On the Hard-Real-Time Scheduling of Embedded Streaming Applications

Bamakhrama, M.A.M.; Stefanov, T.P.

Des Autom Embed Syst (2013) 17:221–249
DOI 10.1007/s10617-012-9086-x

On the hard-real-time scheduling of embedded streaming applications

Mohamed A. Bamakhrama · Todor P. Stefanov

Received: 28 February 2012 / Accepted: 5 June 2012 / Published online: 28 June 2012

Abstract In this paper, we consider the problem of hard-real-time (HRT) multiprocessor scheduling of embedded streaming applications modeled as acyclic dataflow graphs. Most of the hard-real-time scheduling theory for multiprocessor systems assumes independent periodic or sporadic tasks. Such a simple task model is not directly applicable to dataflow graphs, where nodes represent actors (i.e., tasks) and edges represent data-dependencies. The actors in such graphs have data-dependency constraints and do not necessarily conform to the periodic or sporadic task models. In this work, we prove that the actors in acyclic Cyclo-Static Dataflow (CSDF) graphs can be scheduled as periodic tasks. Moreover, we provide a framework for computing the periodic task parameters (i.e., period and start time) of each actor, and handling sporadic input streams. Furthermore, we define formally a class of CSDF graphs called *matched input/output (I/O) rates* graphs which represents more than 80 % of streaming applications. We prove that strictly periodic scheduling is capable of achieving the maximum achievable throughput of an application for matched I/O rates graphs. Therefore, hard-real-time schedulability analysis can be used to determine the minimum number of processors needed to schedule matched I/O rates applications while delivering the maximum achievable throughput. This can be of great use for system designers during the Design Space Exploration (DSE) phase.

Keywords Real-time multiprocessor scheduling · Embedded streaming systems

1 Introduction

The ever-increasing complexity of embedded systems realized as Multi-Processor Systems-on-Chips (MPSoCs) is imposing several challenges on systems designers [18]. Two major challenges in designing streaming software for embedded MPSoCs are: (1) How to express

M.A. Bamakhrama (✉) · T.P. Stefanov
Leiden Institute of Advanced Computer Science, Leiden University, Leiden, The Netherlands
e-mail: mohamed@liacs.nl

T.P. Stefanov
e-mail: stefanov@liacs.nl

parallelism found in applications efficiently?, and (2) How to allocate the processors to provide guaranteed services to multiple running applications, together with the ability to dynamically start/stop applications without affecting other already running applications?

Model-of-Computation (MoC) based design has emerged as a de-facto solution to the first challenge [10]. In MoC-based design, the application can be modeled as a directed graph where nodes represent actors (i.e., tasks) and edges represent communication channels. Different MoCs define different rules and semantics on the computation and communication of the actors. The main benefits of a MoC-based design are the explicit representation of important properties in the application (e.g., parallelism) and the enhanced design-time analyzability of the performance metrics (e.g., throughput). One particular MoC that is popular in the embedded signal processing systems community is the Cyclo-Static Dataflow (CSDF) model [5] which extends the well-known Synchronous Data Flow (SDF) model [15].

Unfortunately, no such de-facto solution exists yet for the second challenge of processor allocation [23]. For a long time, self-timed scheduling was considered the most appropriate policy for streaming applications modeled as dataflow graphs [14, 28]. However, the need to support multiple applications running on a single system without prior knowledge of the properties of the applications (e.g., required throughput, number of tasks, etc.) at system design-time is forcing a shift towards run-time scheduling approaches as explained in [13]. Most of the existing run-time scheduling solutions assume applications modeled as task graphs and provide best-effort or soft-real-time quality-of-service (QoS) [23]. Few run-time scheduling solutions exist which support applications modeled using a MoC and provide hard-real-time QoS [4, 11, 20, 21]. However, these solutions either use simple MoCs such as SDF/PGM graphs or use Time-Division Multiplexing (TDM)/Round-Robin (RR) scheduling. Several algorithms from the hard-real-time multiprocessor scheduling theory [9] can perform *fast* admission and scheduling decisions for incoming applications while providing hard-real-time QoS. Moreover, these algorithms provide *temporal isolation* which is the ability to dynamically start/run/stop applications without affecting other already running applications. However, these algorithms from the hard-real-time multiprocessor scheduling theory received little attention in the embedded MPSoC community. This is mainly due to the fact that these algorithms assume independent periodic or sporadic tasks [9]. Such a simple task model is not directly applicable to modern embedded streaming applications. This is because a modern streaming application is typically modeled as a directed graph where nodes represent actors, and edges represent data-dependencies. The actors in such graphs have *data-dependency constraints* and do not necessarily conform to the periodic or sporadic task models.

Therefore, in this paper we investigate the applicability of the hard-real-time scheduling theory for periodic tasks to streaming applications modeled as acyclic CSDF graphs. In such graphs, the actors are data-dependent. However, we analytically prove that they (i.e., the actors) can be scheduled as periodic tasks. As a result, a variety of hard-real-time scheduling algorithms for periodic tasks can be applied to schedule such applications with a certain guaranteed throughput. By considering acyclic CSDF graphs, our investigation findings and proofs are applicable to most streaming applications since it has been shown recently that around 90 % of streaming applications can be modeled as acyclic SDF graphs [30]. Note that SDF graphs are a subset of the CSDF graphs we consider in this paper.

1.1 Problem statement

Given a streaming application modeled as an acyclic CSDF graph, determine whether it is possible to execute the graph actors as periodic tasks. A periodic task τ_i is defined by a 3-

tuple $\tau_i = (S_i, C_i, T_i)$. The interpretation is as follows: τ_i is invoked at time instants $t = S_i + kT_i$ and it has to execute for C_i time-units before time $t = S_i + (k+1)T_i$ for all $k \in \mathbb{N}_0$, where S_i is the start time of τ_i and T_i is the task period. This scheduling approach is called *Strictly Periodic Scheduling (SPS)* [22] to avoid confusion with the term *periodic scheduling* used in the dataflow scheduling theory to refer to a repetitive finite sequence of actors invocations. The sequence is periodic since it is repeated infinitely with a constant period. However, the individual actors invocations are not guaranteed to be periodic. In the remainder of this paper, periodic scheduling/schedule refers to strictly periodic scheduling/schedule.

1.2 Paper contributions

Given a streaming application modeled as an acyclic CSDF graph, we analytically prove that it is possible to execute the graph actors as periodic tasks. Moreover, we present an analytical framework for computing the periodic task parameters for the actors, that is the period and the start time, together with the minimum buffer sizes of the communication channels such that the actors execute as periodic tasks. The proposed framework is also capable of handling sporadic input streams. Furthermore, we define formally two classes of CSDF graphs: *matched input/output (I/O) rates* graphs and *mis-matched I/O rates* graphs. Matched I/O rates graphs constitute around 80 % of streaming applications [30]. We prove that strictly periodic scheduling is capable of delivering the maximum achievable throughput for matched I/O rates graphs. Applying our approach to matched I/O rates applications enables using a plethora of schedulability tests developed in the real-time scheduling theory [9] to easily determine the minimum number of processors needed to schedule a set of applications using a certain algorithm to provide the maximum achievable throughput. This can be of great use for embedded systems designers during the Design Space Exploration (DSE) phase.

The remainder of this paper is organized as follows: Sect. 2 gives an overview of the related work. Section 3 introduces the CSDF model and the considered system model. Section 4 presents the proposed analytical framework. Section 5 presents the results of empirical evaluation of the framework presented in Sect. 4. Finally, Sect. 6 ends the paper with conclusions.

2 Related work

Parks and Lee [25] studied the applicability of non-preemptive Rate-Monotonic (RM) scheduling to dataflow programs modeled as SDF graphs. The main difference compared to our work is: (1) they considered non-preemptive scheduling. In contrast, we consider only preemptive scheduling. Non-preemptive scheduling is known to be NP-hard in the strong sense even for the uniprocessor case [12], and (2) they considered SDF graphs which are a subset of the more general CSDF graphs.

Goddard [11] studied applying real-time scheduling to dataflow programs modeled using the Processing Graphs Method (PGM). He used a task model called *Rate-Based Execution (RBE)* in which a real-time task τ_i is characterized by a 4-tuple $\tau_i = (x_i, y_i, d_i, c_i)$. The interpretation is as follows: τ_i executes x_i times in time period y_i with a relative deadline d_i per job release and c_i execution time per job release. For a given PGM, he developed an analysis technique to find the RBE task parameters of each actor and buffer size of each channel. Thus, his approach is closely related to ours. However, our approach uses CSDF graphs which are more expressive than PGM graphs in that PGM supports only a *constant*

production/consumption rate on edges (same as SDF), whereas CSDF supports *varying* (but predefined) production/consumption rates. As a result, the analysis technique in [11] is not applicable to CSDF graphs.

Bekooij et al. presented a dataflow analysis for embedded real-time multiprocessor systems [4]. They analyzed the impact of TDM scheduling on applications modeled as SDF graphs. Moreira et al. have investigated real-time scheduling of dataflow programs modeled as SDF graphs in [20–22]. They formulated a resource allocation heuristic [20] and a TDM scheduler combined with static allocation policy [21]. Their TDM scheduler improves the one proposed in [4]. In [22], they proved that it is possible to derive a strictly periodic schedule for the actors of a cyclic SDF graph iff the periods are greater than or equal to the maximum cycle mean of the graph. They formulated the conditions on the start times of the actors in the equivalent Homogeneous SDF (HSDF, [15]) graph in order to enforce a periodic execution of every actor as a Linear Programming (LP) problem.

Our approach differs from [4, 20–22] in: (1) using the periodic task model which allows applying a variety of proven hard-real-time scheduling algorithms for multiprocessors, and (2) using the CSDF model which is more expressive than the SDF model.

3 Background

3.1 Cyclo-static dataflow (CSDF)

In [5], the CSDF model is defined as a directed graph $G = \langle V, E \rangle$, where V is a set of actors and $E \subseteq V \times V$ is a set of communication channels. Actors represent functions that transform incoming data streams into outgoing data streams. The communication channels carry streams of data, and an atomic data object is called a *token*. A channel $e_u \in E$ is a first-in, first-out (FIFO) queue with unbounded capacity, and is defined by a tuple $e_u = (v_i, v_j)$. The tuple means that e_u is directed from v_i (called source) to v_j (called destination). The number of actors in a graph G is denoted by $N = |V|$. An actor receiving an input stream of the application is called *input actor*, and an actor producing an output stream of the application is called *output actor*. A path $w_{a \rightsquigarrow z}$ between actors v_a and v_z is an ordered sequence of channels defined as $w_{a \rightsquigarrow z} = \{(v_a, v_b), (v_b, v_c), \ldots, (v_y, v_z)\}$. A path $w_{i \rightsquigarrow j}$ is called *output path* if v_i is an input actor and v_j is an output actor. $\mathcal{W}$ denotes the set of all output paths in G. In this work, we consider only acyclic CSDF graphs. An acyclic graph G has a number of levels, denoted by $\mathcal{L}$, which is given by Algorithm 1. The level of an actor $v_i \in V$ is denoted by σ_i. Each actor $v_i \in V$ is associated with four sets:

1. The successors set, denoted by $\mathsf{succ}(v_i)$, and given by:

$$\mathsf{succ}(v_i) = \left\{ v_j \in V : \exists e_u = (v_i, v_j) \in E \right\} \tag{1}$$

2. The predecessors set, denoted by $\mathsf{prec}(v_i)$, and given by:

$$\mathsf{prec}(v_i) = \left\{ v_j \in V : \exists e_u = (v_j, v_i) \in E \right\} \tag{2}$$

3. The input channels set, denoted by $\mathsf{inp}(v_i)$, and given by:

$$\mathsf{inp}(v_i) = \begin{cases} \{e_u \in E : e_u = (v_j, v_i)\}, & \text{if } \sigma_i > 1 \\ \text{The set of channels delivering the input streams to } v_i & \text{if } \sigma_i = 1 \end{cases} \tag{3}$$

4. The output channels set, denoted by $\mathsf{out}(v_i)$, and given by:

$$\mathsf{out}(v_i) = \begin{cases} \{e_u \in E : e_u = (v_i, v_j)\}, & \text{if } \sigma_i < \mathcal{L} \\ \text{The set of channels carrying the output streams from } v_i, & \text{if } \sigma_i = \mathcal{L} \end{cases} \quad (4)$$

Algorithm 1 LEVELS(G)

Require: Acyclic CSDF graph $G = \langle V, E \rangle$

1: $i \leftarrow 1$
2: **while** $V \neq \emptyset$ **do**
3: $A_i \leftarrow \{v_j \in V : \mathsf{prec}(v_j) = \emptyset\}$
4: $Z_i \leftarrow \{e_u \in E : \exists v_k \in A_i$ that is the source of $e_u\}$
5: $V \leftarrow V \setminus A_i$
6: $E \leftarrow E \setminus Z_i$
7: $i \leftarrow i + 1$
8: **end while**
9: $\mathcal{L} \leftarrow i - 1$
10: **return** $\mathcal{L}$ disjoint sets $A_1, A_2, \ldots, A_{\mathcal{L}}$, where $\bigcup_{i=1}^{\mathcal{L}} A_i = V$

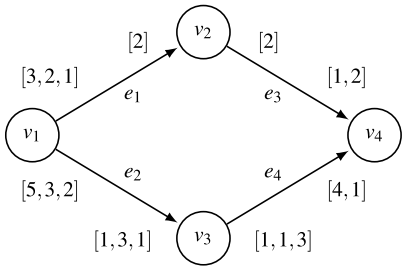
Every actor $v_j \in V$ has an execution sequence $[f_j(1), f_j(2), \ldots, f_j(P_j)]$ of length P_j. The interpretation of this sequence is: The nth time that actor v_j is fired, it executes the code of function $f_j(((n-1) \bmod P_j) + 1)$. Similarly, production and consumption of tokens are also sequences of length P_j in CSDF. The token production of actor v_j on channel e_u is represented as a sequence of constant integers $[x_j^u(1), x_j^u(2), \ldots, x_j^u(P_j)]$. The nth time that actor v_j is fired, it produces $x_j^u(((n-1) \bmod P_j) + 1)$ tokens on channel e_u. The consumption of actor v_k is completely analogous; the token consumption of actor v_k from a channel e_u is represented as a sequence $[y_k^u(1), y_k^u(2), \ldots, y_k^u(P_j)]$. The firing rule of a CSDF actor v_k is evaluated as "true" for its nth firing iff all its input channels contain at least $y_k^u(((n-1) \bmod P_j) + 1)$ tokens. The total number of tokens produced by actor v_j on channel e_u during the first n invocations, denoted by $X_j^u(n)$, is given by $X_j^u(n) = \sum_{l=1}^{n} x_j^u(l)$. Similarly, the total number of tokens consumed by actor v_k from channel e_u during the first n invocations, denoted by $Y_k^u(n)$, is given by $Y_k^u(n) = \sum_{l=1}^{n} y_k^u(l)$.

Example 1 Figure 1 shows a CSDF graph consisting of four actors and four communication channels. Actor v_1 is the input actor with a successors set $\mathsf{succ}(v_1) = \{v_2, v_3\}$, and v_4 is the output actor with a predecessors set $\mathsf{prec}(v_4) = \{v_2, v_3\}$. There are two output paths in the graph: $w_1 = \{(v_1, v_2), (v_2, v_4)\}$ and $w_2 = \{(v_1, v_3), (v_3, v_4)\}$. The production sequences are shown between square brackets at the start of edges (e.g., $[5, 3, 2]$ for actor v_1 on edge e_2), while the consumption sequences are shown between square brackets at the end of the edges (e.g., $[1, 3, 1]$ for v_3 on e_2).

An important property of the CSDF model is its decidability, which is the ability to derive at compile-time a schedule for the actors. This is formulated in the following definitions and results from [5].

Definition 1 (Valid static schedule [5]) Given a connected CSDF graph G, a *valid static schedule* for G is a *finite sequence of actors invocations* that can be repeated infinitely on the incoming sample stream while the amount of data in the buffers remains bounded. A vector

Fig. 1 Example CSDF graph



$\mathbf{q} = [q_1, q_2, \ldots, q_N]^T$, where $q_j > 0$, is a *repetition vector* of G if each q_j represents the number of invocations of an actor v_j in a valid static schedule for G. The repetition vector of G in which all the elements are relatively prime[1] is called the *basic repetition vector* of G, denoted by $\dot{\mathbf{q}}$. G is *consistent* if there exists a repetition vector. If a deadlock-free schedule can be found, G is said to be *live*. Both consistency and liveness are required for the existence of a valid static schedule.

Theorem 1 ([5]) *In a CSDF graph G, a repetition vector $\mathbf{q} = [q_1, q_2, \ldots, q_N]^T$ is given by*

$$\mathbf{q} = \mathbf{P} \cdot \mathbf{r}, \quad \textit{with } P_{jk} = \begin{cases} P_j, & \textit{if } j = k \\ 0, & \textit{otherwise} \end{cases} \tag{5}$$

where $\mathbf{r} = [r_1, r_2, \ldots, r_N]^T$ is a positive integer solution of the balance equation

$$\Gamma \cdot \mathbf{r} = \mathbf{0} \tag{6}$$

and where the topology matrix *$\Gamma \in \mathbb{Z}^{|E| \times |V|}$ is defined by*

$$\Gamma_{uj} = \begin{cases} X_j^u(P_j), & \textit{if actor } v_j \textit{ produces on channel } e_u \\ -Y_j^u(P_j), & \textit{if actor } v_j \textit{ consumes from channel } e_u \\ 0, & \textit{Otherwise} \end{cases} \tag{7}$$

Definition 2 For a consistent and live CSDF graph G, an *actor iteration* is the invocation of an actor $v_i \in V$ for q_i times, and a *graph iteration* is the invocation of *every* actor $v_i \in V$ for q_i times, where $q_i \in \mathbf{q}$.

Corollary 1 (From [5]) *If a consistent and live CSDF graph G completes n iterations, where $n \in \mathbb{N}$, then the net change to the number of tokens in the buffers of G is zero.*

Lemma 1 *Any acyclic consistent CSDF graph is live.*

Proof Bilsen et al. proved in [5] that a CSDF graph is live iff every cycle in the graph is live. Equivalently, a CSDF graph deadlocks only if it contains at least one cycle. Thus, absence of cycles in a CSDF graph implies its liveness. □

[1] I.e., $\gcd\{q_1, q_2, \ldots, q_N\} = 1$.

Example 2 For the CSDF graph shown in Fig. 1

$$\Gamma = \begin{bmatrix} 6 & -2 & 0 & 0 \\ 10 & 0 & -5 & 0 \\ 0 & 2 & 0 & -3 \\ 0 & 0 & 5 & -5 \end{bmatrix}, \qquad \mathbf{r} = \begin{bmatrix} 1 \\ 3 \\ 2 \\ 2 \end{bmatrix},$$

$$\mathbf{P} = \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}, \qquad \text{and} \qquad \dot{\mathbf{q}} = \begin{bmatrix} 3 \\ 3 \\ 6 \\ 4 \end{bmatrix}$$

3.2 System model and scheduling algorithms

In this section, we introduce the system model and the related schedulability results.

3.2.1 System model

A system Ω consists of a set $\pi = \{\pi_1, \pi_2, \ldots, \pi_m\}$ of m homogeneous processors. The processors execute a task set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n periodic tasks, and a task may be preempted at any time. A periodic task $\tau_i \in \tau$ is defined by a 4-tuple $\tau_i = (S_i, C_i, T_i, D_i)$, where $S_i \geq 0$ is the start time of τ_i, $C_i > 0$ is the worst-case execution time of τ_i, $T_i \geq C_i$ is the task period, and D_i, where $C_i \leq D_i \leq T_i$, is the relative deadline of τ_i. A periodic task τ_i is invoked (i.e., *releases a job*) at time instants $t = S_i + kT_i$ for all $k \in \mathbb{N}_0$. Upon invocation, τ_i executes for C_i time-units. The relative deadline D_i is interpreted as follows: τ_i has to finish executing its kth invocation before time $t = S_i + kT_i + D_i$ for all $k \in \mathbb{N}_0$. If $D_i = T_i$, then τ_i is said to have *implicit-deadline*. If $D_i < T_i$, then τ_i is said to have *constrained-deadline*. If all the tasks in a task-set τ have the same start time, then τ is said to be *synchronous*. Otherwise, τ is said to be *asynchronous*.

The utilization of a task τ_i is $U_i = C_i/T_i$. For a task set τ, the total utilization of τ is $U_{\text{sum}} = \sum_{\tau_i \in \tau} U_i$ and the maximum utilization factor of τ is $U_{\max} = \max_{\tau_i \in \tau} U_i$.

In the remainder of this paper, a task set τ refers to an asynchronous set of implicit-deadline periodic tasks. As a result, we refer to a task τ_i with a 3-tuple $\tau_i = (S_i, C_i, T_i)$ by omitting the implicit deadline D_i which is equal to T_i.

3.2.2 Scheduling asynchronous set of implicit deadline periodic tasks

Given a system Ω and a task set τ, a *valid schedule* is one that allocates a processor to a task $\tau_i \in \tau$ for exactly C_i time-units in the interval $[S_i + kT_i, S_i + (k+1)T_i)$ for all $k \in \mathbb{N}_0$ with the restriction that a task may not execute on more than one processor at the same time. A necessary and sufficient condition for τ to be scheduled on Ω to meet all the deadlines (i.e., τ is *feasible*) is:

$$U_{\text{sum}} \leq m \tag{8}$$

The problem of constructing a periodic schedule for τ can be solved using several algorithms [9]. These algorithms differ in the following aspects: (1) *Priority Assignment*: A task can have *fixed* priority, *job-fixed* priority, or *dynamic* priority, and (2) *Allocation*: Based on whether a task can migrate between processors upon preemption, algorithms are classified into:

– *Partitioned*: Each task is allocated to a processor and no migration is permitted
– *Global*: Migration is permitted for all tasks
– *Hybrid*: Hybrid algorithms mix partitioned and global approaches and they can be further classified to:

  1. *Semi-partitioned*: Most tasks are allocated to processors and few tasks are allowed to migrate
  2. *Clustered*: Processors are grouped into clusters and the tasks that are allocated to one cluster are scheduled by a global scheduler

An important property of scheduling algorithms is *optimality*. A scheduling algorithm $\mathcal{A}$ is said to be optimal iff it can schedule any feasible task set τ on Ω. Several global and hybrid algorithms were proven optimal for scheduling asynchronous sets of implicit-deadline periodic tasks (e.g., [2, 3, 8, 16]). The minimum number of processors needed to schedule τ using an optimal scheduling algorithm, denoted by M_{OPT}, is given by:

$$M_{\mathsf{OPT}} = \lceil U_{\text{sum}} \rceil \tag{9}$$

Partitioned algorithms are known to be non-optimal for scheduling implicit-deadline periodic tasks [7]. However, they have the advantage of not requiring task migration. One prominent example of partitioned scheduling algorithms is the Partitioned Earliest Deadline First (P-EDF) algorithm. EDF is known to be optimal for scheduling arbitrary task sets on a uniprocessor system [6]. In a multiprocessor system, EDF can be combined with different processor allocation algorithms (e.g., Bin-packing heuristics such as First-Fit (FF) and Worst-Fit (WF)). López et al. derived in [17] the worst-case utilization bounds for a task set τ to be schedulable using P-EDF. These bounds serve as a simple sufficient schedulability test. Based on these bounds, they derived the minimum number of processors needed to schedule a task set τ under P-EDF, denoted by $M_{\text{P-EDF}}$:

$$M_{\text{P-EDF}} \geq \begin{cases} 1, & \text{if } U_{\text{sum}} \leq 1 \\ \min(\lceil \frac{n}{\beta} \rceil, \lceil \frac{(\beta+1)U_{\text{sum}}-1}{\beta} \rceil), & \text{if } U_{\text{sum}} > 1, \end{cases} \tag{10}$$

where $\beta = \lfloor 1/U_{\max} \rfloor$. A task set τ with total utilization U_{sum} and maximum utilization factor $U_{\max}$ is always guaranteed to be schedulable on $M_{\text{P-EDF}}$ processors. Since $M_{\text{P-EDF}}$ is derived based on a sufficient test, it is important to note that τ may be schedulable on less number of processors. We define M_{PAR} as the minimum number of processors on which τ can be partitioned assuming bin packing allocation (e.g., First-Fit (FF)) with each set in the partition having a total utilization of at most 1. M_{PAR} can be expressed as:

$$M_{\mathsf{PAR}} = \min\{x \in \mathbb{N} : B \text{ is } x\text{-partition of } \tau \text{ and } U_{\text{sum}} \leq 1 \text{ for all } y \in B\} \tag{11}$$

M_{PAR} is specific to the task set τ for which it is computed. Another task set $\hat{\tau}$ with the same total utilization and maximum utilization factor as τ might not be schedulable on M_{PAR} processors due to partitioning issues.

4 Strictly periodic scheduling of acyclic CSDF graphs

This section presents our analytical framework for scheduling the actors in acyclic CSDF graphs as periodic tasks. The construction it uses arranges the actors forming the CSDF graph into a set of levels as shown in Sect. 3. All actors belonging to a certain level depend

directly only on the actors in the previous levels. Then, we derive, for each actor, a period and start time, and for each channel, a buffer size. These derived parameters ensure that a strictly periodic schedule can be achieved in the form of a pipelined sequence of invocations of all the actors in each level.

4.1 Definitions and assumptions

In the remainder of this paper, a graph G refers to an acyclic consistent CSDF graph. We base our analysis on the following assumptions:

Assumption 1 A graph G has a set $I = \{I_1, I_2, \ldots, I_{\mathcal{K}}\}$ of $\mathcal{K}$ sporadic input streams connected to the input actors of G. The set of input streams to an actor v_i is denoted by Z_i. We make the following assumptions about the input streams:

1. $Z_i \cap Z_j = \emptyset \; \forall v_i, v_j \in V$.
2. The first samples of all the streams arrive prior to or at the same time when the actors of G start executing
3. Each input stream I_j is characterized by a minimum inter-arrival time (also called period) of the samples, denoted by γ_j. This minimum inter-arrival time is assumed to be equal to the period of the input actor which receives I_j. This assumption indicates that the inter-arrival time for input streams can be controlled by the designer to match the periods of the actors.

Assumption 2 An actor v_i consumes its input data immediately when it starts its firing and produces its output data just before it finishes its firing.

We start with the following definition:

Definition 3 (Execution time vector) For a graph G, an *execution time vector* $\boldsymbol{\mu}$, where $\boldsymbol{\mu} \in \mathbb{N}^N$, represents the worst-case execution times, measured in time-units, of the actors in G. The worst-case execution time of an actor $v_j \in V$ is given by

$$\mu_j = \max_{k=1}^{P_j} \left(T^R \cdot \sum_{e_l \in \mathsf{inp}(v_j)} y_j^l(k) + T^W \cdot \sum_{e_r \in \mathsf{out}(v_j)} x_j^r(k) + T_j^C(k) \right) \tag{12}$$

where P_j is the length of CSDF firing/production/consumption sequences of actor v_j, T^R is the worst-case time needed to read a single token from an input channel, y_j^l is the consumption sequence of v_j from channel e_l, T^W is the worst-case time needed to write a single token to an output channel, x_j^r is the production sequence of v_j into channel e_r, and $T_j^C(k)$ is the worst-case computation time of v_j in firing k.

Let $\eta = \max_{v_i \in V}(\mu_i q_i)$ and $Q = \mathsf{lcm}\{q_1, q_2, \ldots, q_N\}$ (lcm denotes the least-common-multiple operator). Now, we give the following definition.

Definition 4 (Matched input/output rates graph) A graph G is said to be *matched input/output (I/O) rates graph* if and only if

$$\eta \bmod Q = 0 \tag{13}$$

If (13) does not hold, then G is said to be *mis-matched I/O rates graph*.

The concept of matched I/O rates applications was first introduced in [30] as the applications with *low value of* Q. However, the authors did not establish exact test for determining whether an application is matched I/O rates or not. The test in (13) is a novel contribution of this paper. If $\eta \bmod Q = 0$, then there exists at least a single actor in the graph which is fully utilizing the processor on which it runs. This, as shown later in Sect. 4.3.3, allows the graph to achieve optimal throughput. On the other hand, if $\eta \bmod Q \neq 0$, then there exist idle durations in the period of each actor which results in sub-optimal throughput. This is illustrated later in Example 3 which shows the strictly periodic schedule of a mis-matched I/O rates application.

Definition 5 (Output path latency) Let $w_{a \rightsquigarrow z} = \{(v_a, v_b), \ldots, (v_y, v_z)\}$ be an output path in a graph G. The *latency* of $w_{a \rightsquigarrow z}$ under periodic input streams, denoted by $L(w_{a \rightsquigarrow z})$, is the elapsed time between the start of the first firing of v_a which produces data to (v_a, v_b) and the finish of the first firing of v_z which consumes data from (v_y, v_z).

Consequently, we define the maximum latency of G as follows:

Definition 6 (Graph maximum latency) For a graph G, the *maximum latency* of G under periodic input streams, denoted by $L(G)$, is given by:

$$L(G) = \max_{w_{i \rightsquigarrow j} \in \mathcal{W}} L(w_{i \rightsquigarrow j}) \tag{14}$$

Definition 7 (Self-timed schedule) A self-timed schedule (STS) is one where all the actors are fired as soon as their input data are available.

Self-timed scheduling has been shown in [28] to achieve the maximum achievable throughput and minimum achievable latency of a Homogeneous SDF (HSDF, [15]) graph. This results extends to CSDF graphs since any CSDF graph can be converted to an equivalent HSDF graph. For acyclic graphs, the STS throughput of an actor v_i, denoted by $R_{\mathsf{STS}}(v_i)$, is given by:

$$R_{\mathsf{STS}}(v_i) = q_i / \eta \tag{15}$$

Definition 8 (Strictly periodic actor) An actor $v_i \in V$ is *strictly periodic* iff the time period between any two consecutive firings is constant.

Definition 9 (Period vector) For a graph G, a *period vector* $\boldsymbol{\lambda}$, where $\boldsymbol{\lambda} \in \mathbb{N}^N$, represents the periods, measured in time-units, of the actors in G. $\lambda_j \in \boldsymbol{\lambda}$ is the period of actor $v_j \in V$. $\boldsymbol{\lambda}$ is given by the solution to both

$$q_1 \lambda_1 = q_2 \lambda_2 = \cdots = q_{N-1} \lambda_{N-1} = q_N \lambda_N \tag{16}$$

and

$$\boldsymbol{\lambda} - \boldsymbol{\mu} \geq \mathbf{0}, \tag{17}$$

where $q_j \in \dot{\mathbf{q}}$ (the basic repetition vector of G according to Definition 1).

Definition 9 implies that all the actors have the same iteration period. This is captured in the following definition:

Definition 10 (Iteration period) For a graph G, the iteration period under strictly periodic scheduling, denoted by α, is given by

$$\alpha = q_i \lambda_i \quad \text{for any } v_i \in V \tag{18}$$

Now, we prove the existence of a strictly periodic schedule when the input streams are strictly periodic. An input stream I_j connected to input actor v_i is strictly periodic iff the inter-arrival time between any two consecutive samples is constant. Based on Assumption 1-3, it follows that $\gamma_j = \lambda_i$. Later on, we extend the results to handle periodic with jitter and sporadic input streams.

4.2 Existence of a strictly periodic schedule

Lemma 2 *For a graph G, the minimum period vector of G, denoted by $\boldsymbol{\lambda}^{\min}$, is given by*

$$\lambda_i^{\min} = \frac{Q}{q_i}\left\lceil \frac{\eta}{Q} \right\rceil \quad \textit{for } v_i \in V \tag{19}$$

.

Proof Equation (16) can be re-written as:

$$\Delta \cdot \boldsymbol{\lambda} = \mathbf{0}, \tag{20}$$

where $\Delta \in \mathbb{Z}^{(N-1)\times N}$ is given by

$$\Delta_{ij} = \begin{cases} q_1, & \text{if } j = 1 \\ -q_j, & \text{if } j = i+1 \\ 0, & \text{otherwise} \end{cases} \tag{21}$$

Observe that nullity$(\Delta) = 1$. Thus, there exists a single vector which forms the basis of the null-space of Δ. This vector can be represented by taking any unknown λ_k as the free-unknown and expressing the other unknowns in terms of it which results in:

$$\boldsymbol{\lambda} = \lambda_k [q_k/q_1, q_k/q_2, \ldots, q_k/q_N]^T$$

The minimum $\lambda_k \in \mathbb{N}$ is

$$\lambda_k = \mathsf{lcm}\{q_1, q_2, \ldots, q_N\}/q_k$$

Thus, the minimum $\boldsymbol{\lambda} \in \mathbb{N}$ that solves (16) is given by

$$\lambda_i = Q/q_i \quad \text{for } v_i \in V \tag{22}$$

Let $\hat{\boldsymbol{\lambda}}$ be the solution given by (22). Equations (16) and (17) can be re-written as:

$$\Delta(c\hat{\boldsymbol{\lambda}}) = \mathbf{0} \tag{23}$$

$$c\hat{\lambda}_1 \geq \mu_1, \qquad c\hat{\lambda}_2 \geq \mu_2, \ldots, c\hat{\lambda}_N \geq \mu_N \tag{24}$$

where $c \in \mathbb{N}$. Equation (24) can be re-written as:

$$c \geq \mu_1 q_1/Q, \qquad c \geq \mu_2 q_2/Q, \ldots, c \geq \mu_N q_N/Q \tag{25}$$

Fig. 2 Schedule Π_1

time	$[0,\alpha)$	$[\alpha,2\alpha)$	$[2\alpha,3\alpha)$	$\cdots$	$[(\mathcal{L}-1)\alpha,\mathcal{L}\alpha)$
level	$A_1(1)$	$A_2(1)$	$A_3(1)$	$\cdots$	$A_{\mathcal{L}}(1)$

Fig. 3 Schedule Π_2

time	$[0,\alpha)$	$[\alpha,2\alpha)$	$[2\alpha,3\alpha)$	$\cdots$	$[(\mathcal{L}-1)\alpha,\mathcal{L}\alpha)$
level	$A_1(1)$	$A_2(1)$	$A_3(1)$	$\cdots$	$A_{\mathcal{L}}(1)$
	$A_1(2)$	$A_2(2)$	$\cdots$		$A_{\mathcal{L}-1}(2)$

It follows that c must be greater than or equal to $\max_{v_i \in V}(\mu_i q_i)/Q = \eta/Q$. However, η/Q is not always guaranteed to be an integer. As a result, the value is rounded by taking the ceiling. It follows that the minimum $\boldsymbol{\lambda}$ which satisfies both of (16) and (17) is given by

$$\lambda_i = Q/q_i \lceil \eta/Q \rceil \quad \text{for } v_i \in V$$

□

Theorem 2 *For any graph G, a periodic schedule Π exists such that every actor $v_i \in V$ is strictly periodic with a constant period $\lambda_i \in \boldsymbol{\lambda}^{\min}$ and every communication channel $e_u \in E$ has a bounded buffer capacity.*

Proof Recall that in this proof we assume that the input streams to level-1 actors are strictly periodic with periods equal to the input actors periods. Therefore, it follows that level-1 actors can execute periodically since their input streams are always available when they fire. By Definition 2, level-1 actors will complete one iteration when they fire q_i times, where q_i is the repetition of $v_i \in A_1$. Assume that level-1 actors start executing at time $t = 0$. Then, by time $t = \alpha$, level-1 actors are guaranteed to finish one iteration. According to Theorem 1, level-1 actors will also generate enough data such that every actor $v_k \in A_2$ can execute q_k times (i.e., one iteration) with a period λ_k. According to (16), firing v_k for q_k times with a period λ_k takes α time-units. Thus, starting level-2 actors at time $t = \alpha$ guarantees that they can execute periodically with their periods given by Definition 9 for α time-units. Similarly, by time $t = 2\alpha$, level-3 actors will have enough data to execute for one iteration. Thus, starting level-3 actors at time $t = 2\alpha$ guarantees that they can execute periodically for α time-units. By repeating this over all the $\mathcal{L}$ levels, a schedule Π_1 (shown in Fig. 2) is constructed in which all the actors that belong to A_i are started at *start time* ϕ_i given by

$$\phi_i = (i-1)\alpha \tag{26}$$

$A_j(k)$ denotes level-j actors executing their kth iteration. For example, $A_2(1)$ denotes level-2 actors executing their first iteration. At time $t = \mathcal{L}\alpha$, G completes one iteration. It is trivial to observe from Π_1 that as soon as level-1 actors finish one iteration, they can immediately start executing the next iteration since their input streams arrive periodically. If level-1 actors start their second iteration at time $t = \alpha$, their execution will overlap with the execution of the level-2 actors. By doing so, level-2 actors can start immediately their second iteration after finishing their first iteration since they will have all the needed data to execute one iteration periodically at time $t = 2\alpha$. This overlapping can be applied to all the levels to yield the schedule Π_2 shown in Fig. 3.

Now, the overlapping can be applied $\mathcal{L}$ times on schedule Π_1 to yield a schedule $\Pi_{\mathcal{L}}$ as shown in Fig. 4.

Starting from time $t = \mathcal{L}\alpha$, a schedule Π_∞ can be constructed as shown in Fig. 5.

Fig. 4 Schedule $\Pi_{\mathcal{L}}$

time	$[0,\alpha)$	$[\alpha,2\alpha)$	$[2\alpha,3\alpha)\cdots$	$[(\mathcal{L}-1)\alpha,\mathcal{L}\alpha)$
level	$A_1(1)$	$A_2(1)$	$A_3(1)\ \cdots$	$A_{\mathcal{L}}(1)$
		$A_1(2)$	$A_2(2)\ \cdots$	$A_{\mathcal{L}-1}(2)$
			$A_1(3)\ \cdots$	$A_{\mathcal{L}-2}(3)$
			$\cdots$	$A_{\mathcal{L}-3}(4)$
			$\cdots$	$\cdots$
				$A_1(\mathcal{L})$

Fig. 5 Schedule Π_{∞}

time	$[0,\alpha)$	$[\alpha,2\alpha)$	$[2\alpha,3\alpha)\cdots$	$[(\mathcal{L}-1)\alpha,\mathcal{L}\alpha)$	$[\mathcal{L}\alpha,(\mathcal{L}+1)\alpha)\cdots$	
level	$A_1(1)$	$A_2(1)$	$A_3(1)\ \cdots$	$A_{\mathcal{L}}(1)$	$A_{\mathcal{L}}(2)$	$\cdots$
		$A_1(2)$	$A_2(2)\ \cdots$	$A_{\mathcal{L}-1}(2)$	$A_{\mathcal{L}-1}(3)$	$\cdots$
			$A_1(3)\ \cdots$	$A_{\mathcal{L}-2}(3)$	$A_{\mathcal{L}-2}(4)$	$\cdots$
			$\cdots$	$A_{\mathcal{L}-3}(4)$	$A_{\mathcal{L}-3}(5)$	$\cdots$
			$\cdots$	$\cdots$	$\cdots$	$\cdots$
				$A_1(\mathcal{L})$	$A_1(\mathcal{L}+1)$	$\cdots$

In schedule Π_{∞}, every actor v_i is fired every λ_i time-unit once it starts. The start time defined in (26) guarantees that actors in a given level will start only when they have enough data to execute one iteration in a periodic way. The overlapping guarantees that once the actors have started, they will always find enough data for executing the next iteration since their predecessors have already executed one additional iteration. Thus, schedule Π_{∞} shows the existence of a periodic schedule of G where every actor $v_j \in V$ is strictly periodic with a period equal to λ_j.

The next step is to prove that Π_{∞} executes with bounded memory buffers. In Π_{∞}, the largest delay in consuming the tokens occurs for a channel $e_u \in E$ connecting a level-1 actor and a level-$\mathcal{L}$ actor. This is illustrated in Fig. 5 by observing that the data produced by iteration-1 of a level-1 source actor will be consumed by iteration-1 of a level-$\mathcal{L}$ destination actor after $(\mathcal{L}-1)\alpha$ time-units. In this case, e_u must be able to store at least $(\mathcal{L}-1)X_1^u(q_1)$ tokens. However, starting from time $t = \mathcal{L}\alpha$, both of the level-1 and level-$\mathcal{L}$ actors execute in parallel. Thus, we increase the buffer size by $X_1^u(q_1)$ tokens to account for the overlapped execution. Hence, the total buffer size of e_u is $\mathcal{L}X_1^u(q_1)$ tokens. Similarly, if a level-2 actor, denoted v_2, is connected directly to a level-$\mathcal{L}$ actor via channel e_v, then e_v must be able to store at least $(\mathcal{L}-1)X_2^v(q_2)$ tokens. By repeating this argument over all the different pairs of levels, it follows that each channel $e_u \in E$, connecting a level-i source actor and a level-j destination actor, where $j \geq i$, will store according to schedule Π_{∞} at most:

$$b_u = (j - i + 1)X_k^u(q_k) \tag{27}$$

tokens, where v_k is the level-i actor, and $q_k \in \dot{\mathbf{q}}$. Thus, an upper bound on the FIFO sizes exists. □

Example 3 We illustrate Theorem 2 by constructing a periodic schedule for the CSDF graph shown in Fig. 1. Assume that the CSDF graph has an execution vector $\boldsymbol{\mu} = [5, 2, 3, 2]^T$. Given $\dot{\mathbf{q}} = [3, 3, 6, 4]^T$ as computed in Example 2, we use (19) to find $\boldsymbol{\lambda}^{\min} = [8, 8, 4, 6]^T$. Figure 6 illustrates the periodic schedule of the actors for the first graph iteration. $\mathcal{L} = 3$ and the levels consist of three sets: $A_1 = \{v_1\}$, $A_2 = \{v_2, v_3\}$, and $A_3 = \{v_4\}$. A_1 actors start at time $t = 0$. Since $\alpha = q_i\lambda_i = 24$ for any v_i in the graph, A_2 actors start at time $t = \alpha = 24$ and A_3 actors start at time $t = 2\alpha = 48$. Every actor v_j in the graph executes for μ_j time-units every λ_j time-units. For example, actor v_2 starts at time $t = 24$ and executes for 2 time-units every 8 time-units.

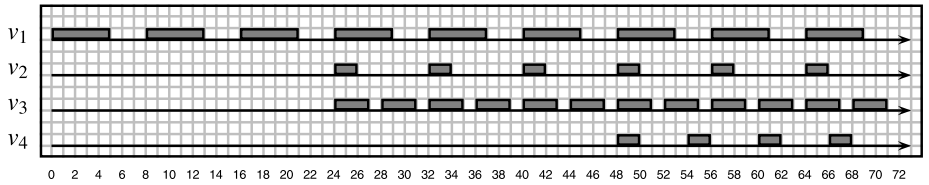


Fig. 6 Strictly periodic schedule for the CSDF graph shown in Fig. 1. The x-axis represents the time axis.

4.3 Earliest start times and minimum buffer sizes

Now, we are interested in finding the earliest start times of the actors, and the minimum buffer sizes of the communication channels that guarantee the existence of a periodic schedule. Minimizing the start times and buffer sizes is crucial since it minimizes the initial response time and the memory requirements of the applications modeled as acyclic CSDF graphs.

4.3.1 Earliest start times

In the proof of Theorem 2, the notion of *start time* was introduced to denote when the actor is started on the system. The start time values used in the proof of the theorem were not the minimum ones. Here, we derive the earliest start times. We start with the following definitions:

Definition 11 (Cumulative production function) The cumulative production function of actor v_i producing into channel e_u during the interval $[t_s, t_e)$, denoted by $\mathsf{prd}_{[t_s,t_e)}(v_i, e_u)$, is the sum of the number of tokens produced by v_i into e_u during the interval $[t_s, t_e)$.

In case of implicit-deadline periodic tasks, $\mathsf{prd}_{[t_s,t_e)}(v_i, e_u)$ is given by:

$$\underset{[t_s,t_e)}{\mathsf{prd}}(v_i, e_u) = \begin{cases} X_i^u(\lfloor \frac{t_e - t_s}{\lambda_i} \rfloor), & \text{if } (t_e - t_s) \geq \lambda_i \\ 0, & \text{if } (t_e - t_s) < \lambda_i \end{cases} \tag{28}$$

Similarly, we define the cumulative consumption function as follows:

Definition 12 (Cumulative consumption function) The cumulative consumption function of actor v_i consuming from channel e_u over the interval $[t_s, t_e]$, denoted by $\mathsf{cns}_{[t_s,t_e]}(v_i, e_u)$, is the sum of the number of tokens consumed by v_i from e_u during the interval $[t_s, t_e]$.

Similar to (28), $\mathsf{cns}_{[t_s,t_e]}(v_i, e_u)$ is given by:

$$\underset{[t_s,t_e]}{\mathsf{cns}}(v_i, e_u) = \begin{cases} 0, & \text{if } t_e < t_s \\ Y_i^u(\lceil \frac{t_e - t_s}{\lambda_i} \rceil + 1), & \text{if } (t_e - t_s) \bmod \lambda_i = 0 \\ Y_i^u(\lceil \frac{t_e - t_s}{\lambda_i} \rceil), & \text{if } (t_e - t_s) \bmod \lambda_i \neq 0 \end{cases} \tag{29}$$

Recall that $\mathsf{prec}(v_i)$ is the predecessors set of actor v_i, Y_i^u is the consumption sequence of an actor v_i from channel e_u, and α is the iteration period. Now, we give the following lemma:

Lemma 3 *For a graph G, the earliest start time of an actor $v_j \in V$, denoted by ϕ_j, under a strictly periodic schedule is given by*

$$\phi_j = \begin{cases} 0, & \text{if } \mathsf{prec}(v_j) = \emptyset \\ \max\limits_{v_i \in \mathsf{prec}(v_j)} (\phi_{i \to j}), & \text{if } \mathsf{prec}(v_j) \neq \emptyset \end{cases} \tag{30}$$

where

$$\phi_{i \to j} = \min_{t \in [0, \phi_i + \alpha]} \left\{ t : \mathop{\mathsf{prd}}_{[\phi_i, \max(\phi_i, t) + k)} (v_i, e_u) \geq \mathop{\mathsf{cns}}_{[t, \max(\phi_i, t) + k]} (v_j, e_u) \ \forall k = 0, 1, \ldots, \alpha \right\} \tag{31}$$

Proof Theorem 2 proved that starting a level-k actor v_j at a start time

$$\phi_j = (k-1)\alpha \tag{32}$$

guarantees strictly periodic execution of the actor v_j. Any start time later than that guarantees also strictly periodic execution since v_j will always find enough data to execute in a strictly periodic way.

Equation (32) can be re-written as:

$$\phi_j = \begin{cases} 0, & \text{if } \mathsf{prec}(v_j) = \emptyset \\ \max\limits_{v_i \in \mathsf{prec}(v_j)} (\phi_i) + \alpha, & \text{if } \mathsf{prec}(v_j) \neq \emptyset \end{cases} \tag{33}$$

The equivalence follows from observing that a level-k actor, where $k > 1$, has a level-$(k-1)$ predecessor. Hence, applying (33) to a level-k actor, where $k > 1$, yields:

$$\phi_j = \max\big((k-2)\alpha, (k-3)\alpha, \ldots, 0\big) + \alpha = (k-1)\alpha$$

Now, we are interested in starting $v_j \in A_k$, where $k > 1$, earlier. That is:

$$\phi_j \leq \max_{v_i \in \mathsf{prec}(v_j)} (\phi_i) + \alpha \tag{34}$$

ϕ_j has also a lower-bound by observing that an actor v_j can not start before the application is started. That is:

$$0 \leq \phi_j \leq \max_{v_i \in \mathsf{prec}(v_j)} (\phi_i) + \alpha \quad \Rightarrow \quad 0 \leq \phi_j \leq \max_{v_i \in \mathsf{prec}(v_j)} (\phi_i + \alpha) \tag{35}$$

If we select ϕ_j such that

$$\phi_j = \max_{v_i \in \mathsf{prec}(v_j)} (\phi_{i \to j}), \phi_{i \to j} = \hat{t}, \quad \hat{t} \in [0, \phi_i + \alpha] \tag{36}$$

then this guarantees that ϕ_j also satisfies (35).

In (36), a valid start time candidate $\phi_{i \to j}$ must satisfy extra conditions to guarantee that the number of produced tokens on edge $e_u = (v_i, v_j)$ at any time instant $t \geq \hat{t}$ is greater than or equal to the number of consumed tokens at the same instant. To satisfy these extra conditions, we consider the following two possible cases:

Case I: $\hat{t} \geq \phi_i$. This case is illustrated in Fig. 7. In this case, a valid start time candidate $\hat{t}$ must satisfy:

$$\mathop{\mathsf{prd}}_{[\phi_i, \hat{t} + k)} (v_i, e_u) \geq \mathop{\mathsf{cns}}_{[\hat{t}, \hat{t} + k]} (v_j, e_u) \quad \forall k = 0, 1, \ldots, \alpha \tag{37}$$

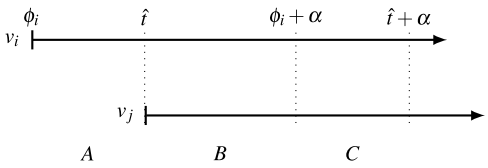


Fig. 7 Timeline of v_i and v_j when $\hat{t} \geq \phi_i$

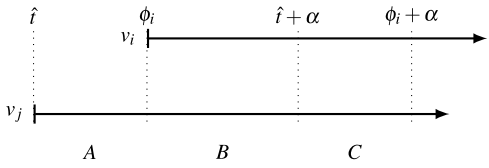


Fig. 8 Timeline of v_i and v_j when $\hat{t} < \phi_i$

Satisfying (37) guarantees that v_j can fire at times $t = \hat{t}, \hat{t} + \lambda_j, \ldots, \hat{t} + \alpha$. Thus, a valid value of $\hat{t}$ guarantees that once v_j is started, it always finds enough data to fire for one iteration. As a result, v_j executes in a strictly periodic way.

Case II: $\hat{t} < \phi_i$. This case is illustrated in Fig. 8. A valid start time candidate $\hat{t}$ must satisfy:

$$\underset{[\phi_i, \phi_i + k)}{\mathsf{prd}}(v_i, e_u) \geq \underset{[\hat{t}, \phi_i + k]}{\mathsf{cns}}(v_j, e_u) \quad \forall k = 0, 1, \ldots, \alpha \tag{38}$$

This case occurs when v_j consumes zeros tokens during the interval $[\hat{t}, \phi_i]$. This is a valid behavior since the consumption rates sequence can contain zero elements. Since $\hat{t} < \phi_i$, it is sufficient to check the cumulative production and consumption over the interval $[\phi_i, \phi_i + \alpha]$ since by time $t = \phi_i + \alpha$ both v_i and v_j are guaranteed to have finished one iteration. Thus, $\hat{t}$ also guarantees that once v_j is started, it always finds enough data to fire. Hence, v_j executes in a strictly periodic way.

Now, we can merge (37) and (38) which results in:

$$\underset{[\phi_i, \max(\phi_i, \hat{t}) + k)}{\mathsf{prd}}(v_i, e_u) \geq \underset{[\hat{t}, \max(\phi_i, \hat{t}) + k]}{\mathsf{cns}}(v_j, e_u) \quad \forall k = 0, 1, \ldots, \alpha \tag{39}$$

Any value of $\hat{t}$ which satisfies (39) is a valid start time value that guarantees strictly periodic execution of v_j. Since there might be multiple values of $\hat{t}$ that satisfy (39), we take the minimum value because it is the earliest start time that guarantees strictly periodic execution of v_j. □

4.3.2 Minimum buffer sizes

Lemma 4 *For a graph G, the minimum bounded buffer size b_u of a communication channel $e_u \in E$ connecting a source actor v_i with start time ϕ_i, and a destination actor v_j with start time ϕ_j, where $v_i, v_j \in V$, under a strictly periodic schedule is given by*

$$b_u = \begin{cases} \max\limits_{k \in [0,1,\ldots,\alpha]} (\mathsf{prd}_{[\phi_i, \phi_j + k)}(v_i, e_u) - \mathsf{cns}_{[\phi_j, \phi_j + k)}(v_j, e_u)), & \text{if } \phi_i \leq \phi_j \\ \max\limits_{k \in [0,1,\ldots,\alpha]} (\mathsf{prd}_{[\phi_i, \phi_i + k)}(v_i, e_u) - \mathsf{cns}_{[\phi_j, \phi_i + k)}(v_j, e_u)), & \text{if } \phi_i > \phi_j \end{cases} \tag{40}$$

Proof Equation (40) tracks the maximum cumulative number of unconsumed tokens in e_u during one iteration for v_i and v_j. There are two cases:

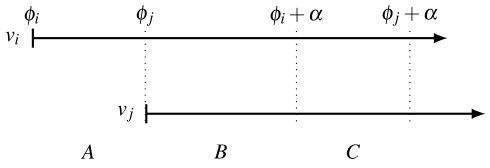


Fig. 9 Execution time-lines of v_i and v_j when $\phi_i \leq \phi_j$

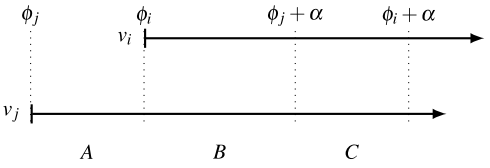


Fig. 10 Execution time-lines of v_i and v_j when $\phi_i > \phi_j$

Case I: $\phi_i \leq \phi_j$. In this case, (40) tracks the maximum cumulative number of unconsumed tokens in e_u during the time interval $[\phi_i, \phi_j + \alpha)$. Figure 9 illustrates the execution time-lines of v_i and v_j when $\phi_i \leq \phi_j$. In interval A, v_i is actively producing tokens while v_j has not yet started executing. As a result, it is necessary to buffer all the tokens produced in this interval in order to prevent v_i from blocking on writing. Thus, b_u must be greater than or equal to $\mathsf{prd}_{[\phi_i, \phi_j)}(v_i, e_u)$. Starting from time $t = \phi_j$, both of v_i and v_j are executing in parallel (i.e., overlapped execution). In the proof of Theorem 2, an additional $X_i^u(q_i)$ tokens were added to the buffer size of e_u to account for the overlapped execution. However, this value is a "worst-case" value. The minimum number of tokens that needs to be buffered is given by the maximum number of unconsumed tokens in e_u at any time over the time interval $[\phi_j, \phi_j + \alpha)$ (i.e., intervals B and C in Fig. 9). Taking the maximum number of unconsumed tokens guarantees that v_i will always have enough space to write to e_u. Thus, b_u is sufficient and minimum for guaranteeing strictly periodic execution of v_i and v_j in the time interval $[\phi_i, \phi_j + \alpha)$. At time $t = \phi_j + \alpha$, both of v_i and v_j have completed one iteration and the number of tokens in e_u is the same as at time $t = \phi_j$ (Follows from Corollary 1). Due to the strict periodicity of v_i and v_j, the pattern shown in Fig. 9 repeats. Thus, b_u is also sufficient and minimum for any $t \geq \phi_j + \alpha$.

Case II: $\phi_i > \phi_j$. Figure 10 illustrates this case. According to Lemma 3, ϕ_j can be smaller than ϕ_i iff v_i consumes zero tokens in interval A. Therefore, the intervals in which there is actually production/consumption of tokens are B and C. During interval B, there is overlapped execution and b_u gives the maximum number of unconsumed tokens in e_u during $[\phi_i, \phi_j + \alpha)$ which guarantees that v_i always have enough space to write to e_u and v_j has enough data to consume from e_u. At time $t = \phi_j + \alpha$, v_j finishes one iteration and interval C starts. During interval C, v_i is producing data to e_u while v_j is consuming zero tokens. Therefore, e_u has to accommodate all the tokens produced during interval C and b_u must be greater than or equal to $\mathsf{prd}_{[\phi_j + \alpha, \phi_i + \alpha]}(v_i, e_u)$. As in Case I, b_u is sufficient and minimum for guaranteeing strictly periodic execution of v_i and v_j in the interval $[\phi_j, \phi_i + \alpha]$. At time $t = \phi_i + \alpha$, both of v_i and v_j have completed one iteration and e_u contains a number of tokens equal to the number of tokens at time $t = \phi_i$. Due to the strict periodicity of v_i and v_j, their execution pattern repeats. Thus, b_u is also sufficient and minimum for any $t \geq \phi_i + \alpha$. □

Theorem 3 *For a graph G, let τ_G be a task set such that $\tau_i \in \tau_G$ corresponds to $v_i \in V$. τ_i is given by*:

$$\tau_i = (\phi_i, \mu_i, \lambda_i), \tag{41}$$

where ϕ_i is the earliest start time of v_i given by (30), $\mu_i \in \boldsymbol{\mu}$, and $\lambda_i \in \boldsymbol{\lambda}^{\min}$ is the period given by (19). τ_G is schedulable on M processors using any hard-real-time scheduling algorithm $\mathcal{A}$ for asynchronous sets of implicit-deadline periodic tasks if:

1. *every edge $e_u \in E$ has a capacity of at least b_u tokens, where b_u is given by (40)*
2. *τ_G satisfies the schedulability test of $\mathcal{A}$ on M processors*

Proof Follows from Theorem 2, and Lemmas 3 and 4. □

Example 4 This is an example to illustrate Lemmas 3, 4, and Theorem 3. First, we calculate the earliest start times and the corresponding minimum buffer sizes for the CSDF graph shown in Fig. 1. Applying Lemmas 3 and 4 on the CSDF graph results in:

$$\begin{bmatrix}\phi_1\\ \phi_2\\ \phi_3\\ \phi_4\end{bmatrix} = \begin{bmatrix}0\\ 8\\ 8\\ 20\end{bmatrix} \quad \text{and} \quad \begin{bmatrix}b_1\\ b_2\\ b_3\\ b_4\end{bmatrix} = \begin{bmatrix}3\\ 5\\ 3\\ 5\end{bmatrix},$$

where ϕ_i denotes the earliest start time of actor v_i, and b_j denotes the minimum buffer size of communication channel e_j. Given $\boldsymbol{\mu}$ and $\boldsymbol{\lambda}^{\min}$ computed in Example 3, we construct a task set $\tau_G = \{(0, 5, 8), (8, 2, 8), (8, 3, 4), (20, 2, 6)\}$. We compute the minimum number of required processors to schedule τ_G according to (9), (10), and (11):

$$M_{\mathsf{OPT}} = \lceil 5/8 + 2/8 + 3/4 + 2/6 \rceil = \lceil 47/24 \rceil = 2$$
$$M_{\mathsf{P\text{-}EDF}} = \min\left\{\lceil 4/1 \rceil, \left\lceil (2 \times 47/24 - 1)/1 \right\rceil\right\} = 3$$
$$M_{\mathsf{PAR}} = \min_{x \in \mathbb{N}} \{x : B \text{ is } x\text{-partition of } \tau \text{ and } U_{\text{sum}} \leq 1 \text{ for all } y \in B\} = 3$$

τ_G is schedulable using an optimal scheduling algorithm on 2 processors, and is schedulable using P-EDF on 3 processors.

4.3.3 Throughput and latency analysis

Now, we analyze the throughput of the graph actors under strictly periodic scheduling and compare it with the maximum achievable throughput. We also present a formula to compute the latency for a given CSDF graph under strictly periodic scheduling. We start with the following definitions:

Definition 13 (Actor throughput) For a graph G, the throughput of actor $v_i \in V$ under strictly periodic scheduling, denoted by $R_{\mathsf{SPS}}(v_i)$, is given by

$$R_{\mathsf{SPS}}(v_i) = 1/\lambda_i \tag{42}$$

Definition 14 (Rate-optimal strictly periodic schedule [22]) For a graph G, a strictly periodic schedule that delivers the same throughput as a self-timed schedule for all the actors is called *Rate-Optimal Strictly Periodic Schedule (ROSPS)*.

Now, we provide the following result.

Theorem 4 *For a matched I/O rates graph G, the maximum achievable throughput of the graph actors under strictly periodic scheduling is equal to their maximum throughput under self-timed scheduling.*

Proof The maximum achievable throughput under strictly periodic scheduling is the one obtained when $\lambda_i = \lambda_i^{\min}$. Recall from (19) that

$$\lambda_i^{\min} = \frac{Q}{q_i}\left\lceil \frac{\eta}{Q} \right\rceil \tag{43}$$

Let us re-write η as $\eta = pQ + r$, where $p = \eta \div Q$ ($\div$ is the integer division operator), and $r = \eta \bmod Q$. Now, (43) can be re-written as

$$\lambda_i^{\min} = \begin{cases} \eta/q_i, & \text{if } \eta \bmod Q = 0 \\ (p+1)Q/q_i, & \text{if } \eta \bmod Q \neq 0 \end{cases} \tag{44}$$

Recall from (15) that

$$R_{\mathsf{STS}}(v_i) = q_i/\eta \tag{45}$$

Now, recall from Definition 4 that a matched I/O rates graph satisfies the following condition:

$$\eta \bmod Q = 0 \tag{46}$$

Therefore, the maximum achievable throughput of the actors of a matched I/O rates graph under strictly periodic scheduling is:

$$R_{\mathsf{SPS}}(v_i) = q_i/\eta = R_{\mathsf{STS}}(v_i) \tag{47}$$

□

Equation (44) shows that the throughput under SPS depends solely on the relationship between Q and η. Recall from Definition 3 that the execution time μ used by our framework is the maximum value over all the actual execution times of the actor. Therefore, if $\eta \bmod Q = 0$, then $R_{\mathsf{SPS}}(v_i)$ is exactly the same as $R_{\mathsf{STS}}(v_i)$ for SDF graphs and CSDF graphs where all the firings of an actor v_i require the same actual execution time. If $\eta \bmod Q \neq 0$ and/or the actor actual execution time differs per firing, then $R_{\mathsf{SPS}}(v_i)$ is lower than $R_{\mathsf{STS}}(v_i)$. These findings illustrate that our framework has high potential since it allows the designer to analytically determine the type of the application (i.e., matched vs. mis-matched) and accordingly to select the proper scheduler needed to deliver the maximum achievable throughput.

Now, we prove the following result regarding matched I/O rates applications:

Corollary 2 *For a matched I/O rates graph G scheduled using its minimum period vector $\boldsymbol{\lambda}^{\min}$, $U_{\max} = 1$.*

Proof Recall from Sect. 3.2.1 that the utilization of a task τ_i is defined as $U_i = C_i/T_i$, where $C_i \leq T_i$. Therefore, the maximum possible value for U_i is when $C_i = T_i$ which leads to $U_i = 1.0$. Now, let v_m be the actor with the maximum product of actor execution time and repetition. That is

$$\mu_m q_m = \max_{v_i \in V}(\mu_i q_i) = \eta \tag{48}$$

The period of v_m is λ_m given by

$$\lambda_m = \frac{Q}{q_m}\left\lceil \frac{\eta}{Q} \right\rceil \tag{49}$$

Now, let us write η as $\eta = pQ + r$, where $p = \eta \div Q$ ($\div$ is the integer division operator), and $r = \eta \bmod Q$. Then, we can re-write (48) as

$$\lambda_m = \frac{Q}{q_m}\left\lceil p + \frac{r}{Q} \right\rceil \tag{50}$$

For matched I/O rates applications, $r = 0$ (see Definition 4). Therefore, (50) can be re-written as

$$\lambda_m = \frac{pQ}{q_m} \tag{51}$$

The utilization of v_m is U_m given by

$$U_m = \frac{\mu_m}{\lambda_m} = \frac{\mu_m q_m}{pQ} \tag{52}$$

Since $r = 0$ and $\eta = pQ = \mu_m q_m$, (52) becomes

$$U_m = \frac{\eta}{\eta} = 1.0 \tag{53}$$

□

Recall from Sect. 3.2.2 that $\beta = \lfloor 1/U_{\max} \rfloor$. It follows from Corollary 2 that $\beta = 1$ for matched I/O rates applications scheduled using their minimum period vectors.

Let ϕ_i be the earliest start time of an actor $v_i \in V$. Then, according to Definitions 5 and 6, the graph latency $L(G)$ is given by:

$$L(G) = \max_{w_{i \rightsquigarrow j} \in \mathcal{W}} \left(\phi_j + \left(g_j^C + 1\right)\lambda_j - \left(\phi_i + g_i^P \lambda_i\right)\right) \tag{54}$$

where ϕ_j and ϕ_i are the earliest start times of the output actor v_j and the input actor v_i, respectively, λ_j and λ_i are the periods of v_j and v_i, and g_j^C and g_i^P are two constants, such that for an output path $w_{i \rightsquigarrow j}$ in which e_r is the first channel and e_u is the last channel, g_i^P and g_j^C are given by:

$$g_i^P = \min\left\{k \in \mathbb{N} : x_i^r(k) > 0\right\} - 1 \tag{55}$$

$$g_j^C = \min\left\{k \in \mathbb{N} : y_j^u(k) > 0\right\} - 1 \tag{56}$$

where x_i^r and y_j^u are production/consumption rates sequences introduced in Sect. 3.

4.4 Handling sporadic input streams

In case the input streams are not strictly periodic, there are several techniques to accommodate the aperiodic nature of the streams. We present here some of these techniques.

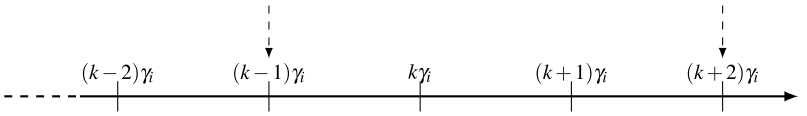


Fig. 11 Occurrence of the maximum inter-arrival time

4.4.1 De-jitter buffers

In case of periodic with jitter input streams, it is possible to use de-jitter buffers to hide the effect of jitter. We assume that a jittery input stream I_i starts at time $t = t_0$ and has a constant inter-arrival time γ_i equal to the input actor period (see Assumption 1-3) and jitter bounds $[\varepsilon_i^-, \varepsilon_i^+]$. The interpretation of the jitter bounds is that the kth sample of the stream is expected to arrive in the interval $[t_0 + k\gamma_i - \varepsilon_i^-, t_0 + k\gamma_i + \varepsilon_i^+]$. If a sample arrives in the interval $[t_0 + k\gamma_i - \varepsilon_i^-, t_0 + k\gamma_i)$, then it is called an *early sample*. On the other hand, if the sample arrives in the interval $(t_0 + k\gamma_i, t_0 + k\gamma_i + \varepsilon_i^+]$, then it is called a *late sample*. It is trivial to show that early samples do not affect the periodicity of the input actor as the samples arrive prior to the actor release time. Late samples, however, pose a problem as they might arrive after an actor is released.

For late samples, it is possible to insert a buffer before each input actor v_i receiving a jittery input stream I_j to hide the effect of jitter. The buffer delays delivering the samples to the input actor by a certain amount of time, denoted by $t_{\text{buffer}}(I_j)$. $t_{\text{buffer}}(I_j)$ has to be computed such that once the input actor is started, it always finds data in the buffer. Assume that ε_i^- and $\varepsilon_i^+ \in [0, \gamma_i]$, then we can derive the minimum value for $t_{\text{buffer}}(I_j)$ and the minimum buffer size. In order to do that, we start with proving the following lemma:

Lemma 5 *Let I_j be a jittery input stream with $\varepsilon_i^-, \varepsilon_i^+ \in [0, \gamma_i]$. Then, the maximum inter-arrival time between any two consecutive samples in I_j, denoted by $t_{\text{MIT}}(I_j)$, satisfies*:

$$t_{\text{MIT}}(I_j) = 3\gamma_i \tag{57}$$

Proof Based on the jitter model, t_{MIT} occurs when the kth sample is early by the maximum value of jitter (i.e., arrives at time $t = k\gamma_i - \gamma_i$), and the $(k + 1)$ sample is late by the maximum value of jitter (i.e., arrives at time $t = (k+1)\gamma_i + \gamma_i$). This is illustrated in Fig. 11. □

Lemma 6 *An input actor $v_i \in V$ is guaranteed to always find an input sample in each of its input de-jitter buffers if the following holds*:

$$t_{\text{buffer}}(I_j) \geq 2\gamma_j \quad \forall I_j \in Z_i \tag{58}$$

Proof During a time interval $(t, t + t_{\text{MIT}}(I_j))$, v_i can fire at most twice. Therefore, it is necessary to buffer up to 2 samples in order to guarantee that the input actor v_i can continue firing periodically when the samples are separated by t_{MIT} time-units. □

Lemma 7 *Let v_i be an input actor and I_j be a jittery input stream to v_i. Suppose that I_j starts at time $t = t_0$ and v_i starts at time $t = t_0 + t_{\text{buffer}}(I_j)$. The de-jitter buffer must be able to hold at least* 3 *samples*.

Proof Suppose that the $(k-1)$ and $(k+1)$ samples arrive late and early, respectively, by the maximum amount of jitter. This means that they arrive at time $t = t_0 + k\gamma_i$. Now, suppose that the kth sample arrives with no jitter. This means that at time $t = t_0 + k\gamma_i$ there are 3 samples arriving. Hence, the de-jitter buffer must be able to store them. During the interval $[t_0 + k\gamma_i, t_0 + (k+1)\gamma_i)$, there are no incoming samples and v_i processes the $(k-1)$ sample. At time $t = t_0 + (k+1)\gamma_i$, the $(k+2)$ sample might arrive which means that there are again 3 samples available to v_i. By the periodicity of v_i and I_j, the previous pattern can repeat. □

The main advantage of the de-jitter buffer approach is that the actors are still treated and scheduled as periodic tasks. However, the major disadvantage is the extra delay encountered by the input stream samples and the extra memory needed for the buffers.

4.4.2 Resource reservation

For sporadic streams in general, we can consider the actors as aperiodic tasks and apply techniques for aperiodic task scheduling from real-time scheduling theory [6]. One popular approach is based on using a *server* task to service the aperiodic tasks. Servers provide resource reservation guarantees and temporal isolation. Several servers have been proposed in the literature (e.g., [1, 27]). The advantages of using servers are the enforced isolation between the tasks, and the ability to support arbitrarily input streams. When using servers, we can schedule each actor using a server which has an execution budget C_s equal to the actor execution time and a period P_s equal to the actor's period.

One particular issue when scheduling the actors using servers is how to generate the aperiodic task requests. For the CSDF model, the requests can be generated when the firing rule of an actor is evaluated as "true" (see Sect. 3). Detecting when the firing rule is evaluated as "true" can be done in the following ways:

1. The underlying operating system (OS) or scheduler has a monitoring mechanism which polls the buffers to detect when an actor has enough data to fire. Once it detects that an actor has enough data to fire, it releases an actor job.
2. Modify the actor implementation such that the polling happens within the actor. In this approach, an actor job is always released at the start of the actor period. When the actor is activated (i.e., running), it checks its input buffers for data. If enough data is available, then it executes its function. Otherwise, it exhausts its budget and waits until the next period. This mechanism is summarized in Fig. 12.

The first approach (i.e., polling by the OS) does not require modifications to the actors' implementations. However, it requires an additional task which always checks all the buffers. This task can become a bottleneck if there are many channels. The second approach is better in terms of scalability and overhead. However, it might cause delays in the processing of the data.

5 Evaluation results

We evaluate our proposed framework in Sect. 4 by performing an experiment on a set of 19 real-life streaming applications. The objective of the experiment is to compare the throughput of streaming applications when scheduled using our strictly periodic scheduling to their maximum achievable throughput obtained via self-timed scheduling. After that, we discuss the implications of our results from Sect. 4 and the throughput comparison experiment. For

```
actor() {

 /* can_fire() returns "true" iff the firing rule is "true" */
 if (can_fire()) {
 read_fifos();

 /* Execute the core function of the actor */
 execute();

 write_fifos();
 }
 else {
 /* wait() exhausts the budget and
 puts the task in sleep state until the next period */
 wait();
 }

}
```

Fig. 12 Polling within the actor to detect when the actor is eligible to fire

brevity, we refer in the remainder of this section to our strictly periodic scheduling/schedule as SPS and the self-timed scheduling/schedule as STS.

The streaming applications used in the experiment are real-life streaming applications coming from different domains (e.g., signal processing, communication, multimedia, etc.). The benchmarks are described in details in the next section.

5.1 Benchmarks

We collected the benchmarks from several sources. The first source is the StreamIt benchmark [30] which contributes 11 streaming applications. The second source is the SDF^3 benchmark [29] which contributes 5 streaming applications. The third source is individual research articles which contain real-life CSDF graphs such as [19, 24, 26]. In total, 19 applications are considered as shown in Table 1. The graphs are a mixture of CSDF and SDF graphs. The actors execution times of the StreamIt benchmark are specified by its authors in clock cycles measured on MIT RAW architecture, while the actors execution times of the SDF^3 benchmark are specified for ARM architecture. For the graphs from [24, 26], the authors do not mention explicitly the actors execution times. As a result, we made assumptions regarding the execution times which are reported below Table 1.

We use the SDF^3 tool-set [29] for several purposes during the experiments. SDF^3 is a powerful analysis tool-set which is capable of analyzing CSDF and SDF graphs to check for consistency errors, compute the repetition vector, compute the maximum achievable throughput, etc. SDF^3 accepts the graphs in XML format. For StreamIt benchmarks, the StreamIt compiler is capable of exporting an SDF graph representation of the stream program. The exported graph is then converted into the XML format required by SDF^3. For the graphs from the research articles, we constructed the XML representation for the CSDF graphs manually.

5.2 Experiment: throughput and latency comparison

In this experiment, we compare the throughput and latency resulting from our SPS approach to the maximum achievable throughput and minimum achievable latency of a streaming

Table 1 Benchmarks used for evaluation

Domain	No.	Application	Source
Signal Processing	1	Multi-channel beamformer	[30]
	2	Discrete cosine transform (DCT)	
	3	Fast Fourier transform (FFT) kernel	
	4	Filterbank for multirate signal processing	
	5	Time delay equalization (TDE)	
Cryptography	6	Data Encryption Standard (DES)	
	7	Serpent	
Sorting	8	Bitonic Parallel Sorting	
Video processing	9	MPEG2 video	
	10	H.263 video decoder	[29]
Audio processing	11	MP3 audio decoder	
	12	CD-to-DAT rate converter (SDF)[a]	[24]
	13	CD-to-DAT rate converter (CSDF)	
	14	Vocoder	[30]
Communication	15	Software FM radio with equalizer	
	16	Data modem	[29]
	17	Satellite receiver	
	18	Digital Radio Mondiale receiver	[19]
Medical	19	Heart pacemaker[b]	[26]

[a]We use two implementations for CD-to-DAT: SDF and CSDF and we refer to them as CD2DAT-S and CD2DAT-C, respectively. The execution times assumed are $\boldsymbol{\mu} = [5, 2, 3, 1, 4, 6]^T$ µs.

[b]We assume the following execution times: Motion Est.: 4 µs, Rate Adapt.: 3 µs, Pacer: 5 µs, and EKG: 2 µs.

application. Recall from Definition 7 that the maximum achievable throughput and minimum achievable latency of a streaming application modeled as a CSDF graph are the ones achieved under self-timed scheduling. In this experiment, we report the throughput for the output actors (i.e., the actors producing the output streams of the application, see Sect. 3). For latency, we report the graph maximum latency according to Definition 6. For SPS, we used the minimum period vector given by Lemma 2. The STS throughput and latency are computed using the SDF^3 tool-set. SDF^3 defines $R_{\mathsf{STS}}(G)$ as the graph throughput under STS, and $R_{\mathsf{STS}}(v_i) = q_i R_{\mathsf{STS}}(G)$ as the actor throughput. Similarly, $L_{\mathsf{STS}}(G)$ denotes the graph latency under self-timed scheduling. We use the `sdf3analysis` tool from SDF^3 to compute the throughput and latency for the STS with auto-concurrency disabled and assuming unbounded FIFO channel sizes. Computing the throughput is performed using the throughput algorithm, while latency is computed using the `latency(min_st)` algorithm.

Now, Table 2 shows the results of comparing the throughput of the output actor for every application under both STS and SPS schedules. The most important column in the table is the last column which shows the *ratio* of the SPS schedule throughput to the STS schedule throughput ($R_{\mathsf{SPS}}(v_{\text{out}})/R_{\mathsf{STS}}(v_{\text{out}})$), where v_{out} denotes the output actor. We clearly see that our SPS delivers the same throughput as STS for 16 out of 19 applications. All these 16 applications are matched I/O rates applications. This result conforms with Theorem 4 proved in Sect. 4. Only three applications (CD2DAT-(S,C) and Satellite) are mis-matched and have lower throughput under our SPS. Table 2 confirms also the observation made by the authors in [30] who reported an interesting finding: *Neighboring actors often have matched*

Table 2 Results of throughput comparison. v_{out} denotes the output actor

Application	$\dot{q}_{\text{out}}$	$R_{\mathsf{STS}}(v_{\text{out}})$	η	Q	$R_{\mathsf{SPS}}(v_{\text{out}})$	$R_{\mathsf{SPS}}(v_{\text{out}})/R_{\mathsf{STS}}(v_{\text{out}})$
Beamformer	1	1.97×10^{-4}	5076	1	1/5076	1.0
DCT	1	2.1×10^{-5}	47616	1	1/47616	1.0
FFT	1	8.31×10^{-5}	12032	1	1/12032	1.0
Filterbank	1	8.84×10^{-5}	11312	1	1/11312	1.0
TDE	1	2.71×10^{-5}	36960	1	1/36960	1.0
DES	1	9.765×10^{-4}	1024	1	1/1024	1.0
Serpent	1	2.99×10^{-4}	3336	1	1/3336	1.0
Bitonic	1	1.05×10^{-2}	95	1	1/95	1.0
MPEG2	1	1.30×10^{-4}	7680	1	1/7680	1.0
H.263	1	3.01×10^{-6}	332046	594	1/332046	1.0
MP3	2	5.36×10^{-7}	3732276	2	1/1866138	1.0
CD2DAT-S	160	1.667×10^{-1}	960	23520	1/147	0.04
CD2DAT-C	160	1.361×10^{-1}	1176	23520	1/147	0.05
Vocoder	1	1.1×10^{-4}	9105	1	1/9105	1.0
FM	1	6.97×10^{-4}	1434	1	1/1434	1.0
Modem	1	6.25×10^{-2}	16	16	1/16	1.0
Satellite	240	2.27×10^{-1}	1056	5280	1/22	0.2
Receiver	288000	4.76×10^{-2}	6048000	288000	1/21	1.0
Pacemaker	64	2.0×10^{-1}	320	320	1/5	1.0

I/O rates. This reduces the opportunity and impact of advanced scheduling strategies proposed in the literature. According to [30], the advanced scheduling strategies proposed in the literature (e.g., [28]) are suitable for *mis-matched* I/O rates applications. Looking into the results in Table 2, we see that our SPS approach performs very-well for matched I/O applications.

Figure 13 shows the ratios of the SPS latency (denoted by $L_{\mathsf{SPS}}(G)$) to the STS latency. For all the applications, the average SPS latency is 5× the STS latency. We also see that the mis-matched applications have large latency which conforms with their sub-optimal throughput. If we exclude the mis-matched applications, then the average SPS latency is 4x the STS latency. For latency-insensitive applications, this is acceptable as long as they can be scheduled using SPS to achieve the maximum achievable throughput. For latency-sensitive applications, reducing the latency can be done by, for example, using the constrained deadline model (see Sect. 3.2.1). The constrained deadline model assigns for each task τ_i a deadline $D_i < T_i$, where T_i is the task period. For example, the Vocoder application has ratio of $L_{\mathsf{SPS}}(G)/L_{\mathsf{STS}}(G) \approx 13.5$ under the implicit-deadline model. This ratio is reduced to 1.0 if the deadline of each task is set to its execution time. However, using the constrained-deadline model requires different schedulability analysis. Therefore, a detailed treatment of how to reduce the latency is outside the scope of this paper.

5.3 Discussion

Suppose that an engineer wants to design an embedded MPSoC which will run a set of matched I/O rates streaming applications. How can he/she determine easily the *minimum* number of processors needed to schedule the applications to deliver the maximum achievable throughput? Our SPS framework in Sect. 4 provides a very fast and accurate answer,

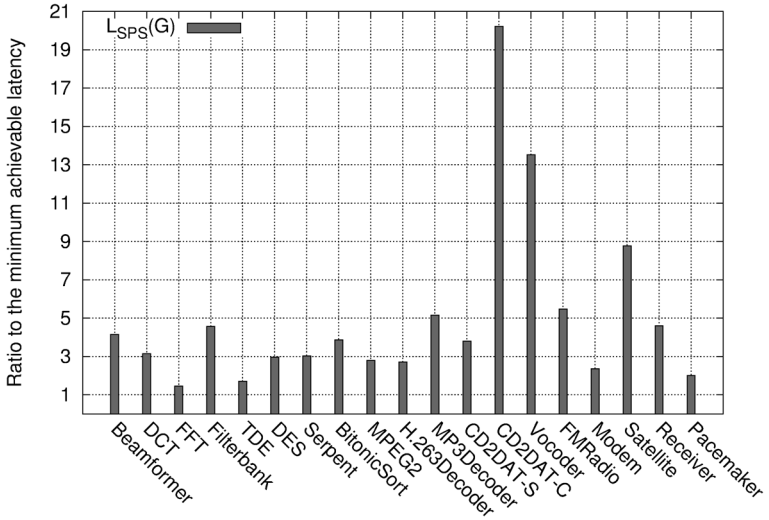


Fig. 13 Results of the latency comparison

thanks to Theorems 3 and 4. They allows easy computation of the minimum number of processors needed by different hard-real-time scheduling algorithms for periodic tasks to schedule any matched I/O streaming application, modeled as an acyclic CSDF graph, while guaranteeing the maximum achievable throughput. Figure 14 illustrates the ability to easily compute the minimum number of processors required to schedule the benchmarks in Table 1 using optimal and partitioned hard-real-time scheduling algorithms for asynchronous sets of implicit-deadline periodic tasks. For optimal algorithms, the minimum number of processors is denoted by M_{OPT} and computed using (9). For partitioned algorithms, we choose P-EDF algorithm combined with First-First (FF) allocation, abbreviated as P-EDF-FF. For P-EDF-FF, the minimum number of processors is computed using (10) ($M_{\mathsf{P\text{-}EDF}}$) and (11) ($M_{\mathsf{PAR}}$). For matched I/O applications scheduled using the minimum periods obtained by Lemma 2, Corollary 2 shows that β defined in Sect. 3.2.2 is equal to 1. This implies that for matched I/O applications, $M_{\mathsf{P\text{-}EDF}} = \lceil 2U_{\text{sum}} - 1 \rceil$ which is approximately twice as M_{OPT} for large values of U_{sum}. M_{PAR} provides less resource usage compared to $M_{\mathsf{P\text{-}EDF}}$ with the restriction that it is valid only for the specific task set τ_G for which it is computed. Another task set $\hat{\tau}_G$ with the same total utilization and maximum utilization factor as τ_G may not be schedulable on M_{PAR} due to the partitioning issues. Comparing M_{PAR} to M_{OPT}, we see that P-EDF-FF requires in around 44 % of the cases an average of 14 % more processors than an optimal algorithm due to the bin-packing effects.

Unfortunately, such easy computation as discussed above of the minimum number of processors is not possible for STS. This is because the minimum number of processors required by STS, denoted by M_{STS}, can not be easily computed with equations such as (9), (10), and (11). Finding M_{STS} in practice requires Design Space Exploration (DSE) procedures to find the best allocation which delivers the maximum achievable throughput. This fact shows one more advantage of using our SPS framework compared to using STS in cases where our SPS gives the same throughput as STS.

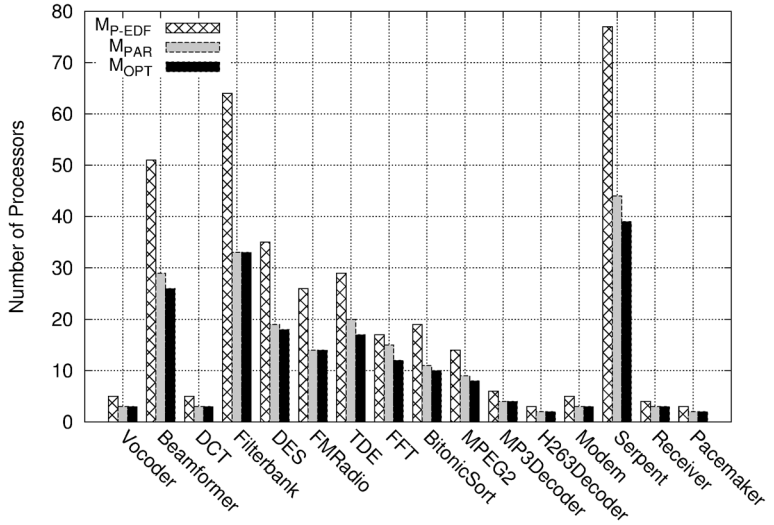


Fig. 14 Number of processors required by an optimal algorithm and P-EDF-FF

6 Conclusions

We prove that the actors of a streaming application, modeled as an acyclic CSDF graph, can be scheduled as periodic tasks. As a result, a variety of hard-real-time scheduling algorithms for periodic tasks can be applied to schedule such applications with a certain guaranteed throughput. We present an analytical framework for computing the periodic task parameters for the actors together with the minimum channel sizes such that a strictly periodic schedule exists. We also show how the proposed framework can handle sporadic input streams. We define formally a class of CSDF graphs called matched I/O rates applications which represents more than 80 % of streaming applications. We prove that strictly periodic scheduling is capable of delivering the maximum achievable throughput for matched I/O rates applications together with the ability to analytically determine the minimum number of processors needed to schedule the applications.

Acknowledgements This work is supported by CATRENE/MEDEA+ 2A718 TSAR (Terascale multicore processor architecture) project. We would like to thank William Thies and Sander Stuijk for their support with StreamIt and SDF^3 benchmarks, respectively.

References

1. Abeni L, Buttazzo G (1998) Integrating multimedia applications in hard real-time systems. In: Proceedings of the 19th IEEE real-time systems symposium (RTSS), pp 4–13. doi:10.1109/REAL.1998.739726
2. Anderson JH, Srinivasan A (2001) Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. In: Proceedings of the 13th Euromicro conference on real-time systems (ECRTS 2001), pp 76–85. doi:10.1109/EMRTS.2001.934004

3. Andersson B, Tovar E (2006) Multiprocessor scheduling with few preemptions. In: Proceedings of the 12th IEEE international conference on embedded and real-time computing systems and applications (RTCSA 2006), pp 322–334. doi:10.1109/RTCSA.2006.45
4. Bekooij M, Hoes R, Moreira O, Poplavko P, Pastrnak M, Mesman B, Mol J, Stuijk S, Gheorghita V, Meerbergen J (2005) Dataflow analysis for real-time embedded multiprocessor system design. In: Dynamic and robust streaming in and between connected consumer-electronic devices, vol 3. Springer, Amsterdam, pp 81–108. doi:10.1007/1-4020-3454-7_4
5. Bilsen G, Engels M, Lauwereins R, Peperstraete J (1996) Cyclo-static dataflow. IEEE Trans Signal Process 44(2):397–408. doi:10.1109/78.485935
6. Buttazzo GC (2011) Hard real-time computing systems, 3rd edn. Springer, Berlin. doi:10.1007/978-1-4614-0676-1
7. Carpenter J, Funk S, Holman P, Srinivasan A, Anderson J, Baruah S (2004) A categorization of real-time multiprocessor scheduling problems and algorithms. In: Leung JYT (ed) Handbook of scheduling: algorithms, models, and performance analysis. CRC Press, Boca Raton. doi:10.1201/9780203489802.ch30
8. Cho H, Ravindran B, Jensen ED (2010) T-L plane-based real-time scheduling for homogeneous multiprocessors. J Parallel Distrib Comput 70(3):225–236. doi:10.1016/j.jpdc.2009.12.003
9. Davis RI, Burns A (2011) A survey of hard real-time scheduling for multiprocessor systems. ACM Comput Surv 43:35:1–35:44. doi:10.1145/1978802.1978814
10. Gerstlauer A, Haubelt C, Pimentel AD, Stefanov TP, Gajski DD, Teich J (2009) Electronic system-level synthesis methodologies. IEEE Trans Comput-Aided Des Integr Circuits Syst 28(10):1517–1530. doi:10.1109/TCAD.2009.2026356
11. Goddard S (1998) On the management of latency in the synthesis of real-time signal processing systems from processing graphs. PhD thesis, University of North Carolina at Chapel Hill
12. Jeffay K, Stanat D, Martel C (1991) On non-preemptive scheduling of periodic and sporadic tasks. In: Proceedings of the 12th real-time systems symposium (RTSS 1991), pp 129–139. doi:10.1109/REAL.1991.160366
13. Karam L, AlKamal I, Gatherer A, Frantz G, Anderson D, Evans B (2009) Trends in multicore DSP platforms. IEEE Signal Process Mag 26(6):38–49. doi:10.1109/MSP.2009.934113
14. Lee EA, Ha S (1989) Scheduling strategies for multiprocessor real-time DSP. In: IEEE global telecommunications conference and exhibition: communications technology for the 1990s and beyond (GLOBECOM 1989), vol 2, pp 1279–1283. doi:10.1109/GLOCOM.1989.64160
15. Lee EA, Messerschmitt DG (1987) Synchronous data flow. Proc IEEE 75(9):1235–1245. doi:10.1109/PROC.1987.13876
16. Levin G, Funk S, Sadowski C, Pye I, Brandt S (2010) DP-FAIR: a simple model for understanding optimal multiprocessor scheduling. In: Proceedings of the 22nd Euromicro conference on real-time systems (ECRTS 2010), pp 3–13. doi:10.1109/ECRTS.2010.34
17. López JM, Díaz JL, García DF (2004) Utilization bounds for EDF scheduling on real-time multiprocessor systems. Real-Time Syst 28:39–68. doi:10.1023/B:TIME.0000033378.56741.14
18. Martin G (2006) Overview of the MPSoC design challenge. In: Proceedings of the 43rd annual design automation conference (DAC 2006), pp 274–279. doi:10.1145/1146909.1146980
19. Moonen A, Bekooij M, van den Berg R, van Meerbergen J (2008) Cache aware mapping of streaming applications on a multiprocessor system-on-chip. In: Proceedings of the conference on design, automation and test in Europe (DATE 2008), pp 300–305. doi:10.1145/1403375.1403448
20. Moreira O, Mol JD, Bekooij M, van Meerbergen J (2005) Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix. In: Proceedings of the 11th IEEE real time and embedded technology and applications symposium (RTAS 2005), pp 332–341. doi:10.1109/RTAS.2005.33
21. Moreira O, Valente F, Bekooij M (2007) Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In: Proceedings of the 7th ACM & IEEE international conference on embedded software (EMSOFT 2007), pp 57–66. doi:10.1145/1289927.1289941
22. Moreira OM, Bekooij MJG (2007) Self-timed scheduling analysis for real-time applications. EURASIP J Adv Signal Process 2007:1–15. doi:10.1155/2007/83710
23. Nollet V, Verkest D, Corporaal H (2010) A safari through the MPSoC run-time management jungle. Signal Process Syst 60:251–268. doi:10.1007/s11265-008-0305-4
24. Oh H, Ha S (2004) Fractional rate dataflow model for efficient code synthesis. J VLSI Signal Process 37:41–51. doi:10.1023/B:VLSI.0000017002.91721.0e
25. Parks T, Lee E (1995) Non-preemptive real-time scheduling of dataflow systems. In: Proceedings of the 1995 international conference on acoustics, speech, and signal processing (ICASSP 1995), vol 5, pp 3235–3238. doi:10.1109/ICASSP.1995.479574
26. Pellizzoni R, Meredith P, Nam MY, Sun M, Caccamo M, Sha L (2009) Handling mixed-criticality in SoC-based real-time embedded systems. In: Proceedings of the 7th ACM international conference on embedded software (EMSOFT 2009), pp 235–244. doi:10.1145/1629335.1629367

27. Sprunt B, Sha L, Lehoczky J (1989) Aperiodic task scheduling for hard-real-time systems. Real-Time Syst 1:27–60. doi:10.1007/BF02341920
28. Sriram S, Bhattacharyya SS (2009) Embedded multiprocessors: scheduling and synchronization, 2nd edn. CRC Press, Boca Raton. doi:10.1201/9781420048025
29. Stuijk S, Geilen M, Basten T (2006) SDFT^3: SDF for free. In: Proceedings of the 6th international conference on application of concurrency to system design (ACSD 2006), pp 276–278. doi:10.1109/ACSD.2006.23
30. Thies W, Amarasinghe S (2010) An empirical characterization of stream programs and its implications for language and compiler design. In: Proceedings of the 19th international conference on parallel architectures and compilation techniques (PACT 2010), pp 365–376. doi:10.1145/1854273.1854319