LUF}}{M_{OPT}}$) and latency increase (on average +90.2%, see column $\frac{L_{LUF}}{L_{OPT}}$) compared to FFD. Moreover, we note that for most applications the number of required processors is equal or greater when using LUF ($m_{LUF}$) compared to our FFD-SP ($m_{SP}$), with the exception of the Serpent performance application. This means that for most of the benchmarks our FFD-SP heuristic is either equal or more efficient than LUF in exploiting the available processing resources.

### VII. CONCLUSIONS

In this paper we prove that streaming applications modeled as acyclic CSDF graphs can be scheduled using a semi-partitioned soft real-time scheduler, providing hard real-time guarantees on the input/output interfaces between the application and the environment. We propose a novel heuristic that is aimed at reducing the number of required processors while keeping a low buffer size and latency overhead. In this semi-partitioned approach is used. Finally, we show on a set of real-life benchmarks that our approach can reduce the number of processors required to schedule those applications which incur bin-packing limitations of partitioned approaches, guaranteeing the same throughput and with a reasonable overhead in terms of memory requirements and application latency.

### REFERENCES