Improved hard real-time scheduling and transformations for embedded Streaming Applications

Spasic, J.

Chapter 3

Hard Real-Time Scheduling Framework

Jelena Spasic, Di Liu, Emanuele Cannella, Todor Stefanov, "On the Improved Hard Real-Time Scheduling of Cyclo-Static Dataflow", *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 15, Issue 4, Article 68, August 2016.

Jelena Spasic, Di Liu, Emanuele Cannella, Todor Stefanov, "Improved Hard Real-Time Scheduling of CSDF-modeled Streaming Applications", *In Proceedings of the IEEE/ACM/IFIP International Conference on HW/SW Codesign and System Synthesis (CODES+ISSS'15)*, pp. 65–74, Amsterdam, The Netherlands, October 4-9, 2015.

IN this chapter, we present a scheduling approach to provide timing guarantees for streaming applications mapped on MPSoCs. In particular, we describe in more detail our solution approach, introduced in Section 1.4, to the research problem, **Problem 1**, described in Section 1.3.

The remainder of this chapter is organized as follows. Section 3.1 continues the introduction by describing in more detail the addressed research problem. 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. Then, we give an example in Section 3.4 to motivate the need for our scheduling approach. The proposed scheduling approach is described in Section 3.5. The experimental evaluation of our proposed scheduling approach is presented in Section 3.6. The concluding discussion is given in Section 3.7.

3.1 Problem Statement

Recently, the authors in [BS13] proposed a framework to schedule streaming applications modeled as acyclic CSDF graphs as a set of real-time periodic tasks on an MPSoC platform. They also derive the minimum number of processors needed to schedule the applications on a platform. However, in that framework, the authors use one and the same worst-case execution time (WCET) value for all execution phases of a task in the CSDF graph, although a task in the CSDF graph may have a different WCET value for every phase. The authors simply take and use the maximum WCET value among the WCET values for all phases of a task. By doing this, the cyclically changing execution nature of an application modeled by the CSDF model is hidden, which leads to underestimation of the throughput, overestimation of the latency, and underutilization of processors. In another recent work [BMKdD13], the authors proposed a framework to evaluate a lower bound of the maximum throughput of a periodically scheduled CSDF-modeled application. However, the authors do not provide a method to determine the number of processors required for scheduling the application. Moreover, their approach does not ensure temporal isolation among applications, that is, the schedule of applications has to be recalculated once a new application comes in the system and hence it may be possible that the previously calculated throughput of an application can no longer be reached. Thus, in this chapter, we investigate the possibility to **schedule streaming applications modeled as acyclic CSDF graphs as real-time periodic tasks** on an MPSoC platform while **considering different WCET values for task's phases** in an acyclic CSDF graph and providing temporal isolation of applications and hard real-time guarantees.

3.2 Contributions

In order to address the problem described in Section 3.1, we propose a scheduling approach which contributions are summarized as follows:

- We prove that considering a different WCET value for each execution phase of a task we can convert the execution phases of each task in an acyclic CSDF graph to strictly periodic real-time tasks. This enables the use of many hard real-time scheduling algorithms to schedule such tasks with a certain guaranteed throughput and latency. (Theorem 3.5.2)
- We prove that our scheduling approach gives equal or higher throughput than the existing hard real-time scheduling approach for acyclic CSDF graphs. (Theorem 3.5.3)

- We propose a method for reducing the latency of an acyclic CSDF graph scheduled as a set of strictly periodic real-time tasks. (Section 3.5.5)
- We show, on a set of real-life streaming applications, that scheduling each execution phase of a CSDF task as a strictly periodic task and considering different WCET per phase lead not only to tighter guarantee on the throughput of an application but also to better utilization of processor resources. (Section 3.6.1)
- We demonstrate, on a set of real-life streaming applications, that the total time required by our approach to derive the schedule of the tasks, to calculate the minimum number of processors needed to schedule the tasks, and to calculate the size of communication buffers between tasks is comparable to the time required by the existing hard real-time scheduling approach for CSDF graphs. In addition, we show that the total time needed by our approach is much shorter in comparison to the existing periodic scheduling and self-timed scheduling approaches for CSDF graphs. (Section 3.6.2)
- We show, on a set of real-life streaming applications, that the latency of the applications scheduled by our scheduling approach can be reduced by our proposed latency reduction method in most cases to the desirable latency values while keeping higher or equal application throughput and requiring equal or smaller number of processors in comparison to the existing scheduling approaches. (Section 3.6.3)

Note that by considering acyclic CSDF graphs, our solution approach is applicable to many streaming applications as it has been shown in [TA10] that around 90% of streaming applications can be modeled as acyclic SDF graphs.

3.3 Related Work

Research on scheduling of streaming applications modeled by parallel MoCs has been active for a long period of time. Below, we compare our approach with some of the existing hard real-time scheduling approaches for streaming applications and with the scheduling approaches which do not provide hard real-time guarantees but are similar to our approach.

[HGWB13] proposes a two parameter (σ, ρ) workload characterization to reduce the difference between the worst-case throughput, determined by the analysis, and the actual throughput of the application. They consider different execution times for task's phases and then the average worst-case execution time is used to improve the minimum guaranteed throughput/latency. Similar to them, we consider different execution times for task's phases in a CSDF

graph. But in contrast to them, we convert task's phases to classical periodic hard real-time tasks, which allows us to calculate the minimum number of processors required to guarantee certain throughput and latency in a fast and analytical way for global scheduling and in a polynomial time for partitioned scheduling by using our algorithm given in Section 3.5.6.

In [BTV12], the authors propose an analysis framework for hard real-time applications modeled as Affine Dataflow (ADF) graphs. The actors in an ADF graph are scheduled as periodic tasks. The ADF model proposed in [BTV12] extends the CSDF model and hence, is more expressive than the CSDF. However, in their approach only one value is considered as the WCET value of a task, while we consider a different WCET value per each phase of a task, thereby efficiently exploiting the cyclic nature of the CSDF model and providing a tighter throughput guarantee.

[BMKdD13] proposes a framework to derive the maximum throughput of a CSDF graph under a periodic schedule and to calculate the buffer sizes in the graph with a throughput constraint. Both problems are represented as Linear Programming (LP) problems and solved approximately. Similar to our work, their work considers different execution times for each phase of a task. However, it is not explicitly given in [BMKdD13] how to compute the number of processors needed to schedule the graph according to the derived schedule. One possible way is to look at the derived schedules and find the maximum number of active tasks at any given point in time. However, this procedure has an exponential time complexity in the worst case. In contrast, in our case the conversion of CSDF task's phases to classical periodic hard real-time tasks enables fast and analytical calculation of the minimum number of processors for global scheduling of the tasks, and a polynomial time derivation of the number of processors for partitioned scheduling by using our algorithm given in Section 3.5.6.

The closest to our work, in terms of scope of work and methods proposed to schedule streaming applications modeled as acyclic CSDF graphs, is the work in [BS13]. The authors in [BS13] convert each task in a CSDF graph to a periodic task by deriving parameters such as period and start time. Then they use hard real-time schedulability analysis to determine the minimum number of processors required to execute the derived task-set. Our approach differs from [BS13] in the following: we use different WCET values for each execution phase of a task and each phase is converted to a periodic task, while in [BS13], only one WCET value is used for a task and every execution of a task is periodic with a calculated period. By considering different WCET values for each task phase and converting each phase to a periodic task, we

can guarantee tighter throughput and better utilization of processor resources.

3.4 Motivational Example

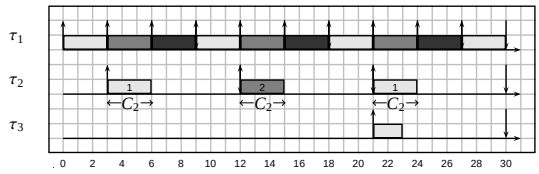
The goal of this section is to show that the real-time strictly periodic scheduling (SPS) approach [BS13] is not efficient in terms of throughput, latency and utilization of processor resources. In the framework proposed in [BS13], every actor v_i in a CSDF graph G is converted to a real-time periodic task τ_i by computing the task parameters S_i, D_i, T_i and C_i, where C_i is computed as the maximum WCET value of actor v_i, that is, $C_i = \max_{1 \leq \varphi \leq \phi_i}\{C_i(\varphi)\}$, where $C_i(\varphi)$ contains the worst-case computation, the worst-case data read and the worst-case data write times of a phase φ of actor v_i. To execute graph G strictly periodically, period T_i for each actor v_i, that is, each corresponding task τ_i, is computed as:

$$T_i = \frac{\text{lcm}(\vec{q})}{q_i} \left\lceil \frac{\max_{v_j \in V}\{C_j \cdot q_j\}}{\text{lcm}(\vec{q})} \right\rceil, \forall v_i \in \mathcal{V}, \tag{3.1}$$

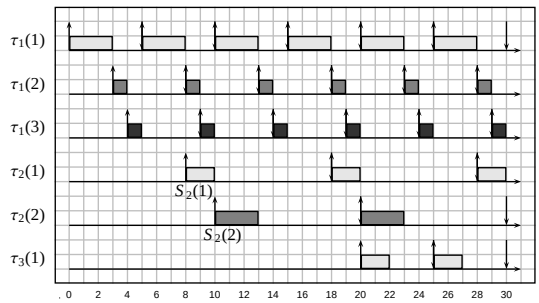
where $\text{lcm}(\vec{q})$ is the least common multiple of all repetition entries in $\vec{q}$. The strictly periodic schedule of all actors in G, given in Figure 2.1, is shown in Figure 3.1(a), under the assumption that data read and write times are 0 (for the sake of simplicity). For example, actor v_2 (task τ_2 in Figure 3.1(a)) executes periodically with the calculated period $T_2 = 9$. Note that for every actor's phase one and the same WCET value is considered, that is, for actor v_2 we have two phases 1 and 2 and the considered WCET value C_2 for each phase is $C_2 = \max\{C_2(1), C_2(2)\} = \max\{C_2^C(1), C_2^C(2)\} = \max\{2, 3\} = 3$.

To demonstrate the need of considering different WCET values of actor's phases and the drawback of strictly periodic schedule between actor phases, we analyze two different schedules of the CSDF graph G in Figure 2.1. The first schedule we consider is SPS, visualized in Figure 3.1(a). The throughput $\mathcal{R}$, latency $\mathcal{L}$ of G and the required number of processors m are given in Table 3.1 under SPS.

However, by taking one and the same value as the WCET for all execution phases of an actor, the cyclic behavior of the CSDF actors is hidden. Assume that we convert each actor v_i in G to a set of ϕ_i IDP tasks $\tau_i(1), \tau_i(2) \cdots \tau_i(\phi_i)$ considering different WCET values for each execution phase and execute them as periodic tasks. The execution schedule of such task-set is given in Figure 3.1(b). Again, here we assume that data read and write times are 0. For example, actor v_2 is converted to 2 IDP tasks $\tau_2(1)$ and $\tau_2(2)$ where each task is executed periodically with a period equal to 10. Moreover, the WCET values



(a)



(b)

Figure 3.1: *(a) The SPS and (b) ISPS of graph G in Figure 2.1.*

Table 3.1: *Throughput, latency and number of processors for G under different scheduling schemes.*

SPS			ISPS		
$\mathcal{R}$	$\mathcal{L}$	m	$\mathcal{R}$	$\mathcal{L}$	m
1/18	30	2	1/10	25	2

of the tasks $\tau_2(1)$ and $\tau_2(2)$ are not the same but $\tau_2(1)$ has WCET $C_2(1) = 2$ and $\tau_2(2)$ has WCET $C_2(2) = 3$, as the original specification in Figure 2.1.

We can see from Table 3.1 under ISPS that by scheduling G in such a way we can obtain almost 2 times higher graph throughput and shorter graph latency while resources in terms of the required number of processors are the same compared with SPS and thus, the processor resources are better utilized in the case of ISPS. This is especially important in case of a timing constraint because it may happen that the graph cannot meet the constraint when scheduled under SPS. Here the throughput and latency under ISPS are calculated by using our approach described in Section 3.5. The required number of processors for both SPS and ISPS is calculated by Equation (2.12). Moreover, the number of processors needed for partitioned scheduling in both cases is the same as the number needed for global scheduling given by Equation (2.12). We can see from the motivational example that the SPS approach from [BS13] yields to lower throughput and larger latency of a graph by using one and the same value for the WCET of each phase of an actor and by strictly periodic scheduling of all executions of the actor. Thus, different WCET values for actor phases should be considered and the constraint on strictly periodic scheduling between the actor phases should be removed.

3.5 Improved Hard Real-Time Scheduling of CSDF

In this section, we present our scheduling framework, called *improved strictly periodic scheduling* (ISPS), which enables a conversion of every actor of an acyclic CSDF graph to a set of periodic tasks. Each set of periodic tasks corresponding to an actor has as many elements as the number of phases of that actor. By taking into account the WCET value of each phase of an actor in a graph, the proposed approach computes the parameters S_i and T_i of tasks corresponding to the actor and the minimum buffer sizes of the communication channels such that ISPS is guaranteed to exist.

The proposed conversion procedure is given in Algorithm 1. First, the periods of tasks corresponding to actors are calculated in lines 1-2, explained in Section 3.5.1. Then, relative deadlines D_i of the tasks corresponding to an actor v_i are selected from the range $D_i \in [\max_{1\leq\varphi\leq\phi_i}\{C_i(\varphi)\}, \check{T}_i]$, lines 3-6. For example, if one wants to minimize the number of processors needed to schedule the converted tasks, he/she should select relative deadlines of the tasks to be equal to the corresponding task periods, that is, $D_i = \check{T}_i$. On the other hand, if one wants to reduce the graph latency, he/she should use our latency reduction method proposed in Section 3.5.5. The start times for each task-set corresponding to an actor are computed in lines 7-12, for details see Section 3.5.2. Finally, the buffer sizes of the communication channels are derived in lines 13-14, for details see Section 3.5.3.

3.5.1 Deriving Periods of Tasks

The first step in constructing the ISPS of a CSDF graph is to derive the valid period for each periodic task corresponding to a phase of an actor in the graph. To calculate the periods, we introduce the following definitions:

Definition 3.5.1. For each actor v_i in an acyclic CSDF graph G, the **WCET sequence** $C_i = [C_i(1), C_i(2), \cdots, C_i(\phi_i)]$, represents the sequence of the WCET values, measured in time units, for each execution phase of v_i. The WCET value $C_i(\varphi)$ for a phase φ is given by:

$$C_i(\varphi) = \Big(C^R \cdot \sum_{e_r \in \text{in}(v_i)} y_i^r(\varphi)\Big) + C_i^C(\varphi) + \Big(C^W \cdot \sum_{e_w \in \text{out}(v_i)} x_i^w(\varphi)\Big), \qquad (3.2)$$

where C^R represents the platform-dependent worst-case time needed to read a single token from an input channel e_r from the set of input channels in(v_i) of actor v_i; analogously, C^W is the worst-case time needed to write a single token to an output channel e_w from the set of output channels out(v_i) of v_i;

Algorithm 1: Procedure to convert a CSDF graph to a set of periodic tasks.

Input: A CSDF graph $G = (\mathcal{V}, \mathcal{E})$.
Output: For each actor $v_i \in \mathcal{V}$, a set of periodic tasks $\mathcal{T}_{v_i} = \{\tau_i(1), \cdots, \tau_i(\phi_i)\}$, and for each channel $e_u \in \mathcal{E}$, the size of the buffer b_u.

for *actor* $\tau_i \in \mathcal{V}$ **do**
  Compute the minimum common period $\check{T}_i$ by using Equation (3.5);
for *actor* $v_i \in \mathcal{V}$ **do**
  Select deadline D_i, where $D_i \in [\max_{1 \leq \varphi \leq \phi_i}\{C_i(\varphi)\}, \check{T}_i]$;
  for *phase* φ *of* v_i, $1 \leq \varphi \leq \phi_i$ **do**
    $\tau_i(\varphi) = (0, C_i(\varphi), D_i, \check{T}_i)$;
for *actor* $v_i \in \mathcal{V}$ **do**
  Compute the start time of the first phase $S_i(1)$ by using Equation (3.11);
  $\tau_i(1) = (S_i(1), C_i(1), D_i, \check{T}_i)$;
  for *phase* φ *of* v_i, $2 \leq \varphi \leq \phi_i$ **do**
    Compute the start time of the φth phase $S_i(\varphi)$ by using Equation (3.9);
    $\tau_i(\varphi) = (S_i(\varphi), C_i(\varphi), D_i, \check{T}_i)$;
for *communication channel* $e_u \in \mathcal{E}$ **do**
  Compute the buffer size b_u by using Equation (3.17);

$y_i^r(\varphi)$ and $x_i^w(\varphi)$ is the number of tokens read from e_r and written to e_w by v_i, respectively, during its execution phase φ; and $C_i^C(\varphi)$ is the worst-case computation time of v_i in its phase φ.

Definition 3.5.2. For each actor v_i in an acyclic CSDF graph G, the **maximum WCET value** MC_i is given by $MC_i = \max_{1 \leq \varphi \leq \phi_i}\{C_i(\varphi)\}$.

Definition 3.5.3. For an acyclic CSDF graph G, an **aggregated execution vector** $\vec{AC}$, where $\vec{AC} \in \mathbb{N}^N$, represents the aggregated WCET values of the actors in G and its elements are given by $AC_i = \sum_{\varphi=1}^{\phi_i} C_i(\varphi)$, where $C_i(\varphi)$ is the WCET value of v_i's phase φ.

Each actor $v_i \in V$ in graph G is converted to a periodic task set $\mathcal{T}_{v_i} = \{\tau_i(1), \cdots, \tau_i(\phi_i)\}$.

Definition 3.5.4. A task $\tau_i(\varphi)$ corresponding to a phase φ of an actor v_i, where $1 \leq \varphi \leq \phi_i$, in an acyclic CSDF graph G is a **strictly periodic task** iff the time period between any two consecutive firings of that task is constant.

All tasks belonging to a periodic task set $\mathcal{T}_{v_i}$ corresponding to an actor v_i have the same period T_i, which we call *common period*.

Definition 3.5.5. For an acyclic CSDF graph G, a **common period vector** $\vec{T}$, where $\vec{T} \in \mathbb{N}^N$, represents the periods, measured in time units, of periodic task-sets corresponding to actors in G. $T_i \in \vec{T}$ is common period of periodic task-set corresponding to actor $v_i \in V$. $\vec{T}$ is given by the solution to both

$$r_1 T_1 = r_2 T_2 = \cdots = r_{N-1} T_{N-1} = r_N T_N \tag{3.3}$$

and

$$\vec{T} - \vec{AC} \geq \vec{0}, \tag{3.4}$$

where $r_i \in \vec{r}$, and $\vec{r}$ is the aggregated repetition vector introduced in Section 2.1.1.

Lemma 3.5.1. *For an acyclic CSDF graph G, the minimum common period vector $\vec{\check{T}}$ is given by:*

$$\check{T}_i = \frac{\text{lcm}(\vec{r})}{r_i} \left\lceil \frac{\max_{v_j \in V}\{AC_j \cdot r_j\}}{\text{lcm}(\vec{r})} \right\rceil, \forall v_i \in V, \tag{3.5}$$

where $lcm(\vec{r})$ is the least common multiple of all phase repetition entries in $\vec{r}$.

Proof. The minimum common period vector $\vec{\check{T}}$ that solves Equation (3.3) is given by:

$$\check{T}_i = \text{lcm}\{r_1, r_2, \cdots, r_N\}/r_i, \forall v_i \in V.$$

Inequality (3.4) can be re-written as:

$$c\check{T}_1 \geq AC_1, c\check{T}_2 \geq AC_2, \cdots, c\check{T}_N \geq AC_N, c \in \mathbb{N}. \tag{3.6}$$

Further, Inequality (3.6) can be re-written as:

$$c \geq AC_1 r_1 / \text{lcm}(\vec{r}), \cdots, c \geq AC_N r_N / \text{lcm}(\vec{r}). \tag{3.7}$$

From Inequality (3.7), it follows that c is greater than or equal to $\max_{v_j \in V}\{AC_j r_j\} / \text{lcm}(\vec{r})$. However, $\max_{v_j \in V}\{AC_j r_j\} / \text{lcm}(\vec{r})$ is not always guaranteed to be an integer. Because of that, the value is rounded up by taking its ceiling. Thus, the minimum common period vector which satisfies both Equation (3.3) and Inequality (3.4) is given by Equation (3.5). ■

For the CSDF graph in Figure 2.1, the derived minimum common periods in time units are $[\check{T}_1, \check{T}_2, \check{T}_3] = [5, 10, 5]$.

Theorem 3.5.1. *For any acyclic CSDF graph G, where G has L topological sort levels, a periodic schedule exists with start times $S_i(\varphi)$, $\varphi \in [1, \phi_i]$, for each level-k actor $v_i \in V$ given by:*

$$S_i(1) = (k-1) \cdot 2\alpha \tag{3.8}$$

and

$$S_i(\varphi) = S_i(\varphi - 1) + C_i(\varphi - 1),\ \forall \varphi \in [2, \phi_i], \tag{3.9}$$

such that every phase of an actor $v_i \in V$ is strictly periodic with a constant period $T_i \in \vec{T}$ and every communication channel $e_u = (v_i, v_j) \in E$ has a bounded buffer capacity, given by:

$$b_u = (l - k + 1) \cdot 2X_i^u(\phi_i r_i), \tag{3.10}$$

where $\alpha = r_1 T_1 = \cdots = r_N T_N$ is the iteration period of G, v_i is level-k actor and v_j is level-l actor, $l \geq k$.

Proof. Let us assume that graph G is partitioned into L levels in a way similar to topological sort. In that way, all input actors belong to level-1, the actors from level-2 have all immediate predecessors in level-1, the actors from level-3 have immediate predecessors in level-2 and can also have immediate predecessors in level-1, and so on. The graph iteration period is $\alpha = r_1 T_1 = \cdots = r_N T_N$. During the iteration period each phase of v_i is executed r_i times. Assume that the first phase of level-1 actors starts at time $t = 0$. Other phases of an actor are scheduled to be fired as soon as the WCET of the previous phase elapses. Recall that every actor v_i in graph G is converted to a set of strictly periodic tasks where a task corresponds to a phase of the actor. Consider now an actor from level-1, denoted as v_1. By time $t = \alpha + S_1(\phi_1)$, the last phase of v_1 will finish its r_1th execution, where $S_1(\phi_1)$ is the start time of the last phase of v_1. Level-1 actors will complete a whole iteration by time $t_1 = \alpha + \max_{v_i \in \text{level-1}}\{S_i(\phi_i)\}$ and will continue executing their second iteration. According to Equation (2.4), level-1 actors will produce enough data on all channels to level-2 actors by time t_1 such that level-2 actors can execute a whole iteration if their first phases are started at t_1, at the earliest. Let us start the first phases of level-2 actors at time $t = 2\alpha$ and all the other phases of a level-2 actor one after the other. Similarly, by time $t_2 = 3\alpha + \max_{v_i \in \text{level-2}}\{S_i(\phi_i) - S_i(1)\}$, level-3 actors will have enough data to execute one iteration. Thus, starting the first phases of level-3 actors at time $t = 4\alpha$ guarantees that the actors can execute a whole iteration. By repeating the same procedure to the actors of the last level, level L, (by starting their first phases at $t = (L - 1) \cdot 2\alpha$ and all the other phases as soon as the WCET of previous phase elapses), we obtain an overlapping schedule σ where all actors execute their corresponding iterations. In the constructed schedule, the first phase of an actor v_j corresponding to a level-i will start execution at time $t = (i - 1) \cdot 2\alpha$ and once it starts it will be fired every T_j time units. The other phases start their executions one after the other and all within period T_j. Once started, each phase is re-executed every T_j time units.

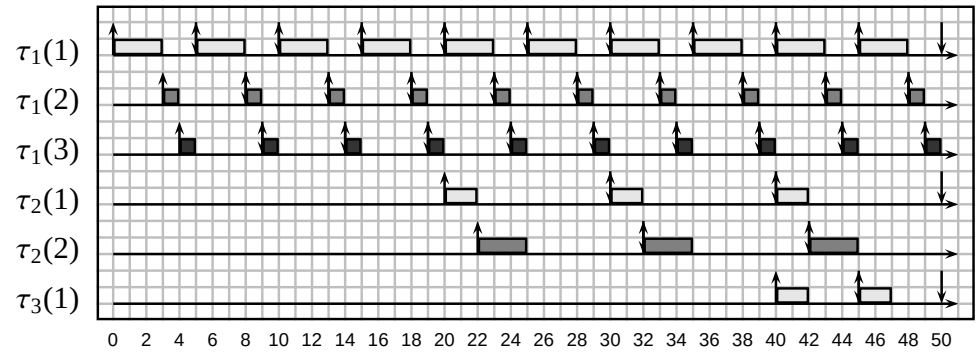


Figure 3.2: *The periodic schedule σ for the CSDF graph G shown in Figure 2.1.*

Now, we will prove that the constructed schedule executes with bounded buffers. The longest delay which may happen between production and consumption of data tokens is in case when there is a dependency e_u between the first iteration of a level-1 actor and the first iteration of a level-L actor. In this case the delay is equal to $(L-1)\cdot 2\alpha$ and during that period the level-1 actor will produce on channel e_u at most $(L-1)\cdot 2X_1^u(\phi_1 r_1)$ data tokens, where $X_1^u(\phi_1 r_1)$ is the number of tokens produced during $\phi_1 r_1$ executions of the level-1 actor. However, starting from $L\cdot 2\alpha$ level-1 and level-L execute in parallel, so we should increase the buffer size by $2X_1^u(\phi_1 r_1)$ which then becomes $L\cdot 2X_1^u(\phi_1 r_1)$. We can now use the methodology described above to determine the buffer size of each communication channel in a graph: each channel $e_u \in E$, connecting a level-i source actor v_k and a level-j destination actor ($j \geq i$) will store according to schedule σ at most:

$$b_u = (j-i+1)\cdot 2X_k^u(\phi_k r_k)$$

tokens. Thus, an upper bound on the buffer sizes exists. ■

For the example graph G given in Figure 2.1, actors in G are grouped into 3 levels such that v_1 is level-1 actor, v_2 level-2 and v_3 is level-3 actor. The calculated graph iteration period α is equal to 10. The periodic schedule resulting from Theorem 3.5.1, namely schedule σ, is depicted in Figure 3.2.

3.5.2 Deriving the Earliest Start Time of Actor's First Phase

In order to represent an actor of a CSDF graph as a set of strictly periodic tasks, in Theorem 3.5.1 we already introduced the start times of phases of the actors corresponding to different levels. However, although start times given

by Equation (3.9) are minimal relative to the start time of the corresponding first phase $S_i(1)$, start times $S_i(1)$ given by Equation (3.8) are not minimal. Minimizing the start times is very important because it has a direct impact on the latency of the graph and the buffer sizes of the communication channels. Therefore, the earliest (minimal) start times of actor's first phase $S_i(1)$ are derived below.

We derive the earliest start times assuming that the token production happens as late as possible (at the deadlines) and the tokens consumption happens as early as possible (at the beginning of execution of each phase).

Lemma 3.5.2. *For an acyclic CSDF graph G, the earliest start time of the first phase of an actor $v_j \in V$, denoted $S_j(1)$, under ISPS is given by:*

$$S_j(1) = \begin{cases} 0 & \text{if } prec(v_j) = \emptyset \\ \max_{v_i \in prec(v_j)}\{S_{i\to j}(1)\} & \text{if } prec(v_j) \neq \emptyset \end{cases} \tag{3.11}$$

where $prec(v_j)$ is the set of predecessors of v_j, and $S_{i\to j}(1)$ is given by:

$$\begin{aligned} S_{i\to j}(1) = \min_{t\in[0,S_i(1)+\alpha+\Delta_i(\phi_i)]}\{t : & \operatorname*{prd^S}_{[S_i(1),\max\{S_i(1),t\}+k)}(v_i,e_u) \\ & \geq \operatorname*{cns^S}_{[t,\max\{S_i(1),t\}+k]}(v_j,e_u),\ \forall k \in [0,\alpha+\Delta_i(\phi_i)]\}, \end{aligned} \tag{3.12}$$

where $S_i(1)$ is the earliest start time of the first phase of a predecessor actor v_i, $\alpha = r_i T_i = r_j T_j$, $\Delta_i(\phi_i) = S_i(\phi_i) - S_i(1)$, $\mathsf{prd}^{\mathsf{S}}_{[t_s,t_e)}(v_i,e_u)$ is the number of tokens produced by v_i into channel e_u during the time interval $[t_s,t_e)$, and $\mathsf{cns}^{\mathsf{S}}_{[t_s,t_e]}(v_j,e_u)$ is the number of tokens consumed by v_j from channel e_u during the time interval $[t_s,t_e]$.

Proof. In Theorem 3.5.1, we have proved the existence of ISPS when the first phase of level-k actors was started at time $(k-1)\cdot 2\alpha$. According to the schedule σ, level-$(k-1)$ predecessor v_i will start the execution of its first phase at $S_i(1) = (k-2)\cdot 2\alpha$. Level-$k$ actor v_j can then start the execution of its first phase at:

$$S_j(1) = (k-1)\cdot 2\alpha = (k-2)\cdot 2\alpha + 2\alpha = S_i(1) + 2\alpha.$$

Observe now that in the proof of Theorem 3.5.1, instead of 2α we could more precisely take $\alpha + S_i(\phi_i) - S_i(1)$, because the last production of an iteration of an actor v_i will happen $\alpha + \Delta_i(\phi_i)$ time units after the start of its first phase, at the latest. Given this and taking into account all predecessors of v_j, we can write:

$$S_j(1) = \max_{v_i \in prec(v_j)}\{S_i(1) + \alpha + \Delta_i(\phi_i)\}.$$

We are now interested in starting the first phase of v_j earlier, which means we search for $S_j(1) \leq \max_{v_i \in prec(v_j)}\{S_i(1) + \alpha + \Delta_i(\phi_i)\}$, and the earliest possible $S_j(1)$ can be at the time when the application starts, which is $t = 0$. This can be written as:

$$S_j(1) = \max_{v_i \in prec(v_j)} \{S_{i \to j}(1)\} \text{ where } S_{i \to j}(1) = t', t' \in [0, S_i(1) + \alpha + \Delta_i(\phi_i)].$$

A valid start time candidate $S_{i \to j}(1)$ must guarantee that the number of tokens available on channel $e_u = (v_i, v_j)$ at any time instant $t \geq t'$ is greater than or equal to the number of consumed tokens at the same instant such that v_j can be executed as a set of strictly periodic tasks. Here, we have two cases:

Case 1: $t' \geq S_i(1)$: In order to guarantee that v_j can fire its first phase at times $t = t', t' + T_j, \cdots, t' + \alpha$ and each other phase φ as early as possible at times $t = t' + \Delta_j(\varphi), t' + \Delta_j(\varphi) + T_j, \cdots, t' + \Delta_j(\varphi) + \alpha - T_j$, where $\Delta_j(\varphi) = \sum_{l=1}^{\varphi-1} C_j(l)$, t' must satisfy:

$$\forall k \in [0, \alpha + \Delta_i(\phi_i)] : \underset{[S_i(1), t'+k)}{\mathsf{prd}^{\mathsf{S}}}(v_i, e_u) \geq \underset{[t', t'+k]}{\mathsf{cns}^{\mathsf{S}}}(v_j, e_u). \tag{3.13}$$

Thus, a valid value of t' guarantees that once v_j starts, it always finds enough data to fire for one iteration.

Case 2: $t' < S_i(1)$: This case happens when v_j consumes zero tokens in the interval $[S_i(1), t']$ or there are initial tokens on the channel. It is sufficient to check the cumulative production and consumption over the interval $[S_i(1), S_i(1) + \alpha + \Delta_i(\phi_i)]$ because by time $t = S_i(1) + \alpha + \Delta_i(\phi_i)$ both v_i and v_j are guaranteed to have finished one iteration:

$$\forall k \in [0, \alpha + \Delta_i(\phi_i)] : \underset{[S_i(1), S_i(1)+k)}{\mathsf{prd}^{\mathsf{S}}}(v_i, e_u) \geq \underset{[t', S_i(1)+k]}{\mathsf{cns}^{\mathsf{S}}}(v_j, e_u). \tag{3.14}$$

By merging Equation (3.13) and Equation (3.14) and then selecting among valid start times t' the minimum one, we obtain Equation (3.12). Start times for the tasks corresponding to the actor phases other than the first phase are obtained by adding the WCET value of the previous phase to the derived start time of the previous phase, which is given by Equation (3.9). The start times derived in such a way enable the serialized execution of tasks corresponding to actor phases, when it is needed, by careful allocation and certain scheduling algorithms, which will be explained in more detail in Section 3.5.4. ■

Note that we derive by Lemma 3.5.2 the earliest start times assuming that the tokens production happens at the deadlines and the tokens consumption

happens at the beginning of the execution of each phase. In this case, the cumulative production and the cumulative consumption functions can be computed efficiently by:

$$\underset{[t_s,t_e)}{\mathsf{prd}^{\mathsf{S}}}(v_i,e_u)=\begin{cases}X_i^u\left(\left(\left\lfloor\frac{t_e-t_s}{T_i}\right\rfloor-1+\left\lfloor\frac{\Delta}{T_i}\right\rfloor\right)\cdot\phi_i+k_1\right) & \text{if } t_e-t_s\geq T_i\\ X_i^u(k_2) & \text{if } D_i\leq t_e-t_s\leq T_i\\ 0 & \text{if } t_e-t_s<D_i\end{cases} \tag{3.15}$$

with $\Delta=(t_e-t_s) \bmod T_i+T_i-D_i$, $k_1=\max_{l\in[1,\phi_i]}\{l:\Delta \bmod T_i\geq\sum_{\varphi=0}^{l-1}C_i(\varphi)\}$, $k_2=\max_{l\in[1,\phi_i]}\{l:t_e-t_s-D_i\geq\sum_{\varphi=0}^{l-1}C_i(\varphi)\}$, and $C_i(0)=0$.

$$\underset{[t_s,t_e]}{\mathsf{cns}^{\mathsf{S}}}(v_i,e_u)=\begin{cases}Y_i^u\left(\left\lfloor\frac{t_e-t_s}{T_i}\right\rfloor+k\right) & \text{if } t_e\geq t_s\\ 0 & \text{if } t_e<t_s\end{cases} \tag{3.16}$$

with $k=\max_{l\in[1,\phi_i]}\{l:(t_e-t_s) \bmod T_i\geq\sum_{\varphi=0}^{l-1}C_i(\varphi)\}$, and $C_i(0)=0$.

For example, the derived earliest start times for phases of actor v_2 in G, shown in Figure 2.1, are $S_2(1)=8$ and $S_2(2)=S_2(1)+C_2(1)=10$, as illustrated in Figure 3.1(b).

3.5.3 Deriving Channel Buffer Sizes

Equation (3.10) in Theorem 3.5.1 shows that ISPS has bounded buffer sizes b_u. These buffer sizes b_u are sufficient but not minimal. Therefore, we want to derive the minimum buffer sizes that guarantee periodic execution of tasks corresponding to actor phases.

We want to derive the minimum buffer size such that the derived buffer size is always valid regardless of when the actor phases are actually scheduled to produce/consume during its common period. Hence, we assume that the token production happens as early as possible (at the beginning of execution of each phase) and the token consumption happens as late as possible (at the deadlines).

Lemma 3.5.3. *For an acyclic CSDF graph G, the minimum buffer size b_u of a communication channel $e_u=(v_i,v_j)$ under ISPS is given by:*

$$b_u=\max_{k\in[0,\alpha+\Delta_j(\phi_j)]}\Big\{\underset{[S_i(1),\max\{S_i(1),S_j(1)\}+k]}{\mathsf{prd}^{\mathsf{B}}}(v_i,e_u)-\underset{[S_j(1),\max\{S_i(1),S_j(1)\}+k)}{\mathsf{cns}^{\mathsf{B}}}(v_j,e_u)\Big\}, \tag{3.17}$$

where $S_i(1)$ is the earliest start time of the first phase of a predecessor actor v_i, $\alpha = r_i T_i = r_j T_j$, $\Delta_j(\phi_j) = S_j(\phi_j) - S_j(1)$, $\mathsf{prd}^{\mathsf{B}}{}_{[t_s,t_e]}(v_i, e_u)$ is the number of tokens produced by v_i into channel e_u during the time interval $[t_s, t_e]$, and $\mathsf{cns}^{\mathsf{B}}{}_{[t_s,t_e)}(v_j, e_u)$ is the number of tokens consumed by v_j from channel e_u during the time interval $[t_s, t_e)$.

Proof. Equation (3.17) tracks the maximum cumulative number of unconsumed tokens on channel e_u during one iteration of v_i and v_j. We have two cases:

Case 1: $S_j(1) \geq S_i(1)$: Here we have two intervals $[S_i(1), S_j(1))$ and $[S_j(1), S_j(1) + \alpha + \Delta_j(\phi_j)]$. During the first interval only phases of actor v_i are executing, so tokens are only produced and buffer size should be large enough to accommodate all produced tokens in that interval. During the second interval phases of both actors execute in parallel. Thus, the minimum number of tokens that needs to be stored is given by the maximum number of unconsumed tokens on e_u at any time over this interval. At time $t = S_j(1) + \alpha + \Delta_j(\phi_j)$, both v_i and v_j have completed one iteration and the number of tokens on e_u is the same as at time $t = S_j(1) + \Delta_j(\phi_j)$ [BELP96]. Due to the periodicity of v_i and v_j, their execution pattern repeats. Thus, b_u given by Equation (3.17) is the minimum buffer size which guarantees periodic execution of v_i and v_j.

Case 2: $S_j(1) < S_i(1)$: Here we have three intervals $[S_j(1), S_i(1))$, $[S_i(1), S_j(1) + \alpha + \Delta_j(\phi_j)]$ and $(S_j(1) + \alpha + \Delta_j(\phi_j), S_i(1) + \alpha + \Delta_j(\phi_j)]$. During the first interval there is no production nor consumption or there are initial tokens on the channel, and hence b_u during that interval is equal to the number of initial tokens. During the second interval phases of both actors execute in parallel and b_u gives the maximum number of unconsumed tokens on e_u. During the third interval phases of actor v_j executes their second iteration, again either there is no consumption, which means that e_u has to accommodate all the tokens produced during this interval or there is consumption and b_u gives the maximum number of unconsumed tokens on e_u. At time $t = S_i(1) + \alpha + \Delta_j(\phi_j)$, both v_i and v_j have completed one iteration and the number of tokens on e_u is the same as at time $t = S_i(1) + \Delta_j(\phi_j)$ [BELP96]. Due to the periodicity of v_i and v_j, their execution pattern repeats. Thus, b_u given by Equation (3.17) is the minimum buffer size which guarantees periodic execution of v_i and v_j. ■

The cumulative production and consumption functions used for the calculation of buffer sizes under the assumption of the earliest token production

and the latest token consumption can be computed efficiently by:

$$\underset{[t_s,t_e)}{\mathsf{prd}^{\mathsf{B}}}(v_i,e_u)=\begin{cases}X_i^u\left(\left\lfloor\frac{t_e-t_s}{T_i}\right\rfloor+k\right) & \text{if } t_e\geq t_s\\ 0 & \text{if } t_e<t_s\end{cases} \tag{3.18}$$

with $k=\max_{l\in[1,\phi_i]}\{l:(t_e-t_s) \bmod T_i\geq\sum_{\varphi=0}^{l-1}C_i(\varphi)\}$, and $C_i(0)=0$.

$$\underset{[t_s,t_e)}{\mathsf{cns}^{\mathsf{B}}}(v_i,e_u)=\begin{cases}Y_i^u\left(\left(\left\lfloor\frac{t_e-t_s}{T_i}\right\rfloor-1+\left\lfloor\frac{\Delta}{T_i}\right\rfloor\right)\cdot\phi_i+k_1\right) & \text{if } t_e-t_s\geq T_i\\ Y_i^u(k_2) & \text{if } D_i\leq t_e-t_s\leq T_i\\ 0 & \text{if } t_e-t_s<D_i\end{cases} \tag{3.19}$$

with $\Delta=(t_e-t_s) \bmod T_i+T_i-D_i$, $k_1=\max_{l\in[1,\phi_i]}\{l:\Delta \bmod T_i\geq\sum_{\varphi=0}^{l-1}C_i(\varphi)\}$, $k_2=\max_{l\in[1,\phi_i]}\{l:t_e-t_s-D_i\geq\sum_{\varphi=0}^{l-1}C_i(\varphi)\}$, and $C_i(0)=0$.

For the example graph G given in Figure 2.1, the calculated buffer sizes in tokens are $[b_1,b_2,b_3]=[4,15,4]$.

3.5.4 Hard Real-Time Schedulability

We give now a theorem which summarizes the presented results for our improved strictly periodic scheduling (ISPS):

Theorem 3.5.2. *For an acyclic CSDF graph G, let $\mathcal{T}_G$ be a set of periodic task sets $\mathcal{T}_{v_i}$ such that $\mathcal{T}_{v_i}$ corresponds to $v_i\in\mathcal{V}$. $\mathcal{T}_{v_i}$ consists of ϕ_i periodic tasks given by:*

$$\tau_i(\varphi)=(S_i(\varphi),C_i(\varphi),D_i,T_i),\ 1\leq\varphi\leq\phi_i, \tag{3.20}$$

where $S_i(\varphi)$ is the earliest start time of a phase φ of actor v_i given by Equation (3.11) and Equation (3.9), $C_i(\varphi)$ is the WCET value of a phase φ given by Equation (3.2), D_i is the relative deadline, $\max_{1\leq\varphi\leq\phi_i}\{C_i(\varphi)\}\leq D_i\leq T_i$, and T_i is the period of $\mathcal{T}_{v_i}$ given by Equation (3.5). $\mathcal{T}_G$ is schedulable on m processors using a hard real-time scheduling algorithm A for periodic tasks if:

1. *A is partitioned Earliest Deadline First, partitioned Rate Monotonic, partitioned Deadline Monotonic or hierarchical global hard real-time scheduling algorithm,*
2. *$\mathcal{T}_G$ satisfies the schedulability test of A on m processors,*
3. *every communication channel $e_u\in E$ has a capacity of at least b_u tokens, where b_u is given by Equation (3.17).*

Proof. According to Theorem 3.5.1, the graph is converted into strictly periodic tasks. The task set $\mathcal{T}_{v_i}$ corresponding to an actor v_i should be scheduled in a way which preserves the dependency between the actor phases. The hard real-time scheduling algorithms which can do this are partitioned Earliest Deadline First (EDF), Rate Monotonic (RM) [LL73] and Deadline Monotonic (DM) [LW82], or hierarchical [HA06], [LB03]. In case of the partitioned algorithms, tasks which correspond to phases of an actor should be allocated to the same processor and scheduled by EDF or DM because the deadlines of the phases are in the same order as the phases themselves thereby, preserving the data-dependencies between the phases, or by RM fixed priority scheduler where ties should be broken in favor of jobs arrived earlier in a system. In hierarchical scheduling a set of tasks are grouped together and scheduled as a single entity, called server task or supertask. When the entity is scheduled, one of its tasks is selected to execute according to an internal scheduling policy. Hence, the supertasks/servers are scheduled globally, while the scheduling of the tasks within a supertask/server is done locally, that is, it is analogous to scheduling on uniprocessor. By grouping the tasks which correspond to phases of an actor with data-dependent phases into a supertask/server and scheduling them by a scheduler which preserves their order (for example, EDF) the synchronization problem of such dependent tasks is solved. ■

3.5.5 Performance Analysis

Once an acyclic CSDF graph has been converted to a set of strictly periodic tasks, the calculated task parameters S_i, C_i, D_i, and T_i, where S_i is the start time of τ_i, C_i is the WCET, D_i is the deadline of τ_i, and T_i is the task period, are used for performance analysis of the graph, that is, for analysis of the graph's throughput and latency.

Throughput Analysis under ISPS

The throughput of a graph G scheduled by ISPS is given by:

$$\mathcal{R}(G) = \frac{1}{\alpha} = \frac{1}{r_i \breve{T}_i}, v_i \in \mathcal{V}, \tag{3.21}$$

where $\breve{T}_i$ is calculated by Equation (3.5). Given that during one graph iteration every actor $v_i \in \mathcal{V}$ is executed q_i times, the throughput of each actor is calculated as:

$$\mathcal{R}_i = \frac{q_i}{\alpha} = \frac{\phi_i}{\breve{T}_i}, v_i \in \mathcal{V}. \tag{3.22}$$

Theorem 3.5.3. *For any acyclic CSDF graph G scheduled by ISPS, the throughput of the graph is never less than the graph throughput when G is scheduled by SPS.*

Proof. The throughput of a graph scheduled under SPS [BS13] is $1/\alpha^{\text{SPS}} = 1/(q_i T_i^{\text{SPS}})$, $v_i \in V$. If the same graph is scheduled under our ISPS, then its throughput is $1/\alpha^{\text{ISPS}} = 1/(r_i T_i^{\text{ISPS}})$, $v_i \in V$. By using Equation (3.1) and Equation (3.5) and denoting $u = \max_{v_j \in V}\{r_j \sum_{\varphi=1}^{\phi_j} C_j(\varphi)\}$ and $w = \max_{v_j \in V}\{q_j \max_{1 \le \varphi \le \phi_j}\{C_j(\varphi)\}\}$, we can write the relation which we want to prove, $\alpha^{\text{ISPS}} \le \alpha^{\text{SPS}}$, as follows:

$$\text{lcm}(\vec{r}) \left\lceil \frac{u}{\text{lcm}(\vec{r})} \right\rceil \le \text{lcm}(\vec{q}) \left\lceil \frac{w}{\text{lcm}(\vec{q})} \right\rceil. \tag{3.23}$$

We have that $u \le w$. Given that the least common multiple of positive integer numbers can be found using prime factorization, and the relation between vectors $\vec{r} = [r_1, \cdots, r_N]^T$ and $\vec{q} = \mathbf{\Phi} \cdot \vec{r} = [\phi_1 r_1, \cdots, \phi_N r_N]^T$, we have that $\text{lcm}(\vec{q})$ is divisible by $\text{lcm}(\vec{r})$.

Finally, to prove relation (3.23) we consider the following cases (with regard to divisibility by the corresponding *lcm* term):

Case 1: workloads u and w on both sides of Inequality (3.23) are divisible by the corresponding *lcm* terms. Then by removing the ceiling operation we obtain inequality $u \le w$, which always holds.

Case 2: u is divisible by $\text{lcm}(\vec{r})$, w is not divisible by $\text{lcm}(\vec{q})$. We can represent the ceiling operation on the right-hand side as $(w + \text{lcm}(\vec{q}) - w \bmod (\text{lcm}(\vec{q})))/\,\text{lcm}(\vec{q})$. In the worst case $w \bmod (\text{lcm}(\vec{q}))$ is equal to $\text{lcm}(\vec{q}) - 1$. By putting this into Inequality (3.23) we obtain $u \le w + 1$, which holds.

Case 3: u is not divisible by $\text{lcm}(\vec{r})$, w is divisible by $\text{lcm}(\vec{q})$ (also divisible by $\text{lcm}(\vec{r})$). We can represent u and w as $k_u \,\text{lcm}(\vec{r}) + u \bmod (\text{lcm}(\vec{r}))$ and $k_w \,\text{lcm}(\vec{r})$, respectively, for some integer constants k_u and k_w, $k_u < k_w$. We represent the ceiling operation as in *Case 2*, so Inequality (3.23) becomes $u + \text{lcm}(\vec{r}) - u \bmod (\text{lcm}(\vec{r})) \le w$. Now, by putting the k_u-representation of u and k_w-representation of w, the inequality becomes $k_u + 1 \le k_w$, which is true and thus, Inequality (3.23) holds.

Case 4: workloads on both sides of Inequality (3.23) are not divisible by the corresponding *lcm* terms. Similarly to *Case 2* and *Case 3*, we can represent the ceiling operation through the modulo operation. In the worst case, we have on the right-hand side the smallest possible value for the ceiling operation which is $(w+1)/\,\text{lcm}(\vec{q})$ and this value is divisible by both $\text{lcm}(\vec{q})$ and $\text{lcm}(\vec{r})$. In the worst case we have $u = w$, which means that u also needs only 1 unit to be rounded up to a value divisible by $\text{lcm}(\vec{r})$. Thus, Inequality (3.23) becomes $w + 1 \le w + 1$, which holds. ■

Latency Analysis under ISPS

The latency of G scheduled by ISPS is given by:

$$\mathcal{L}(G) = \max_{w_{\text{in}\to\text{out}} \in \mathcal{W}} \{S_{\text{out}}(g^C_{\text{out}}) + D_{\text{out}} - S_{\text{in}}(g^P_{\text{in}})\}, \tag{3.24}$$

where $\mathcal{W}$ is the set of all paths from any input actor v_{in} to any output actor v_{out}, and $w_{\text{in}\to\text{out}}$ is one path of the set. $S_{\text{out}}(g^C_{\text{out}})$ and $S_{\text{in}}(g^P_{\text{in}})$ are the earliest start times of the first phase of τ_{out} with non-zero token consumption (phase g^C_{out}) and the first phase of v_{in} with non-zero token production (phase g^P_{in}) on a path $w_{\text{in}\to\text{out}} \in \mathcal{W}$, respectively. D_{out} is the relative deadline of v_{out}.

From Equation (3.24) we can see that the latency of a graph depends on start times and deadlines of the graph's actors. Given that actor start times are dependent on deadlines (see Section 3.5.2), in order to reduce the latency we should reduce actor deadlines, that is, we should change the token production times. However, given that reducing the deadlines increases the number of processors required to schedule the graph, we are interested in selecting the deadlines which lead to required graph latency while the number of processors needed to obtain that latency is minimized. To select deadlines properly, we devise the solution approach presented in this section that formulates the problem of selecting task deadlines under a given latency constraint while the number of processors is minimized when a CSDF graph is converted to real-time periodic tasks by using our ISPS approach as a mathematical programming problem. In order to formulate our problem as a mathematical programming problem, we need to rewrite the start time computation in a proper form.

Lemma 3.5.4. *For an acyclic CSDF graph G, the earliest start time of the first phase of an actor $\tau_j \in V$, denoted $S_j(1)$, under ISPS is given by:*

$$S_j(1) = \begin{cases} 0 & prec(v_j) = \emptyset \\ \max_{v_i \in prec(v_j)} \{S_i(1) + (S^{min}_{i\to j}(1) - S^{min}_i(1) - MC_i) + D_i\} & prec(v_j) \neq \emptyset \end{cases} \tag{3.25}$$

where $prec(v_j)$ is the set of predecessors of v_j, $S_i(1)$, MC_i, and D_i are the earliest start time of the first phase, the maximum WCET (Definition 3.5.2), and deadline of the predecessor actor v_i, respectively. $S^{min}_i(1)$ is the earliest start time of the first phase of v_i given by Equation (3.11) when $D_k = MC_k, \forall v_k \in V$, and $S^{min}_{i\to j}(1)$ is given by Equation (3.12) when $D_k = MC_k, \forall v_k \in V$.

Proof. Let us consider an arbitrary channel $e_u = (v_i, v_j)$ in a CSDF graph $G = (\mathcal{V}, \mathcal{E})$. Actor v_j starts execution of its first phase after v_i has started and

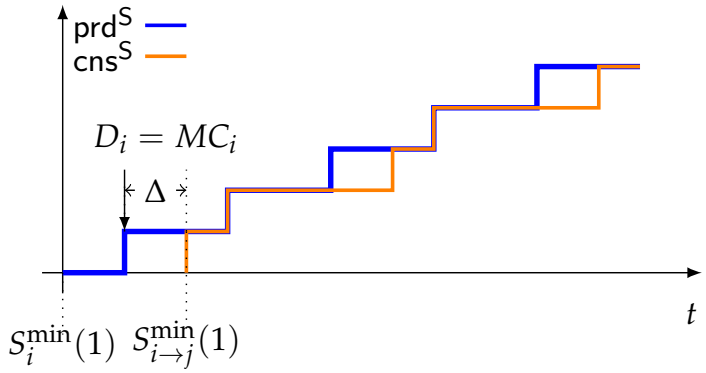


Figure 3.3: *Production and consumption curves on edge* $e_u = (v_i, v_j)$.

fired a certain number of times. This number of firings is independent from the execution speed of the actors and depends only on the production and consumption rates of v_i and v_j on e_u, where cumulative production and cumulative consumption functions are given by Equation (3.15) and Equation (3.16). Suppose that $D_k = MC_k, \forall v_k \in V$. The production ($\mathsf{prd}^{\mathsf{S}}$) and consumption ($\mathsf{cns}^{\mathsf{S}}$) curves of v_i and v_j are shown in Figure 3.3. Interval Δ in Figure 3.3 can be calculated as:

$$\Delta = S_{i\to j}^{\min}(1) - S_i^{\min}(1) - MC_i. \tag{3.26}$$

Now, suppose that $D_k > MC_k, \forall v_k \in V$. The production curve will move to the right for certain time units, and the new start time of the first phase of v_i is $S_i(1)$. If the consumption curve does not move, the relation between the production and consumption given by Equation (3.12) will be violated, that is, it will happen in some point in time that the cumulative consumption is greater than the cumulative production. This means that we have to move the consumption curve to the right by the same number of time units such that the new start time $S_{i\to j}(1)$ satisfies Equation (3.12). Hence, interval Δ will stay the same, and it is given by:

$$\Delta = S_{i\to j}(1) - S_i(1) - D_i. \tag{3.27}$$

By rewriting Equation (3.26) and Equation (3.27), we obtain:

$$S_{i\to j}(1) = S_i(1) + (S_{i\to j}^{\min}(1) - S_i^{\min}(1) - MC_i) + D_i. \tag{3.28}$$

■

We can derive from Equation (3.25) the following set of linear inequality constraints, where the number of the linear inequality constraints is equal to

the number of edges in the CSDF:

$$S_i(1) + (S^{\min}_{i\to j}(1) - S^{\min}_i(1) - MC_i) + D_i \leq S_j(1), \forall e_u \in \mathcal{E}. \tag{3.29}$$

In addition, we can rewrite Equation (3.24) as follows:

$$\mathcal{L}(G) = \max_{w_{\text{in}\to\text{out}}\in\mathcal{W}} \{S_{\text{out}}(1) + \sum_{k=1}^{g^C_{\text{out}}-1} C_{\text{out}}(k) + D_{\text{out}} - S_{\text{in}}(1) - \sum_{k=1}^{g^P_{\text{in}}-1} C_{\text{in}}(k)\}. \tag{3.30}$$

Since the number of processors needed to schedule constrained-deadline periodic (CDP) tasks depends on the total density δ_{sum} of the tasks [DB11], our objective is to minimize δ_{sum} in order to minimize the number of processors. Therefore, we formulate our optimization problem as follows:

$$\text{Minimize} \quad \delta_{\text{sum}} = \sum_{v_k\in V} \frac{AC_k}{D_k} \tag{3.31a}$$

$$\text{subject to:} \quad S_{\text{out}}(1) + D_{\text{out}} - S_{\text{in}}(1) \leq \mathcal{L} - \sum_{k=1}^{g^C_{\text{out}}-1} C_{\text{out}}(k) + \sum_{k=1}^{g^P_{\text{in}}-1} C_{\text{in}}(k), \quad \forall w_{\text{in}\to\text{out}} \in \mathcal{W} \tag{3.31b}$$

$$S_i(1) + D_i - S_j(1) \leq -(S^{\min}_{i\to j}(1) - S^{\min}_i(1) - MC_i), \quad \forall e_u \in \mathcal{E} \tag{3.31c}$$

$$-D_k \leq -MC_k, D_k \leq T_k, \quad \forall v_k \in \mathcal{V} \tag{3.31d}$$

where (3.31a) is the objective function and D_k is an optimization variable. The objective function (3.31a) has $|V|$ optimization variables and is subject to a latency constraint $\mathcal{L}$. Therefore, (3.31b) comes from (3.30). For each channel in a graph we have Equation (3.29), which can be rewritten as (3.31c). In addition, (3.31d) bounds all optimization variables in the objective function. $S_i(1)$ and $S_j(1)$ (including $S_{\text{in}}(1)$, $S_{\text{out}}(1)$) are implicit variables which are not in the objective function (3.31a), but still need to be considered in the optimization procedure. $\mathcal{L}$, g^P_{in}, g^C_{out}, $S^{\min}_{i\to j}(1)$, $S^{\min}_i(1)$, MC_k, and T_k are constants. Given that all variables are integers and both the objective function and the constraints are convex, problem (3.31) is an integer convex programming (ICP) problem [LSZ+14] which can be solved by using existing convex programming solvers, such as CVX solver [GB14].

Algorithm 2: Procedure to derive the number of processors.

Input: A CSDF graph $G = (\mathcal{V}, \mathcal{E})$, a partitioned scheduling algorithm A, an allocation heuristic $\mathcal{H}$.
Output: Number of processors m_{PAR}, task allocation *alloc*.

1 **for** *actor* v_i *in* $\mathcal{V}$ **do**
2 Compute the minimum common period $\check{T}_i$ by using Equation (3.5);
3 $u_{\text{total}} = 0$;
4 $U \leftarrow \emptyset$; (the set of allocation units, initially empty)
5 **for** *actor* $v_i \in \mathcal{V}$ **do**
6 $u_i = 0$;
7 **for** *phase* φ *of* v_i, $1 \leq \varphi \leq \phi_i$ **do**
8 $u_i(\varphi) = \frac{C_i(\varphi)}{\check{T}_i}$;
9 $u_i = u_i + u_i(\varphi)$;
10 $u_{\text{total}} = u_{\text{total}} + u_i(\varphi)$;
11 $U = U \cup u_i$;
12 $m_{\text{PAR}} = m_{\text{OPT}} = \lceil u_{\text{total}} \rceil$;
13 Reorder elements of U if required by an allocation heuristic $\mathcal{H}$;
14 **for** $u \in U$ **do**
15 $\Pi = \{\pi_1, \pi_2, \cdots, \pi_{m_{\text{PAR}}}\}$;
16 Apply bin-packing allocation heuristic $\mathcal{H}$ to u on $\pi_j \in \Pi$ and check the schedulability test of algorithm A on π_j;
17 **if** u *is not allocated to any* $\pi_j \in \Pi$ **then**
18 Allocate u on a new processor $\pi_{m_{\text{PAR}}+1}$;
19 $m_{\text{PAR}} = m_{\text{PAR}} + 1$;
20 **return** m_{PAR}, *alloc*;

3.5.6 Deriving the Number of Processors

As introduced in Section 2.2.5, by using Equation (2.12) one can compute the absolute minimum number of processors m_{OPT} needed to schedule the tasks with deadlines equal to the periods. The tasks can be scheduled on m_{OPT} if an optimal scheduling algorithm is used. The optimal scheduling algorithms are either global or hybrid, and hence, they require task migration. On the other hand, the partitioned scheduling algorithms do not require task migration. In that case the tasks are first allocated to the processors, for example by using a task partitioning heuristic, as described in Section 2.2.5, and then the tasks on each processor are scheduled using a uniprocessor scheduling algorithm.

The procedure to calculate the number of processors required for the partitioned scheduling of the task set obtained by the conversion procedure described in Section 3.5 (see Algorithm 1) is given in Algorithm 2. Algorithm 2 takes as inputs a CSDF graph G, a partitioned scheduling algorithm A

and an allocation heuristic $\mathcal{H}$. The minimum common period for each actor is calculated in lines 1-2 of the algorithm. Once the periods are calculated, then the total utilization of the converted task set and the utilization per task set corresponding to an actor are calculated in lines 3-10. Line 11 in Algorithm 2 ensures that the task set corresponding to an actor is considered as one scheduling entity, that is, one allocation unit. The absolute minimum number of processors m_{OPT} for scheduling the tasks is computed in line 12. Some allocation heuristics require a preprocessing step to be performed on the tasks before applying the heuristic. This preprocessing step is usually sorting the tasks based on some criteria, such as their utilization. That step is done in Algorithm 2 in line 13. The following lines find the number of processors and the allocation of tasks to processors. Given that m_{OPT} is the lower bound on the number of processors m_{PAR} needed by partitioned scheduling algorithms, Algorithm 2 starts with the task partitioning on m_{OPT} processors. If the tasks pass the schedulability test on all m_{PAR} processors, for example, in the case of IDP tasks and EDF scheduler the utilization of the tasks allocated to a processor is not greater than 1, then the algorithm returns m_{PAR} and the corresponding allocation of the tasks to the processors *alloc*.

Let us now analyze the time complexity of Algorithm 2 in the worst case. The first **for loop** in lines 1-2 takes linear time to calculate the minimum common period of each actor, that is, its time complexity is $O(|\mathcal{V}|)$. The second **for loop** in lines 5-11 has a nested for loop and hence, its time complexity in the worst case is given by $O(|\mathcal{V}|\phi)$, where ϕ is the maximum number of execution phases per actor, $\phi = \max_{v_i \in \mathcal{V}}\{\phi_i\}$. If the task sorting in line 13 should be performed prior to performing the task allocation, it will have $O(|\mathcal{V}|\phi \log(|\mathcal{V}|\phi))$ time complexity given that the maximum number of tasks is $|\mathcal{V}|\phi$. The **for loop** in lines 14-19 implements the allocation of the tasks to the processors by applying certain allocation heuristic and scheduling algorithm. Given that the maximum number of tasks is $|\mathcal{V}|\phi$ and the maximum number of processors needed to allocate and schedule an CSDF graph is equal to the number of actors in the graph $|\mathcal{V}|$, the time complexity of finding the number of processors m_{PAR} and the feasible task allocation is $O(|\mathcal{V}|\phi \log |\mathcal{V}|)$ [PZMA04], [BF05]. Thus, we can conclude that the running time of Algorithm 2 is polynomial and its complexity is $O(|\mathcal{V}|\phi \log |\mathcal{V}|)$ or $O(|\mathcal{V}|\phi \log(|\mathcal{V}|\phi))$ if the preprocessing step is performed.

Table 3.2: *Benchmarks used for evaluation.*

Domain	Benchmark	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	$\lvert\mathcal{T}\rvert$	Source
Medical	Heart pacemaker	4	3	67	[PMN+09]
Communication	Reed Solomon Decoder (RSD)	6	6	904	[BMMKM10]
Financial	BlackScholes	41	40	261	[BMKdD13]
Computer Vision	Disparity map	5	6	11	[ZK00]
	Pdetect	58	76	4045	[BMKdD13]
Audio processing	CELP algorithm	9	10	167	[BELP96]
	CD2DAT rate converter	6	5	22	[OH04]
	MP3 Playback	4	3	8	[WBJS07]
Image processing	JPEG2000	240	703	639	[BMKdD13]

3.6 Evaluation

We evaluate our approach in terms of its performance and time complexity by performing experiments on the benchmarks given in Table 5.2. Columns 3, 4 and 5 in Table 5.2 give for each benchmark the number of actors $|\mathcal{V}|$, the number of channels $|\mathcal{E}|$ in the corresponding CSDF graph of a benchmark, and the number of periodic tasks $|\mathcal{T}|$ obtained after converting the actors of the CSDF graph by our approach to a set of periodic tasks $\mathcal{T}$. The WCETs of actors in the benchmarks are given in clock cycles [BMKdD13] or in time units [BMMKM10], [WBJS07]. If the execution times of a benchmark are not given [BELP96], [PMN+09], [OH04], certain values based on a static analysis are assumed. The execution times of benchmark [ZK00] are obtained from the measurements of the benchmark running on a MicroBlaze processor.

Our approach is evaluated by comparison to 3 related scheduling approaches - *strictly periodic scheduling*, SPS, proposed in [BS13], *periodic scheduling*, PS, presented in [BMKdD13], and *self-timed scheduling*, STS, given in [SGB08]. We implemented our approach in Python. The SPS approach was implemented in Python within the *darts* tool-set [Bam12]. The approach in [SGB08] was implemented in C++ within the SDF^3 tool-set [SGB06]. In addition, we implemented the approach in [BMKdD13] in Python as well. We formulated both LP problems [BMKdD13] for finding the period of a graph, and for finding the start times and the buffer sizes as integer linear programming (ILP) problems, and we added the constraint that the periods of all actors in a graph have to be integers. We used CPLEX Optimization Studio [IBM12] to solve the ILP problems and mixed integer disciplined convex programming (MIDCP) in CVX [GB14] to solve our latency reduction problem. We have run all the experiments on a Dell PowerEdge T710 server running Ubuntu 11.04 (64-bit) Server OS.

3.6.1 Performance of the ISPS Approach

The main objective of the evaluation is to compare the throughput of streaming applications and the required number of processors to guarantee the throughput when scheduled by our ISPS with the throughput and the number of processors under SPS [BS13], PS [BMKdD13] and STS [SGB08]. In addition, we compare our ISPS and the other scheduling approaches in terms of application latency and memory resources needed to implement the communication channels.

We used the `sdf3analysis-csdf` tool from SDF^3 [SGB06] to obtain the maximum achievable throughput of a graph, which is the throughput under STS, and to compute the minimum buffer sizes required to achieve that throughput. Unfortunately, the `sdf3analysis-csdf` tool does not support the latency calculation and the calculation of the number of processors. Thus, we were not able to compare them with our approach. We were also not able to obtain the number of processors for a graph scheduled under PS, because the calculation of the number of processors was not considered in [BMKdD13].

Results of the performance evaluation are given in Table 3.3. We report the throughput of the output actors under ISPS, calculated by Equation (3.22), in the second column of Table 3.3. Here t.u. denotes the corresponding time unit of a benchmark. Columns 7, 12 and 15 show the ratio between the throughput of the output actors under our ISPS and SPS, PS and STS, respectively. Given that the main objective of this experiment is to evaluate the throughput of the benchmarks scheduled under ISPS and the minimum number of processors needed to obtain that throughput, our ISPS approach converts the CSDF graphs of the benchmarks to IDP tasks, which minimizes the number of processors required to schedule the benchmarks. For processor requirements in case of ISPS and SPS, we compute the minimum number of processors for IDP tasks under optimal and partitioned First-Fit Decreasing (Utilization) EDF (FFD-EDF) schedulers by using Equation (2.12) and Algorithm 2 for ISPS, and Equation (2.12) and Equation (2.16) for SPS - see columns 4, 5, 9 and 10. By comparing the throughputs under ISPS and SPS, we can see that for the majority of the benchmarks the throughput under our ISPS is higher than the corresponding throughput under SPS. Only in two cases the throughputs are the same for both schedules. The first case is MP3 Playback, which bottleneck actor (the actor with the biggest workload over one iteration period) is the same under both SPS and ISPS, and that actor has only one phase, so the influence of different WCET for actor phases on throughput cannot be seen. However, the influence can be seen from the required number of processors needed for scheduling of MP3 Playback by optimal schedulers,

which is smaller in the case of our ISPS. The second case is CD2DAT. For this benchmark lcm($\vec{q}$) and lcm($\vec{r}$) are equal and much higher than the maximum workload of actors over an iteration period for both SPS and ISPS, which leads to the same iteration period for both schedules. However, the WCET-awareness of ISPS leads to smaller number of processors. Note that if we want to schedule a task-set on smaller number of processors than the one calculated by Equation (2.12) or Equation (2.12)/Algorithm 2, we should scale up the computed actor periods by the same scaling factor [ZBS13]. Hence, to schedule CD2DAT by SPS on the same number of processors required by ISPS, we need to scale up actor periods by 2, which will lead to decrease in throughput by 2. Thus, ISPS outperforms SPS in terms of throughput when CD2DAT is scheduled on 1 processor. Benchmarks JPEG2000 and RSD can achieve much better throughput when scheduled under ISPS, but in that case they require larger number of processors to be scheduled. Note that the throughputs of these two benchmarks cannot be increased under SPS even when the number of processors is increased. If we apply the period scaling technique [ZBS13] for these two benchmarks to schedule them under ISPS on the same number of processors as required under SPS the throughput values for JPEG2000 and RSD under our ISPS are 3.93 and 11.2 times higher, as given in column 7 in parenthesis, than the corresponding values under SPS. Therefore, we can conclude that in all cases the minimum number of processors required to guarantee certain throughput under our ISPS is smaller than or equal to the minimum number of processors under SPS while the throughput under ISPS is increased in most cases, thus, processors are better utilized.

Column 12 in Table 3.3 shows the ratio of the maximum throughput of the output actors achieved by our ISPS to the maximum throughput of the output actors achieved by PS. We can see that both approaches give the same throughput for all benchmarks, which is expected given that PS schedules phases of an actor in a CSDF graph statically within a period of the actor, hence the scheduling granularity is similar between these two approaches.

Table 3.3 shows in column 15 the ratio of the maximum throughput of the output actors achieved by our approach to the absolute maximum throughput of the output actors achieved by self-timed scheduling of actor firings, which is the optimal scheduling in terms of throughput. We can see that the throughput under ISPS is equal or very close to the throughput under STS for the majority of the benchmarks. Differences in the throughput appear as a result of the ceiling operation during the calculation of actor common periods in Equation (3.5). The biggest difference is in the case of the CD2DAT benchmark. For this benchmark lcm($\vec{r}$) is much higher than the maximum

workload of actors over an iteration period, and thus, the calculated actor periods are underutilized, which leads to lower throughput. The throughput value N/A for JPEG2000 indicates that the SDF^3 tool-set [SGB06] returned an infeasible throughput (most likely related to an integer overflow).

Let us now analyze the latency and the memory resources needed to implement the communication channels of the benchmarks. The graph latency under our ISPS is calculated by Equation (3.24) for IDP tasks and shown in column 3 of Table 3.3. Column 8 shows the ratio between the graph latency under our ISPS and SPS. As we can see from columns 4, 5, 7-10 in Table 3.3: for 4 benchmarks (highlighted in the table) under ISPS we obtain higher throughput and smaller latency than under SPS without increasing the number of processors (with JPEG2000 and RSD scheduled on the same number of processors as in case of the SPS); for the other 3 benchmarks (BlackScholes, Disp. map, Pdetect) the obtained increase in throughput is less than the increase in latency on a platform with the same (or 1 less for BlackScholes under ISPS, partitioned scheduling) number of processors; for the rest 2 benchmarks we obtained the same throughput with the increase in latency, but also with the decrease in the number of processors. For the tested benchmarks, the calculated buffer sizes under ISPS are never smaller than the buffer sizes under SPS, see column 11 in Table 3.3. The highest ratio in buffer sizes between ISPS and SPS is obtained for BlackScholes and CD2DAT. However, the actual increase in communication memory resources is 215 KB and less than 1 KB, respectively, which is acceptable given the size of the memory available in modern embedded systems. Note that both latency and buffer sizes under our ISPS can be reduced by carefully selecting deadlines for individual actors (actors phases). This will be shown later in Section 3.6.3.

Column 13 gives the ratio of the maximum latency of benchmarks under our ISPS to the latency of benchmarks under PS. Although [BMKdD13] does not provide the latency calculation for their PS, we were able to extract the latency information from the start times obtained by solving the ILP problem. However, for benchmarks JPEG2000 and Pdetect we could not get a solution from the ILP solver after more than 1 day, so we could not calculate the latency for these two benchmarks. As we can see, the latency of benchmarks under ISPS is always larger than the latency under PS. As mentioned above, reducing the latency under ISPS can be done by carefully selecting deadlines for individual actors (actors phases), as shown in Section 3.6.3. Moreover, ISPS reports the maximum latency while PS reports the actual latency under a certain schedule. The ratio of the calculated buffer sizes under ISPS to the calculated buffer sizes under PS and STS is given in columns 14 and 16, respectively.

Table 3.3: *Comparison of different scheduling approaches.*

Benchmark	ISPS					SPS					PS			STS	
	$\mathcal{R}_{out}^{ISPS}[\frac{1}{t.u.}]$	$\mathcal{L}^{ISPS}$[t.u.]	m_{OPT}^{ISPS}	m_{PAR}^{ISPS}	M^{ISPS}[B]	$\frac{\mathcal{R}_{out}^{ISPS}}{\mathcal{R}_{out}^{SPS}}$	$\frac{\mathcal{L}^{ISPS}}{\mathcal{L}^{SPS}}$	m_{OPT}^{SPS}	m_{PAR}^{SPS}	$\frac{M^{ISPS}}{M^{SPS}}$	$\frac{\mathcal{R}_{out}^{ISPS}}{\mathcal{R}_{out}^{PS}}$	$\frac{\mathcal{L}^{ISPS}}{\mathcal{L}^{PS}}$	$\frac{M^{ISPS}}{M^{PS}}$	$\frac{\mathcal{R}_{out}^{ISPS}}{\mathcal{R}_{out}^{STS}}$	$\frac{M^{ISPS}}{M^{STS}}$
Pacemaker	1/10	1920	2	2	436	1.5	0.99	2	2	1.47	1	2.93	4.95	0.91	5.07
RSD	1/1080 (1/2160)	6295 (11695)	2 (1)	2 (1)	5205 (5460)	22.4 (11.2)	0.05 (0.097)	1	1	1.56 (1.63)	1	2.8	3.23	0.83	–
BlackScholes	1/3234876	24764218	16	16	260284	1.33	1.58	16	17	6.41	1	5.31	11.57	1	–
Disp. map	1/65326	382593	2	2	995520	1.03	1.13	2	2	1	1	3.18	2	1	2
Pdetect	1/2033760	36608557	11	13	13464910	1.0002	1.12	11	13	1.26	1	–	–	1	–
CELP	1/2	964	6	6	1780	1.5	0.99	6	6	1.68	1	2.24	2.38	1	–
CD2DAT	1/147	2637	1	1	116	1	3.18	2	2	4.83	1	8.88	11.6	0.17	5.09
MP3 Playback	1/25	46355	3	4	3860	1	1.84	4	4	1.48	1	2.02	1.76	0.91	1.66
JPEG2000	1/811008 (1/14598144)	27255343 (497471535)	18 (1)	18 (1)	9625878 (10006530)	70.65 (3.93)	0.02 (0.3)	1	1	1.17 (1.21)	1	–	–	N/A	N/A

Table 3.4: *Time complexity (in seconds) of different scheduling approaches.*

Benchmark	ISPS		SPS		PS		STS	
	$t_{\mathcal{R}}^{ISPS}$	$t_{S\&B}^{ISPS}$	$t_{\mathcal{R}}^{SPS}$	$t_{S\&B}^{SPS}$	$t_{\mathcal{R}}^{PS}$	$t_{S\&B}^{PS}$	$t_{\mathcal{R}}^{STS}$	$t_{S\&B}^{STS}$
Pacemaker	1.24e-05	0.056	1.31e-05	0.007	0.19	0.34	0.004	1.52
RSD	1.62e-05	4	1.74e-05	3.3	115.11	146.66	0.06	>1 day
BlackScholes	9.7e-05	1.13	9.46e-05	0.43	0.28	1.22	0.05	>1 day
Disp. map	1.36e-05	0.0014	1.69e-05	0.00087	0.027	0.055	0.004	0.01
Pdetect	0.00014	3.52	0.00013	0.65	83.64	>1 day	0.33	>1 day
CELP	2.26e-05	0.097	2.43e-05	0.029	0.56	0.95	0.01	>1 day
CD2DAT	1.67e-05	0.59	1.76e-05	0.66	0.061	0.17	0.004	108.56
MP3 Playback	1.41e-05	59.07	1.37e-05	55.87	0.021	0.034	0.004	3236.31
JPEG2000	0.00053	27.22	0.00053	3.55	0.51	>1 day	N/A	N/A

Table 3.5: *Time complexity (in seconds) for the calculation of number of processors.*

Benchmark	$t_{m_{OPT}}^{ISPS}$	$t_{m_{PAR}}^{ISPS}$
Pacemaker	4.51e-05	0.00095
RSD	0.00049	0.012
BlackScholes	0.00017	0.0077
Disp. map	1.19e-05	0.00037
Pdetect	0.0028	0.2
CELP	0.0001	0.0029
CD2DAT	1.72e-05	0.0021
MP3 Playback	9.06e-06	0.00039
JPEG2000	0.00048	0.42

Again, for benchmarks JPEG2000 and Pdetect under PS we could not get a solution from the ILP solver after more than 1 day. Similarly, for benchmarks RSD, BlackScholes, Pdetect and CELP under STS we could not get a solution for longer than 1 day. As mentioned before, value N/A for JPEG2000 indicates that SDF^3 tool-set returned an infeasible throughput, and hence the buffer sizes were not calculated. As we can see, the buffer sizes under PS and STS are always smaller than the buffer sizes under ISPS. The highest ratio in buffer sizes between ISPS and PS is obtained for BlackScholes and CD2DAT, with the actual increase in communication memory resources of 232 KB and less than 1 KB, respectively. The highest increase in buffer sizes under ISPS when compared to STS is less than 1 KB. The reason for the difference in the buffer sizes is that in both PS and STS approaches it is assumed that the production of tokens happens at the end of the actor firing, while the consumption happens at the start of the firing, while in our case (and in SPS case) the worst-case scenario is considered, that is, the production of tokens happens at the earliest possible start of the actor firing (at start times), while the consumption happens at the latest possible end of actor firing (at deadlines). Note that in an implementation of a dataflow application, data may be consumed from input channels and produced to output channels at arbitrary points in time during an actor firing. To guarantee that buffer overflow/underflow does not occur, buffer sizes have to be sufficiently large. Thus, the assumption in PS and STS limits the actual implementation of reading and writing of tokens, while the buffers calculated in our case are valid regardless of the actual point in time where reading and writing of tokens happens and thus, our approach does not limit the implementation of the reading and writing of tokens. Moreover, the buffer sizes calculated in PS and STS are valid for that specific schedule and the specific production/consumption pattern, while in the case of our ISPS the computed buffer sizes are valid for any schedule of actor firings during its period and for any production/consumption pattern during its firing.

3.6.2 Time Complexity of the ISPS Approach

In this section, we evaluate the efficiency of our ISPS approach in terms of the execution time of our algorithms to calculate the throughput of an application, and to find a schedule and buffer sizes of communication channels. The execution times are given in Table 3.4. We compare these execution times with the corresponding execution times of related approaches – SPS, PS and STS.

Let us first analyze the time needed to calculate the throughput of an application. The execution times needed to find the application throughput under ISPS, SPS, PS and STS are given in columns 2, 4, 6 and 8, respectively.

As we can see, the times spent on calculating the throughput of an application under ISPS and SPS are similar and much shorter than the time needed for solving the ILP problem to find the application throughput under PS and the time spent on finding the maximum achievable throughput of the application, that is, the throughput under STS. Thus, our approach outperforms PS and STS in terms of time required to calculate the throughput of an application. Given that in most cases ISPS gives higher throughput of an application than SPS within almost the same time, we can say that ISPS outperforms SPS as well.

Next, we compare the time needed to derive the start times of actor firings, that is, the schedule, and the buffer sizes of communication channels. These times are given in columns 3, 5, 7 and 9, for ISPS, SPS, PS and STS, respectively. By comparing the times under ISPS and SPS, we can see that both approaches find the start times and the buffer sizes within less than 4 seconds in most cases, and within a minute in two cases. Then, we compare ISPS with PS. In all but two cases ISPS is faster than PS. For those two cases (CD2DAT and MP3 Playback), the ILP problems for PS are not complex and hence they can be solved very fast. As shown in Table 3.4, ISPS gives a solution for those two cases within a second, and within a minute. On the other hand, for benchmarks Pdetect and JPEG2000 we could not get a solution from the ILP solver for PS after more than a day, while our ISPS produced the results in a couple of seconds and within a minute. By comparing to STS, our ISPS approach is always much faster. Moreover, for 4 benchmarks, we were not able to get the solution for the buffer sizing problem under STS after more than a day.

We report in Table 3.5 the execution time for calculating the minimum number of processors needed to temporally schedule the tasks, obtained by the conversion of an application by using our ISPS approach, under global optimal and partitioned FFD-EDF schedulers. In the case of global optimal scheduling, the minimum number of processors is calculated by Equation (2.12), while the calculation procedure for FFD-EDF partitioned scheduling is presented in Algorithm 2 in Section 3.5.6. As we can see, the number of processors in the case of optimal scheduling can be calculated within a millisecond for most of the benchmarks, while in the case of partitioned scheduling the calculation is done within less than 12 milliseconds for most cases and within less than 420 milliseconds in two cases. Thus, the calculation of the number of processors required to schedule an application under our ISPS is very efficient. We obtained similar times for the calculation of the number of processors under SPS and global and partitioned FFD-EDF schedulers. We could not numer-

ically compare the time complexity of our approach with regard to the PS approach because the calculation of the number of processors was not considered in [BMKdD13]. As mentioned already in Section 3.3, one possible way to find the minimum number of processors under PS is to trace the schedules but that procedure has an exponential time complexity in the worst case, whereas our Algorithm 2 for finding the minimum number of processors under ISPS has a polynomial time complexity, see Section 3.5.6. Finding the minimum number of processors under STS requires complex Design Space Exploration (DSE) procedures, with an exponential time complexity in the worst case, to find the best allocation which delivers the maximum achievable throughput. The SDF^3 tool-set used to compute the self-timed scheduling parameters does not support such design space exploration for self-timed scheduling. Thus, we could not numerically compare the time complexity of ISPS with the time complexity of STS. However, given that ISPS finds the minimum number of processors for scheduling an application in polynomial time in the worst case, as shown in Section 3.5.6, we can conclude that our ISPS is faster than STS.

3.6.3 Reducing Latency under ISPS

We have shown in the previous experiments that when compared to the SPS approach our ISPS delivers in 5 out of 9 cases larger graph latency. When compared to the PS approach, our ISPS approach always results in a graph schedule with larger graph latency. If we want to reduce graph latency under ISPS we could use the latency reduction method presented in Section 3.5.5. We would like to see how close we are in graph latency in comparison to the SPS and PS approaches after applying our latency reduction method. Therefore, in this section, we present results obtained after applying our latency reduction method introduced in Section 3.5.5 on the benchmarks given in Table 5.2. The results are given in Table 3.6. In order to apply our latency reduction method, we should set a latency constraint. To compare our ISPS approach to the SPS approach, we set the latency constraint to be equal to the graph latency obtained under SPS, $\mathcal{L}^{\mathsf{SPS}}$, and we apply our method for latency reduction. We can see from column 3 in Table 3.6 that we significantly reduce latency for the benchmarks that had higher latency under ISPS than SPS, see column 8 in Table 3.3, and that we were able to meet the latency constraint $\mathcal{L}^{\mathsf{SPS}}$ for all the benchmarks. Moreover, we see that reduction in graph latency does not influence the graph throughput, that is, the ratio of the graph throughput under ISPS to the graph throughput under SPS in column 2 is the same as the corresponding ratio given in column 7 in Table 3.3 with the period scaling technique applied for benchmarks RSD and JPEG2000 under ISPS. Columns

4 to 6 give the results on resources in terms of the number of processors required by ISPS and SPS, and the ratio between ISPS and SPS approach in buffer sizes needed to implement communication channels in a graph. We find the minimum number of processors under partitioned First-Fit Increasing Deadlines EDF (FFID-EDF) [BF05] scheduler by using Algorithm 2 for ISPS, and Equation (2.16) for SPS and FFD-EDF scheduler. We can see from columns 2 to 5 that our ISPS approach with our latency reduction method is able to schedule almost all benchmarks on the same number of processors as the SPS approach, while obtaining better graph throughput and shorter graph latency. Only in one case, for benchmark BlackScholes, our approach needs one processor more than the SPS approach. However, our approach delivers better throughput for benchmark BlackScholes than the SPS. Although the ratio between the buffer sizes under ISPS and the buffer sizes under SPS, given in column 6 in Table 3.6, is smaller than the corresponding ratio in Table 3.3, column 11, the buffer sizes under ISPS are still always bigger than the corresponding buffer sizes under SPS.

Columns 7 to 10 give the results when our latency reduction method is applied with the latency constraint dictated by the PS approach, $\mathcal{L}^{\mathsf{PS}}$. Since we could not obtain the solution from the ILP solver in the case of the PS approach after 1 day for Pdetect and JPEG2000 benchmarks – see Table 3.3, we could not provide latency and buffer sizes ratios for these two benchmarks. We can see from column 8 in Table 3.6 that we significantly reduce the latency for all benchmarks, see column 13 in Table 3.3. However, in four cases, for benchmarks BlackScholes, CELP, CD2DAT, and MP3 playback, our latency reduction method was not able to meet the latency constraint $\mathcal{L}^{\mathsf{PS}}$. The reason is that the PS approach gives the actual latency under a static schedule while our ISPS approach calculates the maximum latency for a CSDF graph converted into real-time periodic tasks. For these 4 benchmarks, column 8 gives the shortest achievable latency under ISPS obtained by applying our latency reduction method. The ratio of the graph throughput under ISPS to the graph throughput under PS is given in column 7 and it is the same as the corresponding ratio given in column 12 in Table 3.3. We report in column 9 the minimum number of processors under ISPS and FFID-EDF found by Algorithm 2. We can see that the number of processors needed by all the benchmarks with reduced latency under ISPS is higher than the corresponding number of processors given in Table 3.3, column 5, which is expected. The number of processors for a graph scheduled under PS is not given because the calculation of the number of processors was not considered in [BMKdD13]. Although the ratio between the buffer sizes under ISPS and

the buffer sizes under PS, given in column 10 in Table 3.6, is smaller than the corresponding ratio in Table 3.3, column 14, the buffer sizes under ISPS are still always bigger than the buffer sizes under PS. As explained previously, the reason for the difference in the buffer sizes is that the PS approach considers specific schedule and the specific production/consumption pattern, while in the case of our ISPS the computed buffer sizes are valid for any schedule of actor firings during its periods and for any production/consumption pattern during its firing.

We also measured the execution times of our ISPS approach enhanced with the latency reduction method to find tasks' deadlines and a schedule, that is, tasks' start times, such that the latency constraint is satisfied. In most cases our latency reduction method needed less than a second, and in three cases less than a minute, to find tasks' deadlines and a schedule which meets the latency constraint.

Table 3.6: *Performance of the ISPS approach under different latency constraints.*

Benchmark	$\mathcal{L}_{\text{constraint}} = \mathcal{L}^{\text{SPS}}$					$\mathcal{L}_{\text{constraint}} = \mathcal{L}^{\text{PS}}$			
	$\frac{\mathcal{R}_{\text{out}}^{\text{ISPS}}}{\mathcal{R}_{\text{out}}^{\text{SPS}}}$	$\frac{\mathcal{L}^{\text{ISPS}}}{\mathcal{L}^{\text{SPS}}}$	$m_{\text{PAR}}^{\text{ISPS}}$	$m_{\text{PAR}}^{\text{SPS}}$	$\frac{\text{M}^{\text{ISPS}}}{\text{M}^{\text{SPS}}}$	$\frac{\mathcal{R}_{\text{out}}^{\text{ISPS}}}{\mathcal{R}_{\text{out}}^{\text{PS}}}$	$\frac{\mathcal{L}^{\text{ISPS}}}{\mathcal{L}^{\text{PS}}}$	$m_{\text{PAR}}^{\text{ISPS}}$	$\frac{\text{M}^{\text{ISPS}}}{\text{M}^{\text{PS}}}$
Pacemaker	1.5	0.99	2	2	1.47	1	1	4	2.64
RSD	11.2	0.097	1	1	1.56	1	1	3	1.15
BlackScholes	1.33	1	18	17	5.7	1	1.16	41	6.86
Disp. map	1.03	0.95	2	2	1	1	1	5	1.33
Pdetect	1.0002	0.9	13	13	1.09	1	–	54	–
CELP	1.5	0.99	6	6	1.68	1	1.1	9	1.6
CD2DAT	1	1	2	2	4.75	1	3.35	6	8.8
MP3 Playback	1	1	4	4	1.13	1	1.1	4	1.26
JPEG2000	3.93	0.3	1	1	1.21	1	–	230	–

3.7 Discussion

The theoretical analysis presented in Section 3.5 proves that streaming applications, modeled as acyclic CSDF graphs, can be converted to real-time periodic tasks by using our scheduling approach which converts each actor in a CSDF graph, by considering different WCET value for each actor phase, to a set of strictly periodic tasks. As a result, a variety of hard real-time scheduling algorithms can be applied to temporally schedule the graph on a platform with calculated number of processors with a certain guaranteed throughput and latency. Additionally, the latency reduction method presented in Section 3.5 can be used to reduce the graph latency when the converted tasks are scheduled as real-time periodic tasks. The experiments on a set of real-life applications

showed that our ISPS approach gives tighter guarantee on the throughput and better processor utilization with acceptable increase in terms of communication memory requirements when compared with the SPS hard real-time scheduling approach. By applying our proposed latency reduction method, the ISPS delivers shorter graph latency while providing better throughput and processor utilization than the SPS approach. When compared with the PS approach, our proposed approach gives the same throughput with increased communication memory but takes much shorter time for deriving the schedule and for calculating the minimum number of processors and the size of communication buffers. Finally, our approach gives throughput that is equal or very close to the absolute maximum throughput achieved by the self-timed scheduling (STS) of actor firings but requires much shorter time to derive the schedule.