Abstract—Recently, it has been shown that the classical hard
time scheduling theory can be applied to streaming applica-
tions modeled as acyclic Cyclo-Static Dataflow (CSDF) graphs. 
However, many streaming applications are modeled as cyclic 
CSDF graphs, thus they are not supported by such scheduling 
theory. Therefore, in this paper, we propose an approach which 
also applies to the classical hard real-time scheduling theory 
on streaming applications modeled as cyclic CSDF graphs. The 
proposed approach converts each task in a cyclic CSDF graph to
a constrained-deadline periodic task. This conversion enables 
the utilization of many hard real-time scheduling algorithms which 
also offers properties such as temporal isolation and fast calculation
of the required number of processors for scheduling the tasks. 
We evaluate the performance of our approach in comparison to 
existing scheduling approaches. The evaluation, on a set of real-
life benchmarks, demonstrates that our approach can schedule 
the tasks in an application, modeled as a cyclic CSDF graph, with
guaranteed throughput equal or comparable to the throughput 
obtained by existing scheduling approaches while providing 
hard real-time guarantees for every task in the application 
thereby enabling temporal isolation among concurrently running 
tasks/applications on a multi-processor platform.

I. INTRODUCTION

Streaming applications is an important group of embedded
software that involves a wide spectrum of applications from
different domains such as image processing, video/audio processing, 
and digital signal processing. To handle the ever-increasing 
computational demand and hard real-time constraints of these 
applications, modern embedded systems have been equipped
with Multi-Processor System-on-Chip (MPSoC) to benefit from
parallel execution. Designing such embedded systems, however, 
imposes two main challenges: 1) how to efficiently represent parallelism found in a streaming application and 2) how to map 
and schedule the application tasks on an MPSoC platform such
that hard real-time requirements are satisfied.

To address the first challenge, parallel Models of Computation 
(MoCs) have been adopted as common practice for expressing 
the parallelism in an application to efficiently exploit the com-
putational capacity of MPSoCs [1]. Two well-known parallel 
MoCs are Synchronous Dataflow (SDF) [2] and its generaliza-
cion, Cyclo-Static Dataflow (CSDF) [3]. Within a parallel MoC, a
streaming application is represented as a task graph with concur-
rently executing and communicating tasks. Thus, the parallelism 
is explicitly exposed in the model.

To address the second challenge, recently, scheduling frame-
works, called Strictly Periodic Scheduling (SPS) [4], and Im-
proved Strictly Periodic Scheduling (ISPS) [5], have been pro-
posed to convert the tasks in a streaming application, modeled
as an acyclic CSDF graph, to a set of implicit-deadline periodic 
tasks. As a result, a variety of hard real-time scheduling algo-
rithms for periodic tasks, from the classical real-time scheduling 
theory [6], can be applied to schedule such streaming applica-
tions with a certain guaranteed performance, i.e., throughput/la-
tency. These algorithms can perform fast admission control and
scheduling decisions for new incoming applications in an MPSoC
platform using fast schedulability analysis, while providing hard
real-time guarantees and temporal isolation, i.e., the ability to
start/stop applications at runtime without affecting the timing 
behavior of other concurrently running applications on the same
MPSoC. In addition, these algorithms provide fast analytical
calculation of the minimum number of processors needed to 
schedule the tasks in an application instead of performing a 
complex and time-consuming design space exploration needed
by conventional static scheduling of streaming applications, i.e.,
self-timed scheduling [7].

The SPS and ISPS frameworks, however, are limited to
acyclic CSDF graphs and cannot schedule a streaming application 
modeled as a cyclic CSDF graph, i.e., a graph where the tasks
have cyclic data dependencies. Consequently, hard real-
time scheduling algorithms cannot be applied to many streaming
applications modeled as cyclic CSDF graphs. Therefore, in this
paper, we address the aforementioned limitation of the SPS [4] 
and ISPS [5] frameworks by proposing a novel scheduling frame-
work, called Generalized Strictly Periodic Scheduling (GSPS),
that can handle cyclic CSDF graphs. As a consequence, our
framework can cover a wider range of applications and enable
a variety of proven hard real-time scheduling algorithms [6] for
multiprocessors to be applied. More specifically, the main novel
contributions of this paper are summarized as follows:

• We propose a sufficient test to check for the existence of a
strictly periodic schedule for a streaming application modeled
as a cyclic (C)SDF graph;

• If a strictly periodic schedule exists for an application, the
tasks of the application are converted to a set of constrained-
deadline periodic tasks by computing their periods, deadlines,
and earliest start times. As a consequence, this conversion
enables the utilization of many well-developed hard real-time
scheduling algorithms [6] on streaming applications modeled
as cyclic (C)SDF graphs to benefit from the properties of these
algorithms such as hard real-time guarantees, fast admission
control, temporal isolation, and fast calculation of the number
of required processors;

• We show, on a set of real-life benchmarks, that our approach
can schedule the tasks in an application (cyclic (C)SDF graph)
严格 periodic tasks as cyclic (C)SDF graphs to benefit from the properties of these
algorithms such as hard real-time guarantees, fast admission
control, temporal isolation, and fast calculation of the number
of required processors;

• We show, on a set of real-life benchmarks, that our approach
can schedule the tasks in an application (cyclic (C)SDF graph)
strictly periodic tasks with hard real-time guaranteed
throughput which is equal or comparable to the throughput
obtained by existing scheduling approaches.

Scope of work. In this paper, we consider homogeneous MPSoCs
with distributed program and data memory to ensure predictabil-
ity of the execution at runtime and scalability. We assume that
the communication infrastructure used for inter-processor com-
munication is predictable, i.e., it provides guaranteed commu-
nication latency. We use the worst-case communication latency
to compute the worst-case execution time of a task, which in
our approach includes the worst-case time needed for the task’s
computation and the worst-case time needed to perform inter-task
data communication on the considered platform.

Paper organization. The remainder of the paper is organized as
follows: Section II gives an overview of the related work. Section
III introduces the background material needed for understanding the contributions of this paper. Section IV gives a motivational example. Section V presents the proposed GSPS scheduling framework. Section VI presents the results of the evaluation of the proposed framework. Finally, Section VII ends the paper with conclusions.

II. Related Work

In this section, we compare our hard real-time scheduling framework with the existing hard real-time and periodic scheduling approaches [4], [5], [8]–[10] for streaming applications. In [4] and [5], the authors convert each task in an acyclic CSDF graph to an implicit-deadline periodic task, by deriving the task’s earliest start time and period. In addition, the minimum buffer sizes of channels, that guarantee the strictly periodic execution of the tasks, are computed in [4] and [5]. These approaches, however, are limited to applications modeled as acyclic (C)SDF graphs. In contrast, our approach is more general than the approaches in [4] and [5] and can schedule an application, modeled as a cyclic (C)SDF graph, in strictly periodic fashion, if a strictly periodic schedule exists. As a result, many well-developed hard real-time scheduling algorithms [6] for periodic tasks can be applied to schedule the tasks in a cyclic CSDF graph to provide temporal isolation between concurrently running applications, fast admission control of new incoming applications, and to compute the minimum number of required processors, using fast schedulability tests.

Ali et al. [8] propose an algorithm to convert the tasks in an application to a set of constrained-deadline periodic tasks by extracting the tasks’ offset, arbitrary deadline, and period. Similar to our approach, this algorithm can deal with cyclic data dependencies in the application. However, this approach considers streaming applications modeled as Homogeneous SDF (HSDF) graphs derived by applying a certain transformation on initial (C)SDF graphs. Transforming a graph from (C)SDF to HSDF is a crucial step in which the number of tasks in the streaming application can exponentially grow, e.g., the HSDF graph of the application Echo [9], derived from a cyclic CSDF graph with 38 tasks, has over 42000 tasks. Such exponential growth of the application in terms of number of tasks can lead to a time-consuming analysis. Moreover, such exponential growth results in a significant memory overhead for storing the tasks’ code and significant scheduling overhead due to excessive task preemptions at runtime. In addition, the derived schedule, of a transformed (C)SDF graph to a HSDF graph, is valid if all multirate tasks in the (C)SDF graph are transformed to functionally equivalent single-rate tasks in the HSDF graph which requires modification of the tasks’ code. In contrast, our approach can be directly applied to streaming applications modeled with a more expressive MoC, i.e., (C)SDF graph, which avoids the significant memory and scheduling overheads introduced by large HSDF graphs as well as modification of the tasks’ code is not required. In addition, our approach is faster because it avoids the exponentially complex conversion of (C)SDF to HSDF.

In [9], the authors propose a framework to derive the maximum throughput of a CSDF graph under a periodic schedule and to calculate the minimum buffer sizes under a given throughput constraint. These are formulated as linear programming (LP) problems and solved approximately. In [10], a scheduling framework for exploration of the trade-off between throughput and minimum buffer sizes of (C)SDF graphs under self-timed scheduling is proposed. In [9], however, the calculation of the minimum number of processors required for the derived schedule is not taken into consideration. Moreover, the approaches in [9] and [10] do not provide hard real-time guarantees for every task in an application. Therefore, they do not ensure temporal isolation among tasks/applications. As a consequence, the schedule of already running applications has to be recalculated when a new application comes in the system. In contrast, our approach converts the tasks in applications to constrained-deadline periodic tasks. This conversion enables the utilization of many hard real-time scheduling algorithms [6] to provide temporal isolation and fast calculation of the minimum number of processors needed to schedule the tasks under certain throughput constraint. Moreover, we propose a simple analytical approach to test for the existence of a strictly periodic schedule and derive the maximum throughput of a CSDF graph under the strictly periodic schedule instead of approximately solving LP problems as done in [9].

III. Background

In this section, we provide a brief overview of the considered system model, the CSDF MoC, and the SPS [4] framework. This background is needed to understand the novel contributions of our work.

A. System Model

The considered MPSoC platforms in this work are homogeneous, i.e., they contain a set \( \Pi = [\pi_1, \pi_2, \ldots, \pi_m] \) of \( m \) identical processors with distributed memories. The processors execute a set \( \Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\} \) of \( n \) periodic tasks. Tasks can be preempted at any time. Every periodic task \( \tau_i \in \Gamma \) is represented by a tuple \( \tau_i = (C_i, S_i, D_i, T_i) \), where \( C_i \) is the worst-case execution time (WCET), \( S_i \) is the start time, \( D_i \) is the deadline, and \( T_i \) is the period of the task, where \( C_i \leq D_i \leq T_i \). When \( D_i = T_i \), the task \( \tau_i \) is an implicit-deadline periodic (IDP) task. Otherwise, the task \( \tau_i \) is a constrained-deadline periodic (CDP) task. The utilization of task \( \tau_i \), denoted as \( \omega_i \), is defined as \( \omega_i = C_i/T_i \), where \( \omega_i \in [0, 1] \). For a task set \( \Gamma \), \( \omega^T \) is the total utilization of \( \Gamma \) given by \( \omega^T = \sum_{i \in \Gamma} \omega_i \). Similarly, the density of task \( \tau_i \) is \( \delta_i = C_i/D_i \) and the total density of \( \Gamma \) is \( \delta^T = \sum_{i \in \Gamma} \delta_i \). For the CDP task model, the sufficient schedulability test for the global optimal scheduling [11] is \( \delta^T \leq m \). Therefore, the minimum number of processors according to this test for global optimal scheduling is:

\[
m_{\text{OPT}} = \lceil \delta^T \rceil. \tag{1}\]

The other class of scheduling algorithms for periodic task sets are partitioned algorithms [6] that do not require task migration. With partitioned scheduling, tasks are first allocated to processors and the tasks on each processor are scheduled using a uniprocessor scheduling algorithm. The minimum number of processors needed to schedule a task set \( \Gamma \) assuming partitioned scheduling is:

\[
m_{\text{POH}} = \min \{x | \exists x\text{-partition of } \Gamma \land \forall i \in [1, x] : \Gamma_i \text{ is schedulable on } \pi_i\}. \tag{2}\]

B. Cyclo-Static Data Flow (CSDF)

An application modeled as a CSDF [3] graph is a directed graph \( G = (V, E) \), where \( V \) is a set of tasks and \( E \) is a set of edges. Task \( \tau_i \in V \) represents computation and edge \( e_u = (\tau_i, \tau_j) \in E \) represents the transfer of data tokens from task \( \tau_i \) to task \( \tau_j \). Each task \( \tau_i \in V \) has \( P_i \) phases and an execution sequence \( [f(1), f(2), \ldots, f(P_i)] \) of length \( P_i \). This means that the execution of each phase \( f \) of task \( \tau_i \) is associated with a certain function \( f(\phi) \). Consequently, the execution time and the data production/consumption rate for each output/input edge of task \( \tau_i \) are also defined for each phase. Therefore, each task \( \tau_i \in V \) has the following sequences of length \( P_i \) of the WCETs \( [C(1), C(2), \ldots, C(P_i)] \), a predefined data production sequence of \( [x^{(i)}(1), x^{(i)}(2), \ldots, x^{(i)}(P_i)] \) on its every output channel \( e_u \), and a predefined data consumption sequence.
of $y^0_i(1), y^1_i(2), \cdots, y^y_i(P_i)$ on its every input channel $e_i$. If every task $\tau_j$ in a CSDF graph $G$ has a single phase, i.e., $P_i = 1$, then the graph $G$ is an SDF [2] graph that means the SDF MoC is a subset of the CSDF MoC.

It has been proven in [3] that a valid static schedule of a CSDF graph can be generated at design-time if the graph is consistent and live. A CSDF graph is said to be consistent if a non-trivial solution exists for the repetition vector $\vec{q} = [q_1, q_2, \cdots, q_n]^T \in \mathbb{N}^n$. An entry $q_i$ indicates the number of invocations of task $\tau_j$ in one graph iteration of the CSDF graph. If a deadlock-free schedule can be found, $G$ is then said to be live. Throughout this paper, we consider and use consistent and live CSDF graphs. Fig. 1 shows an example of a cyclic CSDF graph with $\vec{q} = [3, 2, 1, 2]^T$. The sequence of the WCETs of each task $\tau_j$ is shown below its name. For instance, task $\tau_1$ has the sequence of the WCETs $[1, 2, 1]$ time units and its data production sequence on channel $e_1$ is $[1, 0, 1]$.

C. Strictly Periodic Scheduling Framework

In [4], the strictly periodic scheduling (SPS) framework for acyclic CSDF graphs is proposed. In this framework, every task $\tau_j \in V$ in an acyclic CSDF graph $G$ is converted to a real-time IDP task by deriving its period ($T_j$) and earliest start time ($S_j$). In this framework, the period ($T_j$) of every task $\tau_j \in V$ is derived by the following expression:

$$T_j = \frac{lcm(\vec{q})}{q_j} \cdot s, \forall \tau_j \in V.$$  

(3)

where $lcm(\vec{q})$ is the least common multiple of all repetition entries in $\vec{q}$, $W = \max_{j \in V}[C_j/q_j]$, and $C_j = \max_{i \in P_j}[C_i(\phi)]$. Note that when the scaling factor $s = \delta = \frac{W}{lcm(\vec{q})}$, the minimum period ($T_j$) is derived using Eq. (3). In general, the derived periods of tasks satisfy the condition $q_i T_j = \cdots = q_i T_n = \alpha$, where $\alpha$ is the graph iteration period representing the duration needed by the graph to complete one iteration. Once the tasks periods are computed, the throughput of each task $\tau_j$ can be computed as $1/T_j$. The throughput $\mathcal{R}$ of graph $G$, defined as the number of samples the graph can produce during a given time interval, is determined by the period of the output task ($T_{out}$) and is given by $\mathcal{R} = 1/T_{out}$.

Then, the earliest start time of task $\tau_j \in V$, denoted $S_{j}$, is calculated such that $\tau_j$ is never blocked on reading data tokens from any input FIFO channel connected to it during its periodic execution, using the following expression:

$$S_j = \begin{cases} 0 & \text{if } \text{prec}(\tau_j) = \emptyset \\ \max_{r \in \text{prec}(\tau_j)}(S_{r} - \delta) & \text{if } \text{prec}(\tau_j) \neq \emptyset \end{cases}$$  

(5)

where $\text{prec}(\tau_j)$ represents the set of predecessor tasks of $\tau_j$ and $S_{r - \delta}$ is given by:

$$S_{r - \delta} = \min_{[T_r, e_u]} \left\{ t : \prod_{r \in S_j + k} \left( T_r, e_u \right) \geq \text{cns}_{[T_r, e_u]}(\tau_j, e_u), \forall k \in [0, \alpha] \right\}$$  

(6)

where $S_j$ is the earliest start time of a predecessor task $\tau_j$, $\prod_{r \in S_j + k} \left( T_r, e_u \right)$ is the total number of tokens produced by $\tau_i$ to

de edge $e_j$ during the time interval $[t_s, t_e]$, and $\text{cns}_{[T_r, e_u]}(\tau_j, e_u)$ is the total number of tokens consumed by $\tau_j$ from edge $e_u$ during the time interval $[t_s, t_e]$.

In this framework, once the start times of tasks have been calculated, the minimum buffer size of each communication channel $e_j$ connecting tasks $\tau_j$ and $\tau_i$ is calculated that is the maximum number of stored data tokens in channel $e_j$ during the execution of $\tau_j$ and $\tau_i$ in one graph iteration period. The application latency is also calculated as the elapsed time between the arrival of a data sample to the application and the output of the processed sample by the application.

IV. Motivational Example

The goal of this section is to show how the tasks in the cyclic CSDF graph $G$, shown in Fig. 1, can be scheduled in strictly periodic fashion using our GSPS framework proposed in Section V. First, assume that $G$ has no backward edge $e_j$. Then, $G$ has no cycles and the SPS framework [4] can convert the tasks in $G$ to IDP tasks represented by the following tuples: $\tau_1 = (C_1 = 2, S_1 = 0, T_1 = 2), \tau_2 = (2, 3, 3), \tau_3 = (3, 4, 6)$, and $\tau_4 = (3, 9, 3)$. The scheduling for this periodic task set is shown in Fig. 2. Considering $e_5$, however, this schedule is not valid because there is no data token available on $e_5$ for $\tau_1$ to consume at time 8 and therefore the strict periodicity of tasks’ execution is no longer guaranteed.

To solve this problem, we must ensure that $\tau_4$ can produce a data token before the fifth firing of $\tau_1$, as shown by the dashed line in Fig. 2. Therefore, $e_5$ introduces a latency constraint between $\tau_1$ and $\tau_5$. Please note that, the derived periods of the tasks, for the schedule shown in Fig. 2, are the minimum periods ($T_j$) by using the scaling factor $s = \delta = \frac{W}{lcm(\vec{q})}$ = 1 in Eq. (3). But, there exist other longer valid periods for a task by using any integer $s > \delta = \frac{W}{lcm(\vec{q})} = 1$ in Eq. (3). By taking $s = 3$, a new schedule can be derived that can respect the latency constraint introduced by backward edge $e_5$ to guarantee strict periodicity of the tasks’ execution, as shown in Fig. 3. In this schedule, the tasks are CDP tasks that are represented by the following tuples in task set $\Gamma = [\tau_1 = (C_1 = 2, S_1 = 0, D_1 = 3, T_1 = 6), \tau_2 = (2, 6, 3, 9), \tau_3 = (3, 9, 18, 18), \tau_4 = (3, 18, 3, 9)]$. Please note that the deadline ($D_1$) of each task is derived with the goal of minimizing the number of required processors to schedule the tasks. The above example shows that the tasks in the cyclic CSDF graph $G$ can be converted to a set of CDP tasks, thus, a variety of hard real-time scheduling algorithms [6] can be applied to the cyclic CSDF graph $G$ in order to provide temporal isolation, fast admission control, and easy calculation of the minimum required processors. For instance, for the set $\Gamma$ of CDP tasks in Fig. 3, $\delta_1 = 2.5$ and the minimum number of processors for optimal and partitioned First-Fit Increasing Deadlines EDF (FFID-EDF) [6] schedulers are $m_{\text{OPT}} = 3$ and $m_{\text{EDF}} = 3$ according to Eq. (1) and Eq. (2), respectively. Therefore, the goal of our GSPS framework proposed in Section V is to test for the existence and to derive such strictly periodic schedule for an application modeled as a cyclic CSDF graph which implies that the tasks in the graph can be converted to a set of CDP tasks.
V. Proposed Approach

In this section, we present our analytical framework, called Generalized Strictly Periodic Scheduling (GSPS), for scheduling and converting the tasks in a cyclic CSDF graph to a set of CDP tasks. First, we test for the existence of a strictly periodic schedule for a cyclic (C)SDF graph in Section V-A. Then, if a strictly periodic schedule exists, the period ($T_i$), deadline ($D_i$), and earliest start time ($S_i$) of each periodic (CDP) task are computed, in Section V-B, such that all data dependencies between the tasks are satisfied with the goal of minimizing the number of required processors to schedule the tasks.

A. Existence of a Strictly Periodic Schedule

As explained in Section IV, to find a strictly periodic schedule for a cyclic (C)SDF graph, an appropriate scaling factor $s \geq \bar{s}$ has to be determined such that all latency constraints introduced by backward edges are satisfied. Therefore, to test for the existence of a strictly periodic schedule, the existence of such scaling factor $s$ must be tested. To do so, we need to analyze the start times of the tasks belonging to each cycle in the (C)SDF graph. Using Eq. (6) and the minimum periods of each periodic (CDP) task, we can define interval $\Delta_i$ for each edge $e_u = (\tau_i, \tau_j) \in E$ as follows:

$$\Delta_i = S_i - S_j - D_j$$

Eq. (9) for $G$ in Fig. 1, we have $\Delta_1 = 2$, $\Delta_3 = 2$, $\Delta_2 = 3$, $\bar{\Delta}_2 = -1$, $\bar{\Delta}_3 = 2$, and $\bar{\Delta}_1 = -7$.

The $\bar{\Delta}_i$ interval is the key component in our analysis to find a strictly periodic schedule for the tasks in a cyclic (C)SDF graph. Since the $\bar{\Delta}_i$ interval is calculated using the minimum period computed by Eq. (3) with scaling factor $s = \bar{s}$, we need to find how interval $\bar{\Delta}_i$ changes by taking scaling factor $s > \bar{s}$. This is provided by the following lemma:

**Lemma 1.** $\bar{\Delta}_i$ interval changes proportionally to the scaling factor $s$ as follows:

$$\bar{\Delta}_i = \frac{\Delta_i}{\bar{s}} \cdot s$$

where $\bar{s}$ is the minimum scaling factor computed by Eq. (4) and $\bar{\Delta}_i$ is the minimum interval computed by Eq. (7).

**Proof.** Consider an arbitrary edge $e_u = (\tau_i, \tau_j) \in E$ where the data token production and consumption curves can be visualized similarly to Fig. 4. For the minimum periods ($\bar{T}_i$ and $\bar{T}_j$) of tasks $\tau_i$ and $\tau_j$ computed using Eq. (3) with $s = \bar{s}$, we assume that the critical point $\Phi$ happens after $x$ and $y$ executions of $\tau_i$ and $\tau_j$, respectively. Therefore, we have $x \cdot \bar{T}_i = y \cdot \bar{T}_j + \bar{\Delta}_i$, which implies $S_i + D_i \leq \bar{\Delta}_i$ and $S_j - D_j = \bar{\Delta}_i$, which is illustrated in Fig. 4, when $D_i = C_i$. To execute task $\tau_j$ in strictly periodic fashion, the cumulative data token production of $\tau_j$ on channel $e_u$ must always be greater than or equal to the cumulative data token consumption of $\tau_i$ from $e_u$. This is ensured by shifting the consumption curve by $\bar{\Delta}_i$ time units to the right after the deadline of $\tau_i$, as shown in Fig. 4. In Fig. 4, point $\Phi$ is a critical point determining that the consumption curve cannot be shifted to the left because the consumption curve will be above the production curve. Thus $\tau_j$ cannot start execution earlier than $\bar{\Delta}_i$ time units.

To compute $S_j$ using Eq. (6), for edge $e_u$, $S_i$ must be known. Therefore, to use Eq. (6) for each edge independently, we assume

$$S_j = \left\lceil \gamma \left( \sum_{l=0}^{q_j} y_j l ((l - 1) \text{mod} P_j) + 1 \right) + 1\right\rceil$$

where $\gamma$ is the number of initial tokens on channel $e_u$, $\sum_{l=0}^{q_j} y_j l ((l - 1) \text{mod} P_j) + 1$ is the amount of tokens that $\tau_j$ consumes from $e_u$ during one graph iteration, and $\left\lceil \gamma \left( \sum_{l=0}^{q_j} y_j l ((l - 1) \text{mod} P_j) + 1 \right) + 1\right\rceil$ is the maximum number of graph iterations where $\tau_j$ can execute before starting $\tau_i$. This $S_j$ is sufficiently large to ensure that actual $\bar{\Delta}_i$ can be computed. For example, using Eq. (7), Eq. (6), and Eq. (9) for $G$ in Fig. 1, we have $\bar{\Delta}_1 = 2$, $\bar{\Delta}_3 = 2$, $\bar{\Delta}_2 = 3$, $\bar{\Delta}_3 = 2$, and $\bar{\Delta}_1 = -7$.

However, $\bar{\Delta}_j \leq x \cdot \bar{T}_j$ is not possible due to the fact that $y$'s executions of $\tau_j$ cannot finish before finishing $x$'s executions of $\tau_i$ because the consumption curve cannot be above the production curve. Therefore, from Eq. (15), we can only have

$$y \cdot \bar{T}_j \leq \bar{\Delta}_j$$

Now, assume that after taking scaling factor $s > \bar{s}$, a new critical point $\Phi'$ exists after $x'$ and $y'$ executions of $\tau_i$ and $\tau_j$, respectively. Therefore, we have $x' \cdot \bar{T}_i = y' \cdot \bar{T}_j + \bar{\Delta}_i$, which implies $S_j = \bar{\Delta}_i$ and $S_i - D_i = \bar{\Delta}_i$, which is illustrated in Fig. 4, when $D_i = C_i$. To execute task $\tau_j$ in strictly periodic fashion, the cumulative data token production of $\tau_j$ on channel $e_u$ must always be greater than or equal to the cumulative data token consumption of $\tau_i$ from $e_u$. This is ensured by shifting the consumption curve by $\bar{\Delta}_i$ time units to the right after the deadline of $\tau_i$, as shown in Fig. 4. In Fig. 4, point $\Phi'$ is a critical point determining that the consumption curve cannot be shifted to the left because the consumption curve will be above the production curve. Thus $\tau_j$ cannot start execution earlier than $\bar{\Delta}_i$ time units.
Now, we propose a sufficient test for the existence of a strictly periodic schedule for a cyclic (C)SDF graph by formulating a theorem and prove it by using Lemma 1.

**Theorem 1.** For the tasks in a cyclic (C)SDF graph G, a strictly periodic schedule exists if for every cyclic path ℓ = \{τℓ₁→τℓ₂→⋯→τℓₖ→τℓ₁\} ∈ L in G:

\[
\sum_{j=1}^{k} \bar{\Delta}_{\ell-\ell(j \mod \tau_{\ell})} < 0.
\]

where L is a set of all cyclic paths in G and \(\bar{\Delta}_{\ell-\ell(j \mod \tau_{\ell})}\) is computed using Eq. (7).

**Proof.** In a cyclic path \(\ell = \{τℓ₁→τℓ₂→⋯→τℓₖ→τℓ₁\} \in L\) and assuming an arbitrary scaling factor \(s \geq \bar{s}\), the earliest start time \(S_{\ell₁}\) of task \(τ_{\ell₁}\), when \(D_{\ell₁} = C_{\ell₁}\), \(\forall τ_{\ell₁} \in L\), can be computed by considering its predecessor task \(τ_{\ell(k-1)}\) using Eq. (8) as follows:

\[
S_{\ellₖ} = S_{\ell₁} + \sum_{j=1}^{k} C_{\ell_j} + \sum_{j=1}^{k} \bar{\Delta}_{\ell-\ell(j \mod \tau_{\ell})}.
\]

Now, by recursively computing \(S_{\ell(k-1)}\) and substituting it in the above equation, the earliest start time \(S_{\ell₁}\) of task \(τ_{\ell₁}\) is:

\[
S_{\ell₁} = S_{\ell₁} + \sum_{j=1}^{\ell₁-1} C_{\ell_j} + \sum_{j=1}^{\ell₁-1} \bar{\Delta}_{\ell-\ell(j \mod \tau_{\ell})}.
\]

Due to the edge from \(\tau_{\ell₁}\) to \(\tau_{\ell₁}\), the starting time \(S_{\ell₁}\) of \(\tau_{\ell₁}\) is constrained by Eq. (8) as follows:

\[
S_{\ell₁} + C_{\ell₁} + \bar{\Delta}_{\ell₁-\ell₁} \leq S_{\ell₁}.
\]

By using Eq. (10) (Lemma 1) and Eq. (18) in Eq. (19), we have

\[
S_{\ell₁} + \sum_{j=1}^{\ell₁} C_{\ell_j} + \frac{\bar{s}}{s} \sum_{j=1}^{\ell₁} \bar{\Delta}_{\ell-\ell(j \mod \tau_{\ell})} \leq S_{\ell₁}.
\]

\[
\Rightarrow \sum_{j=1}^{\ell₁} C_{\ell_j} + \frac{s}{\bar{s}} \sum_{j=1}^{\ell₁} \bar{\Delta}_{\ell-\ell(j \mod \tau_{\ell})} \leq 0.
\]

Equation (20) holds only if \(\sum_{j=1}^{\ell₁} \bar{\Delta}_{\ell-\ell(j \mod \tau_{\ell})} < 0\), because \(\sum_{j=1}^{\ell₁} C_{\ell_j}\), \(\bar{s}\), and \(s\) are positive numbers by definition and we can always select sufficiently large scaling factor \(s \geq \bar{s}\). \(\Box\)

**B. Deriving Period, Earliest Start Time, and Deadline of Tasks**

In this section, we derive the period, deadline, and earliest start time of each task in a cyclic (C)SDF graph scheduled in strictly periodic fashion, if such schedule exists according to Theorem 1.

(1) Period: Considering Eq. (20), the minimum scaling factor \(s\) that satisfies Eq. (20) is:

\[
s = \frac{\bar{s}}{s} \cdot \frac{\sum_{i=1}^{\ell₁} C_{\ell_i}}{-\sum_{i=1}^{\ell₁} \bar{\Delta}_{\ell-\ell(i \mod \tau_{\ell})} + 1}.
\]

Since there may exist several cyclic paths in the graph, the minimum scaling factor \(s\) for the graph that guarantees strictly periodic execution of all tasks is:

\[
s = \left[\bar{s} \cdot \max_{\ell₁ \in L} \left(\frac{\sum_{i=1}^{\ell₁} C_{\ell_i}}{-\sum_{i=1}^{\ell₁-1} \bar{\Delta}_{\ell-\ell(i \mod \tau_{\ell})} + 1}\right)\right].
\]

Then, using Eq. (3) and the above computed scaling factor \(s\), the periods of the tasks can be derived.

(2) Deadline: Since the number of processors needed to schedule a task set \(Γ\) of CDP tasks depends on the total density \(δ_Γ\) of the tasks [6], our objective to derive the deadline of the tasks is to minimize \(δ_Γ\) in order to minimize the number of processors. Therefore, we formulate our optimization problem as follows:

\[
\begin{align*}
\text{Minimize} & \quad \delta_Γ = \sum_{τ \in Γ} \frac{C_{τ}}{D_{τ}} \\
\text{subject to:} & \quad S_j + D_i - S_j - \Delta_{\tau_i-\tau_j} \forall (\tau_i, \tau_j) \in E \quad (21b) \\
& \quad -D_i \leq -C_i, \quad D_i \leq T_i, \quad \forall τ_i \in Γ \quad (21c)
\end{align*}
\]

where Eq. (21a) is the objective function and \(D_i\) is an optimization variable. In addition, Eqs. (21b) are the constraints given by Eq. (8), and Eqs. (21c) bound all optimization variables in the objective function by the WCET \(C_i\) and period \(T_i\) derived in Section V-B(1). \(S_j\) and \(S_j\) are implicit variables which are not in the objective function Eq. (21a), but still need to be considered in the optimization procedure.

(3) Earliest Start Time: To derive the earliest start times of the tasks, we use the derived deadline of the tasks in Section V-B(2) in the following optimization problem:

\[
\begin{align*}
\text{Minimize} & \quad \sum_{τ \in Γ} S_i \\
\text{subject to:} & \quad S_i - S_j - \Delta_{\tau_i-\tau_j} - D_i \forall (\tau_i, τ_j) \in E \quad (22b) \\
& \quad -S_j \leq 0, \quad ∀ τ_j ∈ Γ \quad (22c)
\end{align*}
\]

where Eq. (22a) is the objective function and \(S_i\) is an optimization variable. In addition, Eqs. (22b) are the constraints given by Eq. (8), and Eqs. (22c) bound all optimization variables in the objective function to be greater or equal to zero. Given that all variables in both problems Eqs. (21) and (22) are integers and both the objective functions and the constraints are convex, the problems are integer convex programming problems [17], which can be solved by using the existing CVX solver [18].

**VI. Evaluation**

In this section, we present experiments to evaluate our GSPS framework proposed in Section V. As explained earlier, our GSPS framework enables the application of many hard real-time scheduling algorithms [6], which offer properties such as hard real-time guarantees, temporal isolation, fast admission control and scheduling decisions for new incoming applications, and easy and fast calculation of the number of processors needed for scheduling the tasks, on streaming applications modeled as cyclic (C)SDF graphs. However, having these properties is not for free. Thus, the goal of these experiments is to show what the cost is for having these properties using our GSPS framework in terms of the maximum achievable application throughput, the application latency, and the buffer sizes of the communication channels compared to scheduling frameworks, such as periodic scheduling (PS) [9] and self-timed scheduling (STS) [10], which also can be applied directly on cyclic (C)SDF graphs but do not provide such properties. The experiments have been performed on a set of ten real-life streaming applications, modeled as cyclic (C)SDF graphs, taken from different sources. These applications are listed in Table I. In this table, |V| and |E| denote the number of tasks and communication channels in a (C)SDF graph, respectively.

The results of the evaluation for throughput \(R\) (one token/time units), latency \(L\) (time units), and buffer sizes of the communication channels \(M\) (number of data tokens) of the applications under our GSPS, PS, and STS are given in Table II. The throughput, latency, and buffer sizes of the applications under our GSPS, denoted by \(R_{\text{GSPS}}, L_{\text{GSPS}}, \frac{M_{\text{GSPS}}}{2}\), are given in columns 2, 3, and 4 in Table II, respectively. Columns 7 and 10 show the ratio between the throughput of the output tasks under our GSPS and PS and STS, respectively. Looking at column 7, we can see that our GSPS can achieve the same throughput.
Table II

Comparison of different scheduling frameworks.

<table>
<thead>
<tr>
<th>Application</th>
<th>$\mathcal{K}_{\text{GSPS}}$</th>
<th>$\mathcal{L}_{\text{GSPS}}$</th>
<th>$\mathcal{L}_{\text{PS}}$</th>
<th>$\mathcal{L}_{\text{STS}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modern</td>
<td>1/16</td>
<td>64</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>MPEG-4 AVC</td>
<td>1/16</td>
<td>15264</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>MPEG-4 SP</td>
<td>1/3600</td>
<td>11088</td>
<td>881</td>
<td>2</td>
</tr>
<tr>
<td>MP3 Decoder</td>
<td>1/3732.288</td>
<td>353903292</td>
<td>42684</td>
<td>4</td>
</tr>
<tr>
<td>MP5 playback</td>
<td>1/25</td>
<td>46355</td>
<td>3958</td>
<td>3</td>
</tr>
<tr>
<td>WLAN</td>
<td>1/6</td>
<td>18</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>TDS-COMA</td>
<td>1/6750000</td>
<td>792289</td>
<td>91</td>
<td>8</td>
</tr>
<tr>
<td>Channel Equalizer</td>
<td>1/260</td>
<td>1899</td>
<td>24</td>
<td>7</td>
</tr>
<tr>
<td>Echo</td>
<td>1/268823760000</td>
<td>80754156016</td>
<td>30287</td>
<td>13</td>
</tr>
</tbody>
</table>

The results in Table II show that our GSPS framework can achieve latency and buffer sizes closer or equal to the latency and buffer sizes of the applications under PS and STS.

VII. Conclusion

In this paper, we have presented our GSPS framework to test for the existence of strictly periodic schedule for streaming applications modeled as cyclic CSDF graphs. Then, if such schedule exists, our GSPS converts each task in the graph to a constrained-deadline periodic task. This conversion enables the utilization of many hard real-time scheduling algorithms which offer properties such as temporal isolation and fast calculation of the required number of processors. Finally, we show, on a set of real-life benchmarks, that strictly periodic scheduling is capable of delivering equal or comparable throughput to existing approaches for the majority of the applications we experimented with.

VIII. Acknowledgment

This research is supported by The Netherlands Organisation for Scientific Research (NWO) under project CPS-P3 (12695).


References