Hard Real-Time Scheduling of Streaming Applications Modeled as Cyclic CSDF Graphs

Sobhan Niknam, Peng Wang and Todor Stefanov
Leiden Institute of Advanced Computer Science, Leiden University, Leiden, The Netherlands
Email: {s.niknam, p.wang, t.p.stefanov}@liacs.leidenuniv.nl

***Abstract*—Recently, it has been shown that the classical hard real-time scheduling theory can be applied to streaming applications 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 enables to apply 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 offer 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 computational capacity of MPSoCs [1]. Two well-known parallel MoCs are Synchronous Dataflow (SDF) [2] and its generalization, Cyclo-Static Dataflow (CSDF) [3]. Within a parallel MoC, a streaming application is represented as a task graph with concurrently executing and communicating tasks. Thus, the parallelism is explicitly exposed in the model.

To address the second challenge, recently, scheduling frameworks, called Strictly Periodic Scheduling (SPS) [4], and Improved Strictly Periodic Scheduling (ISPS) [5], have been proposed 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 algorithms for periodic tasks, from the classical real-time scheduling theory [6], can be applied to schedule such streaming applications with a certain guaranteed performance, i.e., throughput/latency. 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 framework, 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) as 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 predictability of the execution at runtime and scalability. We assume that the communication infrastructure used for inter-processor communication is predictable, i.e., it provides guaranteed communication 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

978-3-9819263-2-3/DATE19/©2019 EDAA

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 multi-rate 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 $\Pi = \{\pi_1, \pi_2, \cdots, \pi_m\}$ of m identical processors with distributed memories. The processors execute a set $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$ 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 τ_i is an implicit-deadline periodic (IDP) task. Otherwise, the task τ_i is a constrained-deadline periodic (CDP) task. The utilization of task τ_i, denoted as u_i, is defined as $u_i = C_i/T_i$, where $u_i \in (0, 1]$. For a task set Γ, u_Γ is the total utilization of Γ given by $u_\Gamma = \sum_{\tau_i \in \Gamma} u_i$. Similarly, the density of task τ_i is $\delta_i = C_i/D_i$ and the total density of Γ is $\delta_\Gamma = \sum_{\tau_i \in \Gamma} \delta_i$. For the CDP task model, the sufficient schedulability test for the global optimal scheduling [11] is $\delta_\Gamma \leq m$. Therefore, the minimum number of processors according to this test for global optimal scheduling is:

$$m_{\text{OPT}} = \lceil \delta_\Gamma \rceil. \quad (1)$$

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 Γ assuming partitioned scheduling is:

$$m_{\text{PAR}} = \min_{x \in \mathbb{N}} \{x \mid \exists\ x\text{-partition of } \Gamma \land \forall i \in [1, x] : \Gamma_i \text{ is schedulable on } \pi_i\}. \quad (2)$$

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 τ_i to task τ_j. Each task $\tau_i \in V$ has P_i phases and an *execution sequence* $[f_i(1), f_i(2), \cdots, f_i(P_i)]$ of length P_i. This means that the execution of each phase ϕ of task τ_i is associated with a certain function $f_i(\phi)$. Consequently, the execution time and the data production/consumption rate for each output/input edge of task τ_i are also defined for each phase. Therefore, each task $\tau_i \in V$ has the following sequences of length P_i: a sequence of the WCETs $[C_i(1), C_i(2), \cdots, C_i(P_i)]$, a predefined data production sequence of $[x_i^u(1), x_i^u(2), \cdots, x_i^u(P_i)]$ on its every output channel e_u, and a predefined data consumption sequence

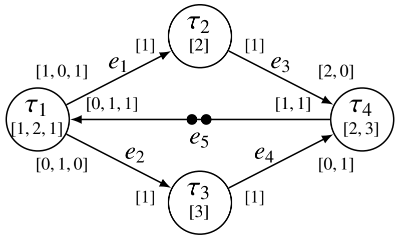

Fig. 1. A **cyclic** CSDF graph G. The backward edge e_5 in G has 2 initial tokens that are represented with black dots.

of $[y_i^u(1), y_i^u(2), \cdots, y_i^u(P_i)]$ on its every input channel e_u. If every task τ_i 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 τ_i 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 τ_i is shown below its name. For instance, task τ_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 $\tau_i \in V$ in an *acyclic* CSDF graph G is converted to a real-time IDP task by deriving its *period* (T_i) and *earliest start time* (S_i). In this framework, the *period* (T_i) of every task $\tau_i \in V$ is derived by the following expression:

$$T_i = \frac{\mathrm{lcm}(\vec{q})}{q_i} \cdot s, \ \forall \tau_i \in V, \quad (3) \qquad\qquad s \geq \left\lceil \frac{\hat{W}}{\mathrm{lcm}(\vec{q})} \right\rceil, \quad (4)$$

where $\mathrm{lcm}(\vec{q})$ is the least common multiple of all repetition entries in $\vec{q}$, $\hat{W} = \max_{\tau_j \in V}\{C_j . q_j\}$ is the maximum task workload of the CSDF graph, and $C_j = \max_{1 \leq \phi \leq P_j}\{C_j(\phi)\}$. *Note that when the scaling factor* $s = \check{s} = \lceil \hat{W}/lcm(\vec{q}) \rceil$*, the minimum period* ($\check{T}_i$) *is derived using Eq. (3).* In general, the derived periods of tasks satisfy the condition $q_1 T_1 = \cdots = q_n T_n = \alpha$, where α is the graph iteration period representing the duration needed by the graph to complete one iteration. Once the task periods are computed, the throughput of each task τ_i can be computed as $1/T_i$. 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 τ_j is never blocked on reading data tokens from any input FIFO channel connected to it during its periodic execution, using the following expression:

$$S_j = \begin{cases} 0 & if \ \mathrm{prec}(\tau_j) = \emptyset \\ \max_{\tau_i \in \mathrm{prec}(\tau_j)}(S_{i \to j}) & if \ \mathrm{prec}(\tau_j) \neq \emptyset \end{cases} \qquad (5)$$

where $\mathrm{prec}(\tau_j)$ represents the set of predecessor tasks of τ_j and $S_{i \to j}$ is given by:

$$\begin{aligned} S_{i \to j} = \min_{t \in [0, S_i + \alpha]} \{t : & \ \mathsf{prd}_{[S_i, \max\{S_i, t\} + k)}(\tau_i, e_u) \\ & \geq \mathsf{cns}_{[t, \max\{S_i, t\} + k]}(\tau_j, e_u), \ \forall k \in [0, \alpha]\}, \end{aligned} \qquad (6)$$

where S_i is the earliest start time of a predecessor task τ_i, $\mathsf{prd}_{[t_s, t_e)}(\tau_i, e_u)$ is the total number of tokens produced by τ_i to edge e_u during the time interval $[t_s, t_e)$, and $\mathsf{cns}_{[t_s, t_e]}(\tau_j, e_u)$ is the total number of tokens consumed by τ_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_u connecting tasks τ_i and τ_j is calculated that is the maximum number of stored data tokens in channel e_u during the execution of τ_i and τ_j 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.

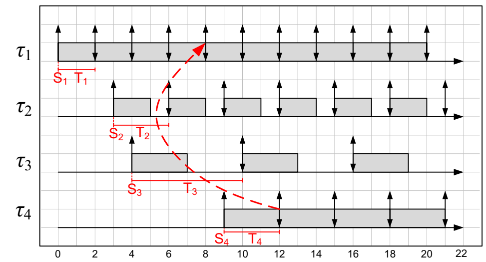

Fig. 2. The SPS of the CSDF graph G in Fig. 1 without considering the backward edge e_5. Up arrows are job releases and down arrows job deadlines.

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_5. Then, G has no cycles and the SPS framework [4] can convert the tasks in G to IDP tasks represented by the following tuples: $\tau_1 = (C_1 = 2, S_1 = 0, \check{T}_1 = 2)$, $\tau_2 = (2, 3, 3)$, $\tau_3 = (3, 4, 6)$, and $\tau_4 = (3, 9, 3)$. The schedule 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 τ_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 τ_4 can produce a data token before the fifth firing of τ_1, as shown by the dashed line in Fig. 2. Therefore, e_5 introduces a latency constraint between τ_1 and τ_5. Please note that the derived periods of the tasks, for the schedule shown in Fig. 2, are the minimum periods ($\check{T}_i$) by using the scaling factor $s = \check{s} = \lceil \hat{W}/\mathrm{lcm}(\vec{q}) \rceil = 1$ in Eq. (3). But, there exist other longer valid periods for a task by using any integer $s > \check{s} = \lceil \hat{W}/\mathrm{lcm}(\vec{q}) \rceil = 1$ 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_i) 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 Γ of CDP tasks in Fig. 3, $\delta_\Gamma = 2.5$ and the minimum number of processors for optimal and partitioned First-Fit Increasing Deadlines EDF (FFID-EDF) [6] schedulers are $m_{\mathrm{OPT}} = 3$ and $m_{\mathrm{PAR}} = 3$ 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.

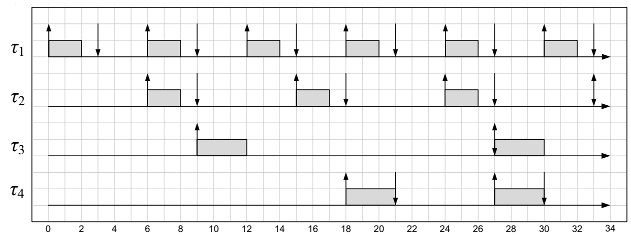

Fig. 3. The GSPS of the CSDF graph G in Fig. 1.

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 $s \geq \check{s}$ 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 the tasks ($\check{T}_i$), we can define interval $\check{\Delta}_{i\to j}$ for each edge $e_u = (\tau_i, \tau_j) \in E$ as follows:

$$\check{\Delta}_{i\to j} = S_{i\to j} - S_i - D_i \tag{7}$$

that is the minimum distance between the deadline (D_i) of task τ_i and the earliest start time ($S_{i\to j}$) of task τ_j due to edge e_u. This means that τ_j cannot start execution earlier than $\check{\Delta}_{i\to j}$ time units after the deadline of τ_i, i.e.,

$$S_i + D_i + \check{\Delta}_{i\to j} \leq S_j. \tag{8}$$

Otherwise, task τ_j cannot find enough data tokens on edge e_u to read in order to execute in strictly periodic fashion. The data token production and consumption curves on edge e_u along with the $\check{\Delta}_{i\to j}$ interval are illustrated in Fig. 4, when $D_i = C_i$. To execute task τ_j in strictly periodic fashion, the cumulative data token production of τ_i on channel e_u must always be greater than or equal to the cumulative data token consumption of τ_j from e_u. This is ensured by shifting the consumption curve by $\check{\Delta}_{i\to j}$ time units to the right after the deadline of τ_i, as shown in Fig. 4. In Fig. 4, point Φ 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 τ_j cannot start execution earlier than $S_{i\to j}$.

To compute $S_{i\to j}$ using Eq. (6) for edge e_u, S_i must be known. Therefore, to use Eq. (6) for each edge independently, we assume

$$S_i = \left(\left\lfloor \gamma / \sum_{l=1}^{q_j} y_j^u(((l-1) \bmod P_j)+1)\right\rfloor + 1\right)\alpha, \tag{9}$$

where γ is the number of initial tokens on channel e_u, $\sum_{l=1}^{q_j} y_j^u(((l-1) \bmod P_j)+1)$ is the amount of tokens that τ_j consumes from e_u during one graph iteration, and $\lfloor \gamma / \sum_{l=1}^{q_j} y_j^u(((l-1) \bmod P_j)+1)\rfloor$ is the maximum number of graph iterations where τ_j can execute before starting τ_i. This S_i is sufficiently large to ensure that actual $\check{\Delta}_{i\to j}$ can be computed. For example, using Eq. (7), Eq. (6), and

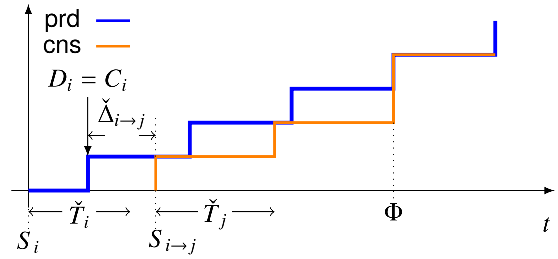

Fig. 4. Production and consumption curves on edge $e_u = (\tau_i, \tau_j)$.

Eq. (9) for G in Fig. 1, we have $\check{\Delta}_{1\to 2} = 1$, $\check{\Delta}_{1\to 3} = 2$, $\check{\Delta}_{2\to 4} = 3$, $\check{\Delta}_{3\to 4} = -3$, and $\check{\Delta}_{4\to 1} = -7$.

The $\check{\Delta}_{i\to j}$ 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 $\check{\Delta}_{i\to j}$ interval is calculated using the minimum period computed by Eq. (3) with scaling factor $s = \check{s}$, we need to find how interval $\check{\Delta}_{i\to j}$ changes by taking scaling factor $s > \check{s}$. This is provided by the following lemma.

Lemma 1. *The $\Delta_{i\to j}$ interval changes proportionally to the scaling factor s as follows:*

$$\Delta_{i\to j} = \frac{\check{\Delta}_{i\to j}}{\check{s}} \cdot s \tag{10}$$

where $\check{s}$ is the minimum scaling factor computed by Eq. (4) and $\check{\Delta}_{i\to j}$ 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 ($\check{T}_i$ and $\check{T}_j$) of tasks τ_i and τ_j computed using Eq. (3) with $s = \check{s}$, we assume that the critical point Φ happens after x and y executions of τ_i and τ_j, respectively, e.g., 3 executions of τ_i and 2 executions of τ_j in Fig. 4, that implies

$$S_i + D_i + x \cdot \check{T}_i = S_{i\to j} + y \cdot \check{T}_j \overset{(7)}{\Longleftrightarrow} x \cdot \check{T}_i = y \cdot \check{T}_j + \check{\Delta}_{i\to j} \tag{11}$$

$$\overset{(3)}{\Longleftrightarrow} (x \cdot \frac{\mathrm{lcm}(\vec{q})}{q_i} - y \cdot \frac{\mathrm{lcm}(\vec{q})}{q_j}) = \frac{\check{\Delta}_{i\to j}}{\check{s}}. \tag{12}$$

Now, we assume that after taking scaling factor $s > \check{s}$, a new critical point Φ' exists after x' and y' executions of τ_i and τ_j, respectively. Therefore, we have

$$x' \cdot T_i = y' \cdot T_j + \Delta_{i\to j} \overset{(3)}{\Longleftrightarrow} (x' \cdot \frac{\mathrm{lcm}(\vec{q})}{q_i} - y' \cdot \frac{\mathrm{lcm}(\vec{q})}{q_j}) = \frac{\Delta_{i\to j}}{s}. \tag{13}$$

Moreover, for the previous critical point Φ, we know that y executions of τ_j cannot finish before finishing x executions of τ_i because the consumption curve cannot be above the production curve. Therefore, after taking scaling factor $s > \check{s}$, we still have

$$x \cdot T_i \leq y \cdot T_j + \Delta_{i\to j} \overset{(3)}{\Longleftrightarrow} (x \cdot \frac{\mathrm{lcm}(\vec{q})}{q_i} - y \cdot \frac{\mathrm{lcm}(\vec{q})}{q_j}) \leq \frac{\Delta_{i\to j}}{s}. \tag{14}$$

Then, by substituting Eq. (12) and Eq. (13) in Eq. (14), we have

$$\frac{\check{\Delta}_{i\to j}}{\check{s}} \leq (x' \cdot \frac{\mathrm{lcm}(\vec{q})}{q_i} - y' \cdot \frac{\mathrm{lcm}(\vec{q})}{q_j}) \overset{(3)}{\Longleftrightarrow} y' \cdot \check{T}_j + \check{\Delta}_{i\to j} \leq x' \cdot \check{T}_i. \tag{15}$$

However, $y' \cdot \check{T}_j + \check{\Delta}_{i\to j} < x' \cdot \check{T}_i$ is not possible due to the fact that y' executions of τ_j cannot finish before finishing x' executions of τ_i for the critical point Φ' because the consumption curve cannot be above the production curve. Therefore, from Eq. (15), we can only have

$$y' \cdot \check{T}_j + \check{\Delta}_{i\to j} = x' \cdot \check{T}_i \overset{(11)}{\Longleftrightarrow} x' \cdot \check{T}_i - y' \cdot \check{T}_j = x \cdot \check{T}_i - y \cdot \check{T}_j$$

$$\overset{(3)}{\Longleftrightarrow} (x' \cdot \frac{\mathrm{lcm}(\vec{q})}{q_i} - y' \cdot \frac{\mathrm{lcm}(\vec{q})}{q_j}) = (x \cdot \frac{\mathrm{lcm}(\vec{q})}{q_i} - y \cdot \frac{\mathrm{lcm}(\vec{q})}{q_j}). \tag{16}$$

From Eq. (12), Eq. (13), and Eq. (16) we can conclude that $\Delta_{i\to j}/s = \check{\Delta}_{i\to j}/\check{s} \Leftrightarrow \Delta_{i\to j} = \check{\Delta}_{i\to j}/\check{s} \cdot s$. □

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* $\ell = \{\tau_{\ell 1} \to \tau_{\ell 2} \to \cdots \to \tau_{\ell x} \to \tau_{\ell 1}\} \in L$ *in G:*

$$\sum_{i=1}^{x} \check{\Delta}_{\ell i \to \ell((i \bmod x)+1)} < 0. \tag{17}$$

where L is a set of all cyclic paths in G and $\check{\Delta}_{\ell i \to \ell((i \bmod x)+1)}$ *is computed using Eq. (7).*

Proof. In a cyclic path $\ell = \{\tau_{\ell 1} \to \tau_{\ell 2} \to \cdots \to \tau_{\ell x} \to \tau_{\ell 1}\} \in L$ and assuming an arbitrary scaling factor $s_\ell \geq \check{s}$, the *earliest start time* $S_{\ell x}$ of task $\tau_{\ell x}$, when $D_{\ell i} = C_{\ell i}$, $\forall \tau_{\ell i} \in L$, can be computed by considering its predecessor task $\tau_{\ell(x-1)}$ using Eq. (8) as follows: $S_{\ell x} = S_{\ell(x-1)} + C_{\ell(x-1)} + \Delta_{\ell(x-1) \to \ell x}$. Now, by recursively computing $S_{\ell(x-1)}$ and substituting it in the above equation, the *earliest start time* $S_{\ell x}$ of task $\tau_{\ell x}$ is:

$$S_{\ell x} = S_{\ell 1} + \sum_{i=1}^{x-1} C_{\ell i} + \sum_{i=1}^{x-1} \Delta_{\ell i \to \ell(i+1)}. \tag{18}$$

Due to the edge from $\tau_{\ell x}$ to $\tau_{\ell 1}$, the *starting time* $S_{\ell 1}$ of $\tau_{\ell 1}$ is constrained by Eq. (8) as follows:

$$S_{\ell x} + C_{\ell x} + \Delta_{\ell x \to \ell 1} \leq S_{\ell 1}. \tag{19}$$

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

$$S_{\ell 1} + \sum_{i=1}^{x} C_{\ell i} + \frac{s_\ell}{\check{s}} \cdot \sum_{i=1}^{x} \check{\Delta}_{\ell i \to \ell((i \bmod x)+1)} \leq S_{\ell 1}$$

$$\Leftrightarrow \sum_{i=1}^{x} C_{\ell i} + \frac{s_\ell}{\check{s}} \cdot \sum_{i=1}^{x} \check{\Delta}_{\ell i \to \ell((i \bmod x)+1)} \leq 0. \tag{20}$$

Eq. (20) holds only if $\sum_{i=1}^{x} \check{\Delta}_{\ell i \to \ell((i \bmod x)+1)} < 0$, because $\sum_{i=1}^{x} C_{\ell i}$, $\check{s}$, and s_ℓ are positive numbers by definition and we can always select sufficiently large scaling factor $s_\ell \geq \check{s}$. □

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_ℓ that satisfies Eq. (20) is:

$$s_\ell = \check{s} \cdot \frac{\sum_{i=1}^{x} C_{\ell i}}{-\sum_{i=1}^{x} \check{\Delta}_{\ell i \to \ell((i \bmod x)+1)}}.$$

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:

$$s = \left\lceil \check{s} \cdot \max\left(\max_{\forall \ell \in L}\left(\frac{\sum_{i=1}^{x} C_{\ell i}}{-\sum_{i=1}^{x} \check{\Delta}_{\ell i \to \ell((i \bmod x)+1)}}\right), 1\right) \right\rceil.$$

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:

$$\text{Minimize} \quad \delta_\Gamma = \sum_{\tau_i \in \Gamma} \frac{C_i}{D_i} \tag{21a}$$

$$\text{subject to:} \quad S_i + D_i - S_j \leq -\Delta_{i \to j} \quad \forall e_u = (\tau_i, \tau_j) \in E \tag{21b}$$

$$-D_i \leq -C_i, \; D_i \leq T_i \quad \forall \tau_i \in \Gamma \tag{21c}$$

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_i 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.

TABLE I
BENCHMARKS USED FOR EVALUATION.

Application	$\lvert V \rvert$	$\lvert E \rvert$	Source
Modem	16	35	[12]
MP3 playback	4	4	
MP3 Decoder	15	21	[13]
MPEG-4 Advanced Video Coding (AVC) Decoder	4	6	
MPEG-4 Simple Profile (SP) Decoder	5	10	
Channel Equalizer	10	22	
WLAN 802.11p transceiver	8	9	[14]
TDS-CDMA receiver	16	25	[15]
Long Term Evolution (LTE)	10	15	[16]
Echo	38	82	[9]

(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:

$$\text{Minimize} \quad \sum_{\tau_i \in \Gamma} S_i \tag{22a}$$

$$\text{subject to:} \quad S_i - S_j \leq -\Delta_{i \to j} - D_i \quad \forall e_u = (\tau_i, \tau_j) \in E \tag{22b}$$

$$-S_i \leq 0 \quad \forall \tau_i \in \Gamma \tag{22c}$$

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 $\mathcal{R}$ (one token/time units), latency $\mathcal{L}$ (time units), and buffer sizes of the communication channels $\mathcal{M}$ (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 $\mathcal{R}_{\text{out}}^{\text{GSPS}}$, $\mathcal{L}^{\text{GSPS}}$, and $\mathcal{M}^{\text{GSPS}}$, 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.

Application	**GSPS**					**PS** [9]			**STS** [10]		
	$\mathcal{R}^{\text{GSPS}}_{\text{out}}[\frac{1}{\text{t.u.}}]$	$\mathcal{L}^{\text{GSPS}}$[t.u.]	$\mathcal{M}^{\text{GSPS}}$[Tkn]	$m^{\text{GSPS}}_{\text{OPT}}$	$m^{\text{GSPS}}_{\text{PAR}}$	$\frac{\mathcal{R}^{\text{GSPS}}_{\text{out}}}{\mathcal{R}^{\text{PS}}_{\text{out}}}$	$\frac{\mathcal{L}^{\text{GSPS}}}{\mathcal{L}^{\text{PS}}}$	$\frac{\mathcal{M}^{\text{GSPS}}}{\mathcal{M}^{\text{PS}}}$	$\frac{\mathcal{R}^{\text{GSPS}}_{\text{out}}}{\mathcal{R}^{\text{STS}}_{\text{out}}}$	$\frac{\mathcal{L}^{\text{GSPS}}}{\mathcal{L}^{\text{STS}}}$	$\frac{\mathcal{M}^{\text{GSPS}}}{\mathcal{M}^{\text{STS}}}$
Modem	1/16	64	50	10	10	1	2.78	1.25	1	2.78	1.25
MPEG-4 AVC	1/7632	15264	6	4	4	1	1.04	1	1	1.04	1
MPEG-4 SP	1/3960	11088	881	2	2	1	2.35	2.02	1	2.35	2.02
MP3 Decoder	1/3732288	33590592	42674	4	4	1	5.46	3.06	1	6.70	-
MP3 playback	1/25	46355	3958	3	4	1	1.12	1.22	0.91	1.30	-
WLAN	1/6	18	14	7	8	1	1.5	1.07	0.92	1.5	0.93
TDS-CDMA	1/675000	792829	44	7	8	1	1.62	1.19	1	1.62	1.19
LTE	1/280	1284	27	5	6	1	2.99	1.28	1	2.99	1.28
Channel Equalizer	1/9264	18989	24	7	7	0.91	1.57	1	0.66	-	1
Echo	1/26882376000	80754156016	30287	13	19	0.19	15.75	1.08	0.19	-	1.08

obtained by PS for 8 out of 10 applications. Looking at column 10, we can also see that the throughput under our GSPS is equal or very close to the throughput under STS, that is the optimal scheduling in terms of throughput, for the majority of the applications. In both comparisons, the biggest difference is in the case of Echo. This is mainly because, our GSPS schedules all the phases of a task in a CSDF graph as a periodic task where different firing of the task corresponds to one of the phases of the task. Therefore, in contrast to PS and STS, the starting time of the execution phases of the task is delayed under our GSPS. As a consequence, if a multi-phase task exists in a cycle, a larger scaling factor may be required by our GSPS to find a strictly periodic schedule that results in a lower throughput compared to PS and STS. From these comparisons, we can conclude that although our GSPS results in a lower throughput for a few applications compared to PS and STS, achieving the properties of the hard real-time scheduling algorithms is for free in terms of the maximum achievable throughput for the majority of the applications under our GSPS.

For processor requirements under our GSPS, we compute the minimum number of processors under optimal and partitioned FFID-EDF [6] schedulers by using Eq. (1) and Eq. (2), denoted with $m^{\text{GSPS}}_{\text{OPT}}$ and $m^{\text{GSPS}}_{\text{PAR}}$ in Table II, respectively. However, for PS, the calculation of the number of processors was not considered in [9], and for STS, finding the minimum number of processors requires complex design space exploration to find the best allocation which delivers the maximum achievable throughput [7]. This fact shows one advantage of using our GSPS compared to using PS and STS when our GSPS gives the same throughput as PS and STS.

Let us now analyze the latency and the buffer sizes of the applications. Columns 8 and 11 gives the ratio of the maximum latency of the applications under our GSPS to the latency of the applications under PS and STS, respectively. As we can see, the average latency of the applications under our GSPS is 3.8 and 2.5 times larger than the latency under PS and STS, respectively. Similarly, the ratio of the buffer sizes of the applications under our GSPS to the buffer sizes under PS and STS is given in columns 9 and 12, respectively. From these columns, we can see that the buffer sizes in our GSPS are on average 1.4 and 1.21 times larger than the buffer sizes under PS and STS. Obviously, the larger latency and buffer sizes of the channels for the applications are the main costs in our GSPS framework to enable the utilization of hard real-time scheduling algorithms on streaming applications modeled as cyclic (C)SDF graphs. Please note that, our GSPS causes larger latency and buffer sizes because of the minimization of the number of processors we perform using Eqs. (21), while PS and STS cause lower latency and buffer sizes because they do not perform such minimization. Therefore, if we also do not perform the processor minimization and only perform minimization of the start times of the tasks using Eqs. (22) with $D_i = C_i, \forall \tau_i \in \Gamma$, our GSPS 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

[1] A. Gerstlauer et al. Electronic system-level synthesis methodologies. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2009.

[2] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[3] G. Bilsen et al. Cycle-static dataflow. *IEEE Transactions on signal processing*, 44(2):397–408, 1996.

[4] M. Bamakhrama and T. Stefanov. On the hard-real-time scheduling of embedded streaming applications. *DAES*, 17(2):221–249, 2013.

[5] J. Spasic et al. On the improved hard real-time scheduling of cyclo-static dataflow. *ACM Trans. Embedded Computing Systems*, 15(4):68, 2016.

[6] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35, 2011.

[7] S. Stuijk et al. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *DAC*, 2007.

[8] H. Ali et al. Generalized extraction of real-time parameters for homogeneous synchronous dataflow graphs. In *PDP*, 2015.

[9] B. Bodin, A. Munier-Kordon, and B. D. de Dinechin. Periodic schedules for cyclo-static dataflow. In *ESTIMedia*, 2013.

[10] S. Stuijk et al. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, 57(10):1331–1345, 2008.

[11] T. Baker and S. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In *Handbook of Real-Time and Embedded Systems*. CRC Press, 2007.

[12] Sdf^3, http://www.es.ele.tue.nl/sdf3/download/examples.php.

[13] B. D. Theelen et al. Scenario-aware dataflow. *Technical Report ESR-2008-08*, 2008, http://www.es.ele.tue.nl/sadf/examples.php.

[14] P. S. Kurtin, J. PHM Hausmans, and M. JG Bekooij. Combining offsets with precedence constraints to improve temporal analysis of cyclic real-time streaming applications. In *RTAS*, 2016.

[15] O. Moreira. Temporal analysis and scheduling of hard real-time radios running on a multi-processor. *ser. PHD Thesis, TU Eindhoven*, 2012.

[16] F. Siyoum et al. Analyzing synchronous dataflow scenarios for dynamic software-defined radio applications. In *SoC*, 2011.

[17] D. Liu et al. Resource optimization for csdf-modeled streaming applications with latency constraints. In *DATE*, 2014.

[18] M. Grant, S. Boyd, and Y. Ye. Cvx: Matlab software for disciplined convex programming, 2008.