Generalized strictly periodic scheduling analysis, resource optimization, and implementation of adaptive streaming applications

Niknam, S.

Chapter 3

Hard Real-Time Scheduling of Cyclic CSDF Graphs

Sobhan Niknam, Peng Wang, Todor Stefanov. "Hard Real-Time Scheduling of Streaming Applications Modeled as Cyclic CSDF Graphs". *In Proceedings of the International Conference on Design, Automation and Test in Europe (DATE'19)*, pp. 1528-1533, Florence, Italy, March 25 - 29, 2019.

IN this chapter, we present our Generalized Strictly Periodic Scheduling (GSPS) framework, which corresponds to the first research contribution, briefly introduced in Section 1.5.1, to address research question **RQ1**, described in Section 1.4.1. The remainder of this chapter is organized as follows. Section 3.1 introduces, in more details, the problem statement and the addressed research question. It is followed by Section 3.2, which gives a summary of the contributions presented in this chapter. An overview of the related work is given in Section 3.3. A motivational example is given in Section 3.4. Then, Section 3.5 presents our proposed GSPS framework. Section 3.6 presents the experimental evaluation of our proposed GSPS framework. Finally, Section 3.7 ends the chapter with conclusions.

3.1 Problem Statement

Recall, from Section 2.3, that the Strictly Periodic Scheduling (SPS) framework [8] has been recently proposed to convert 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 hard real-time scheduling theory [21,29] (briefly introduced in Section 2.2), can be applied to schedule such streaming applications with a certain guaranteed performance, i.e., throughput/latency, on MPSoC platforms. 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. In addition, these algorithms provide a 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 [85]. The SPS framework, however, is limited to acyclic CSDF graphs and cannot schedule a streaming application modeled as a cyclic CSDF graph, i.e., a graph where the actors have cyclic data dependencies. Consequently, hard real-time scheduling algorithms cannot be applied to many streaming applications modeled as cyclic CSDF graphs. Thus, in this chapter, we investigate the possibility to apply scheduling algorithms from the classical hard real-time scheduling theory to streaming applications modeled as cyclic CSDF graphs.

3.2 Contributions

In order to address the problem described in Section 3.1, in this chapter, we propose a novel scheduling framework, called Generalized Strictly Periodic Scheduling (GSPS), that can handle cyclic CSDF graphs. As a consequence, our framework enables the application of a variety of proven hard real-time scheduling algorithms [21,29] for multiprocessor systems on a wider range of applications compared to the SPS framework. More specifically, the main novel contributions of this chapter 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 [29] 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 streaming applications, that our approach can schedule the tasks in an application, modeled as a 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.

3.3 Related Work

In this section, we compare our hard real-time scheduling framework with the existing hard real-time and periodic scheduling approaches [3, 8, 18, 79, 85] for streaming applications. In [8] and [78], the authors convert each actor in an *acyclic* CSDF graph to an implicit-deadline periodic task, by deriving the actor's earliest start time and period. In addition, the minimum buffer sizes of FIFO channels, that guarantee the strictly periodic execution of the tasks, are computed in [8] and [78]. These approaches, however, are limited to applications modeled as *acyclic* (C)SDF graphs. In contrast, our approach is more general than the approaches in [8] and [78] 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 [29] for periodic tasks can be applied to schedule the actors 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. [3] 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 [18], derived from a cyclic CSDF graph with 38 actors, has over 42000 actors. 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 actors

in the (C)SDF graph are transformed to functionally equivalent single-rate actors in the HSDF graph which requires modification of the actors' 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 actors' code is not required. In addition, our approach is faster because it avoids the exponentially complex conversion of (C)SDF to HSDF.

In [18], 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 [85], 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 [18], however, the calculation of the minimum number of processors required for the derived schedule is not taken into consideration. Moreover, the approaches in [18] and [85] 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 [29] 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 [18].

3.4 Motivational Example

The goal of this section is to show how the actors in the cyclic CSDF graph G, shown in Figure 3.1, can be scheduled in strictly periodic fashion using our GSPS framework proposed in Section 3.5. First, assume that G has no backward edge E_5. Then, G has no cycles and the SPS framework [8] (described in Section 2.3) can convert the actors in G to IDP tasks represented by the following tuples: $\tau_1 = (C_1 = 2, \check{T}_1 = 2, S_1 = 0, D_1 = \check{T}_1 = 2)$, $\tau_2 = (2, 3, 3, 3)$, $\tau_3 = (3, 6, 4, 6)$, and $\tau_4 = (3, 3, 9, 3)$. The schedule for this periodic task set is shown in Figure 3.2. Considering E_5, however, this schedule is not valid

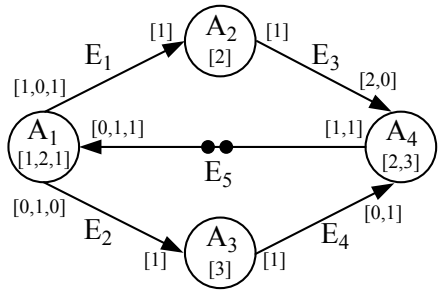

Figure 3.1: *A **cyclic** CSDF graph G. The backward edge E_5 in G has 2 initial tokens that are represented with black dots.*

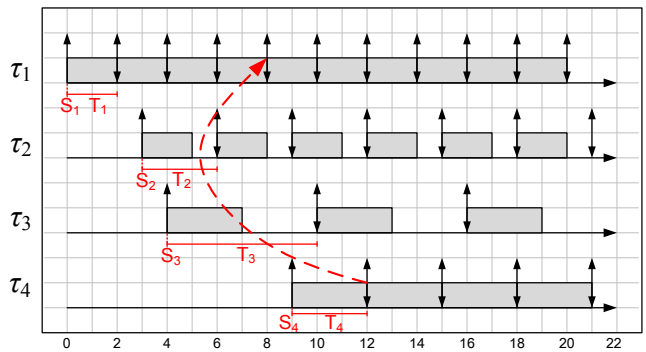

Figure 3.2: *The SPS of the CSDF graph G in Figure 3.1 without considering the backward edge E_5. Up arrows are job releases and down arrows job deadlines.*

because there is no data token available on E_5 for task τ_1 (corresponding to actor A_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 task τ_4 (corresponding to actor A_4) can produce a data token before the fifth firing of task τ_1, as shown by the dashed line in Figure 3.2. Therefore, E_5 introduces a latency constraint between tasks τ_1 and τ_4. Please note that the derived periods of the tasks, for the schedule shown in Figure 3.2, are the minimum periods ($\check{T}_i$) by using the scaling factor $s = \check{s} = \lceil \hat{W}/\text{lcm}(\vec{q}) \rceil = 1$ in Equation (2.12). But, there exist other longer valid periods for a task by using any integer $s > \check{s} = \lceil \hat{W}/\text{lcm}(\vec{q}) \rceil = 1$ in Equation (2.12). 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 Figure 3.3. In this schedule, the tasks are CDP tasks that are represented by the following tuples in task set $\Gamma = \{\tau_1 = (C_1 = 2, T_1 =$

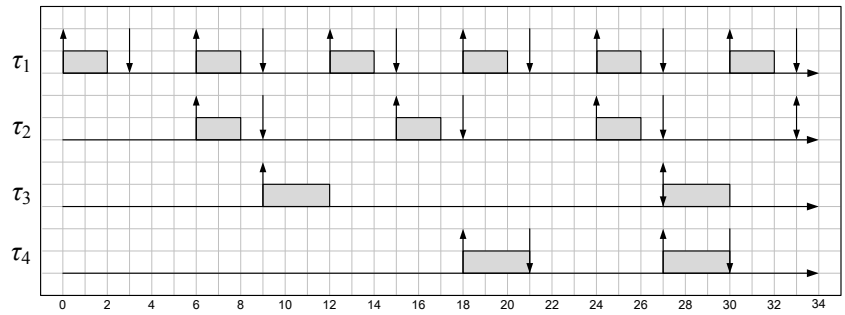

Figure 3.3: *The GSPS of the CSDF graph G in Figure 3.1.*

$6, S_1 = 0, D_1 = 3)$, $\tau_2 = (2, 9, 6, 3)$, $\tau_3 = (3, 18, 9, 18)$, $\tau_4 = (3, 9, 18, 3)\}$. Please note that the deadline 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 actors in the cyclic CSDF graph G can be converted to a set of CDP tasks, thus, a variety of hard real-time scheduling algorithms [29] 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 Figure 3.3, $\delta_\Gamma = 2.5$ and the minimum number of processors for global and partitioned First-Fit Increasing Deadlines EDF (FFID-EDF) [29] schedulers are $\check{m} = 3$ and $\check{m}_{\text{PAR}} = 3$ according to Equation (2.10) and Equation (2.11), respectively. Therefore, the goal of our GSPS framework proposed in Section 3.5 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 actors in the graph can be converted to a set of CDP tasks.

3.5 Our Proposed Framework

In this section, we present our analytical GSPS framework for scheduling and converting the actors 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 3.5.1. Then, if a strictly periodic schedule exists, each actor A_i of the graph is converted to a CDP task τ_i by deriving the period (T_i), deadline (D_i), and earliest start time (S_i) of the task, in Section 3.5.2, such that all data dependencies between the tasks are satisfied with the goal of minimizing the number of required processors to schedule the CDP tasks.

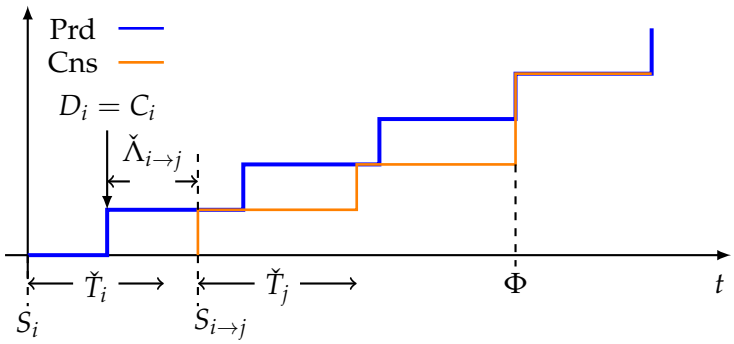

Figure 3.4: *Production and consumption curves on edge $E_u = (A_i, A_j)$.*

3.5.1 Existence of a Strictly Periodic Schedule

As explained in Section 3.4, 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 corresponding to the actors belonging to each cycle in the (C)SDF graph. Using Equation (2.17) and the minimum periods of the tasks ($\check{T}_i$), we can define interval $\check{\Lambda}_{i\to j}$ for each edge $E_u = (A_i, A_j) \in \mathcal{E}$ as follows:

$$\check{\Lambda}_{i\to j} = S_{i\to j} - S_i - D_i \tag{3.1}$$

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

$$S_i + D_i + \check{\Lambda}_{i\to j} \leq S_j. \tag{3.2}$$

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 $\check{\Lambda}_{i\to j}$ interval are illustrated in Figure 3.4, when $D_i = C_i$. To execute task τ_j in strictly periodic fashion, the cumulative data token production of task τ_i on channel E_u must always be greater than or equal to the cumulative data token consumption of task τ_j from E_u. This is ensured by shifting the consumption curve by $\check{\Lambda}_{i\to j}$ time units to the right after the deadline of task τ_i, as shown in Figure 3.4. In Figure 3.4,

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 task τ_j cannot start execution earlier than $S_{i\to j}$.

To compute $S_{i\to j}$ using Equation (2.17) for edge E_u, S_i must be known. Therefore, to use Equation (2.17) for each edge independently, we assume

$$S_i = \left(\left\lfloor \frac{\gamma}{Y_j^u(q_j)} \right\rfloor + 1 \right) H, \tag{3.3}$$

where γ is the number of initial tokens on channel E_u, $Y_j^u(q_j) = \sum_{l=1}^{q_j} y_j^u(((l-1) \bmod \phi_j)+1)$ is the amount of tokens that task τ_j corresponding to actor A_j consumes from E_u during one graph iteration, $\lfloor \gamma / Y_j^u(q_j) \rfloor$ is the maximum number of graph iterations where task τ_j can execute before starting task τ_i, H is the iteration period. This S_i is sufficiently large to ensure that actual $\check{\Lambda}_{i\to j}$ can be computed. For example, using Equation (3.1), Equation (2.17), and Equation (3.3) for G in Figure 3.1, we have $\check{\Lambda}_{1\to 2} = 1$, $\check{\Lambda}_{1\to 3} = 2$, $\check{\Lambda}_{2\to 4} = 3$, $\check{\Lambda}_{3\to 4} = -3$, and $\check{\Lambda}_{4\to 1} = -7$.

The $\check{\Lambda}_{i\to j}$ interval is the key component in our analysis to find a strictly periodic schedule for the actors in a **cyclic** (C)SDF graph. Since the $\check{\Lambda}_{i\to j}$ interval is calculated using the minimum period computed by Equation (2.12) with scaling factor $s = \check{s}$, we need to find how interval $\check{\Lambda}_{i\to j}$ changes by taking scaling factor $s > \check{s}$. This is provided by the following lemma.

Lemma 3.5.1. *The $\Lambda_{i\to j}$ interval changes proportionally to the scaling factor s as follows:*

$$\Lambda_{i\to j} = \frac{\check{\Lambda}_{i\to j}}{\check{s}} \cdot s \tag{3.4}$$

where $\check{s}$ is the minimum scaling factor computed by Equation (2.13) and $\check{\Lambda}_{i\to j}$ is the minimum interval computed by Equation (3.1).

Proof. Consider an arbitrary edge $E_u = (A_i, A_j) \in \mathcal{E}$ where the data token production and consumption curves can be visualized similarly to Figure 3.4. For the minimum periods ($\check{T}_i$ and $\check{T}_j$) of tasks τ_i and τ_j corresponding to actors A_i and A_j computed using Equation (2.12) with $s = \check{s}$, we assume that the critical point Φ happens after x and y executions of tasks τ_i and τ_j, respectively, e.g., 3 executions of task τ_i and 2 executions of task τ_j in Figure 3.4, that implies

$$S_i + D_i + x \cdot \check{T}_i = S_{i\to j} + y \cdot \check{T}_j \overset{(3.1)}{\iff} x \cdot \check{T}_i = y \cdot \check{T}_j + \check{\Lambda}_{i\to j} \tag{3.5}$$

$$\overset{(2.12)}{\iff} \left(x \cdot \frac{\operatorname{lcm}(\vec{q})}{q_i} - y \cdot \frac{\operatorname{lcm}(\vec{q})}{q_j}\right) = \frac{\check{\Lambda}_{i\to j}}{\check{s}}. \tag{3.6}$$

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

$$x' \cdot T_i = y' \cdot T_j + \Lambda_{i\to j} \overset{(2.12)}{\iff} (x' \cdot \frac{\text{lcm}(\vec{q})}{q_i} - y' \cdot \frac{\text{lcm}(\vec{q})}{q_j}) = \frac{\Lambda_{i\to j}}{s}. \tag{3.7}$$

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

$$x \cdot T_i \leq y \cdot T_j + \Lambda_{i\to j} \overset{(2.12)}{\iff} (x \cdot \frac{\text{lcm}(\vec{q})}{q_i} - y \cdot \frac{\text{lcm}(\vec{q})}{q_j}) \leq \frac{\Lambda_{i\to j}}{s}. \tag{3.8}$$

Then, by substituting Equation (3.6) and Equation (3.7) in Equation (3.8), we have

$$\frac{\check{\Lambda}_{i\to j}}{\check{s}} \leq (x' \cdot \frac{\text{lcm}(\vec{q})}{q_i} - y' \cdot \frac{\text{lcm}(\vec{q})}{q_j}) \overset{(2.12)}{\iff} y' \cdot \check{T}_j + \check{\Lambda}_{i\to j} \leq x' \cdot \check{T}_i. \tag{3.9}$$

However, $y' \cdot \check{T}_j + \check{\Lambda}_{i\to j} < x' \cdot \check{T}_i$ is not possible due to the fact that y' executions of task τ_j cannot finish before finishing x' executions of task τ_i for the critical point Φ' because the consumption curve cannot be above the production curve. Therefore, from Equation (3.9), we can only have

$$y' \cdot \check{T}_j + \check{\Lambda}_{i\to j} = x' \cdot \check{T}_i \overset{(3.5)}{\iff} x' \cdot \check{T}_i - y' \cdot \check{T}_j = x \cdot \check{T}_i - y \cdot \check{T}_j$$

$$\overset{(2.12)}{\iff} (x' \cdot \frac{\text{lcm}(\vec{q})}{q_i} - y' \cdot \frac{\text{lcm}(\vec{q})}{q_j}) = (x \cdot \frac{\text{lcm}(\vec{q})}{q_i} - y \cdot \frac{\text{lcm}(\vec{q})}{q_j}). \tag{3.10}$$

From Equation (3.6), Equation (3.7), and Equation (3.10) we can conclude that

$$\frac{\Lambda_{i\to j}}{s} = \frac{\check{\Lambda}_{i\to j}}{\check{s}} \Leftrightarrow \Lambda_{i\to j} = \frac{\check{\Lambda}_{i\to j}}{\check{s}} \cdot s.$$

■

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 3.5.1.

Theorem 3.5.1. *For the tasks corresponding to actors in a **cyclic** (C)SDF graph G, a strictly periodic schedule exists if for every cyclic path* $\vartheta = \{A_{\vartheta 1} \leftrightarrow A_{\vartheta 2} \leftrightarrow \cdots \leftrightarrow A_{\vartheta x} \leftrightarrow A_{\vartheta 1}\} \in \mathcal{V}$ *in G:*

$$\sum_{i=1}^{x} \check{\Lambda}_{\vartheta i \rightarrow \vartheta((i \bmod x)+1)} < 0. \tag{3.11}$$

where $\mathcal{V}$ *is a set of all cyclic paths in G and* $\check{\Lambda}_{\vartheta i \rightarrow \vartheta((i \bmod x)+1)}$ *is computed using Equation (3.1).*

Proof. In a cyclic path $\vartheta = \{A_{\vartheta 1} \leftrightarrow A_{\vartheta 2} \leftrightarrow \cdots \leftrightarrow A_{\vartheta x} \leftrightarrow A_{\vartheta 1}\} \in \mathcal{V}$ and assuming an arbitrary scaling factor $s_\vartheta \geq \check{s}$, the *earliest start time* $S_{\vartheta x}$ of task $\tau_{\vartheta x}$ corresponding to actor $A_{\vartheta x}$, when $D_i = C_i$, $\forall \tau_i \in \Gamma$, can be computed by considering task $\tau_{\vartheta(x-1)}$ corresponding to actor $A_{\vartheta(x-1)}$, that is a predecessor actor of actor $A_{\vartheta x}$, using Equation (3.2) as follows:

$$S_{\vartheta x} = S_{\vartheta(x-1)} + C_{\vartheta(x-1)} + \Lambda_{\vartheta(x-1) \rightarrow \vartheta x}.$$

Now, by recursively computing $S_{\vartheta(x-1)}$ and substituting it in the above equation, the *earliest start time* $S_{\vartheta x}$ of actor $A_{\vartheta x}$ is:

$$S_{\vartheta x} = S_{\vartheta 1} + \sum_{i=1}^{x-1} C_{\vartheta i} + \sum_{i=1}^{x-1} \Lambda_{\vartheta i \rightarrow \vartheta(i+1)}. \tag{3.12}$$

Due to the edge from actor $A_{\vartheta x}$ to actor $A_{\vartheta 1}$, the *start time* $S_{\vartheta 1}$ of task $\tau_{\vartheta 1}$ corresponding to actor $A_{\vartheta 1}$ is constrained by Equation (3.2) as follows:

$$S_{\vartheta x} + C_{\vartheta x} + \Lambda_{\vartheta x \rightarrow \vartheta 1} \leq S_{\vartheta 1}. \tag{3.13}$$

By using Equation (3.4) (Lemma 3.5.1) and Equation (3.12) in Equation (3.13), we have

$$S_{\vartheta 1} + \sum_{i=1}^{x} C_{\vartheta i} + \frac{s_\vartheta}{\check{s}} \cdot \sum_{i=1}^{x} \check{\Lambda}_{\vartheta i \rightarrow \vartheta((i \bmod x)+1)} \leq S_{\vartheta 1}$$

$$\Leftrightarrow \sum_{i=1}^{x} C_{\vartheta i} + \frac{s_\vartheta}{\check{s}} \cdot \sum_{i=1}^{x} \check{\Lambda}_{\vartheta i \rightarrow \vartheta((i \bmod x)+1)} \leq 0. \tag{3.14}$$

Equation (3.14) holds only if $\sum_{i=1}^{x} \check{\Lambda}_{\vartheta i \rightarrow \vartheta((i \bmod x)+1)} < 0$, because $\sum_{i=1}^{x} C_{\vartheta i}$, $\check{s}$, and s_ϑ are positive numbers by definition and we can always select sufficiently large scaling factor $s_\vartheta \geq \check{s}$. ■

3.5.2 Deriving Period, Earliest Start Time, and Deadline of Tasks

Recall that under our GSPS framework, every actor A_i in a **cyclic** CSDF is converted to a CDP task $\tau_i = (C_i, T_i, S_i, D_i)$. Therefore, in this section, we derive the period, deadline, and earliest start time of each task τ_i corresponding to an actor A_i in a **cyclic** (C)SDF graph scheduled in strictly periodic fashion, if such schedule exists according to Theorem 3.5.1.

(a) Period: Considering Equation (3.14), the minimum scaling factor s_ϑ that satisfies Equation (3.14) is:

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

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 corresponding to actors is:

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

Then, using Equation (2.12) and the above computed scaling factor s, the periods of the tasks corresponding to actors can be derived.

(b) Deadline: Since the number of processors needed to schedule CDP tasks depends on the total density δ_Γ of the task set Γ [29], our objective to derive the deadline of the tasks corresponding to actors is to minimize δ_Γ in order to minimize the number of processors. Therefore, we formulate our optimization problem as follows:

$$\text{Minimize} \quad \delta_\Gamma = \sum_{\tau_i \in \Gamma} \frac{C_i}{D_i} \tag{3.15a}$$

$$\text{subject to:} \quad S_i + D_i - S_j \leq -\Lambda_{i \to j} \quad \forall E_u = (A_i, A_j) \in \mathcal{E} \tag{3.15b}$$

$$-D_i \leq -C_i,\ D_i \leq T_i \quad \forall \tau_i \in \Gamma \tag{3.15c}$$

where Equation (3.15a) is the objective function and D_i is an optimization variable. In addition, Equations (3.15b) are the constraints given by Equation (3.2), and Equations (3.15c) bound all optimization variables in the objective function by the WCET C_i and period T_i derived in Section 3.5.2(a). S_i and S_j are implicit variables which are not in the objective function Equation (3.15a), but still need to be considered in the optimization procedure.

(c) Earliest Start Time: To derive the earliest start times of the tasks corresponding to actors, we use the derived deadline of the tasks corresponding to

actors in Section 3.5.2(b) in the following optimization problem:

$$\text{Minimize} \quad \sum_{\tau_i \in \Gamma} S_i \tag{3.16a}$$

$$\text{subject to:} \; S_i - S_j \leq -\Lambda_{i \to j} - D_i \quad \forall E_u = (A_i, A_j) \in \mathcal{E} \tag{3.16b}$$

$$-S_i \leq 0 \quad \forall \tau_i \in \Gamma \tag{3.16c}$$

where Equation (3.16a) is the objective function and S_i is an optimization variable. In addition, Equations (3.16b) are the constraints given by Equation (3.2), and Equations (3.16c) bound all optimization variables in the objective function to be greater or equal to zero. Given that all variables in both problems Equations (3.15) and (3.16) are integers and both the objective functions and the constraints are convex, the problems are integer convex programming problems [56]. To solve the problems in Equations (3.15) and (3.16), we used CVX [38,39], a package for specifying and solving convex programs.

3.6 Experimental Evaluation

In this section, we present experiments to evaluate our GSPS framework proposed in Section 3.5. As explained earlier, our GSPS framework enables the application of many hard real-time scheduling algorithms [29], 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) [18] and self-timed scheduling (STS) [85], 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 3.1. In this table, $|\mathcal{A}|$ and $|\mathcal{E}|$ denote the number of actors 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

Table 3.1: *Benchmarks used for evaluation.*

Application	$\lvert\mathcal{A}\rvert$	$\lvert\mathcal{E}\rvert$	Source
Modem	16	35	[2]
MP3 playback	4	4	
MP3 Decoder	15	21	[87]
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	[49]
TDS-CDMA receiver	16	25	[60]
Long Term Evolution (LTE)	10	15	[76]
Echo	38	82	[18]

given in Table 3.2. The throughput, latency, and buffer sizes of the applications under our GSPS, denoted by $\mathcal{R}_{\text{GSPS}}$, $\mathcal{L}_{\text{GSPS}}$, and $\mathcal{M}_{\text{GSPS}}$, are computed using Equations (2.15), (2.19), and (2.18) and given in columns 2, 3, and 4 in Table 3.2, respectively. Columns 7 and 10 show the ratio between the throughput of our GSPS and PS and STS, respectively. Looking at column 7, we can see that our GSPS can achieve the same throughput 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 largest difference is in the case of Echo. This is mainly because, our GSPS schedules all the phases of an actor in a CSDF graph as jobs of a periodic task, where different job release of the task corresponds to one of the phases of the actor. 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 actor 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 global and partitioned First-Fit Increasing Deadlines EDF (FFID-EDF) [29] schedulers by using Equation (2.10) and Equation (2.11), denoted with $\check{m}$ and $\check{m}_{\text{PAR}}$ in Table 3.2, respectively. However, for PS, the calculation of the number of processors was not considered in [18], and for STS, finding the minimum number of processors requires complex

Table 3.2: *Comparison of different scheduling frameworks.*

Application	GSPS					PS [18]			STS [85]		
	$\mathcal{R}_{\text{GSPS}}[\frac{1}{\text{t.u.}}]$	$\mathcal{L}_{\text{GSPS}}$[t.u.]	$\mathcal{M}_{\text{GSPS}}$[Tkn]	$\hat{m}$	$\hat{m}_{\text{PAR}}$	$\frac{\mathcal{R}_{\text{GSPS}}}{\mathcal{R}_{\text{PS}}}$	$\frac{\mathcal{L}_{\text{GSPS}}}{\mathcal{L}_{\text{PS}}}$	$\frac{\mathcal{M}_{\text{GSPS}}}{\mathcal{M}_{\text{PS}}}$	$\frac{\mathcal{R}_{\text{GSPS}}}{\mathcal{R}_{\text{STS}}}$	$\frac{\mathcal{L}_{\text{GSPS}}}{\mathcal{L}_{\text{STS}}}$	$\frac{\mathcal{M}_{\text{GSPS}}}{\mathcal{M}_{\text{STS}}}$
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

design space exploration to find the best allocation which delivers the maximum achievable throughput [83]. 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 give 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 Equations (3.15), 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 Equations (3.16) 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.

3.7 Conclusions

In this chapter, 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 streaming applications, that strictly periodic scheduling is capable of delivering equal or comparable throughput to existing approaches for the majority of the applications we experimented with.