Hard-Real-Time Scheduling of Data-Dependent Tasks in Embedded Streaming Applications

Mohamed Bamakhrama
mohamed@liacs.nl

Todor Stefanov
stefanov@liacs.nl

Leiden Institute of Advanced Computer Science
Leiden University, Leiden, The Netherlands

ABSTRACT

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 modern embedded streaming applications. This is because a modern streaming application is typically modeled as a directed graph 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. Therefore, in this paper we investigate the applicability of hard-real-time scheduling theory for periodic tasks to streaming applications modeled as acyclic Cyclo-Static Dataflow (CSDF) graphs. In such graphs, the actors are data-dependent, however, we analytically prove that they (i.e. the actors) can be scheduled as implicit-deadline 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 compare the throughput resulting from such scheduling approach to the maximum achievable throughput of an application for a set of 19 real streaming applications. We find that in more than 80% of the cases, the throughput resulting from our approach is equal to the maximum achievable throughput.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: [Real-time and embedded systems]; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

General Terms

Algorithms, Design, Performance

Keywords

Real-time multiprocessor scheduling, streaming applications

1. INTRODUCTION

The ever-increasing complexity of embedded systems realized as Multi-Processor Systems-on-Chips (MPSoCs) is imposing several challenges on systems designers [17]. Two major challenges in designing streaming software for embedded MPSoCs are: 1) How to express 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 [9]. In MoC-based design, the application can be modeled as a directed graph where nodes represents 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 [4] which extends the well-known Synchronous Data Flow (SDF) model [14].

Unfortunately, no such de-facto solution exists yet for the second challenge of processor allocation [22]. For a long time, self-timed scheduling was considered the most appropriate policy for streaming applications modeled as dataflow graphs [13, 26]. 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 [12]. Most of the existing run-time scheduling solutions assume applications modeled as task graphs and provide best-effort or soft-real-time services [22]. Few run-time scheduling solutions exist which support applications modeled using a MoC and provide hard-real-time services [10, 3, 19, 20]. However, these solutions either use simple MoCs such as SDF/PGM graphs or use Time-Division Multiplexing (TDM)/Round-Robin (RR) scheduling combined with heuristics. Several algorithms from the hard-real-time multiprocessor scheduling theory [7] can perform *fast* admission and scheduling decisions for incoming applications while providing hard-real-time services. Moreover, these algorithms provide *temporal isolation* which is the ability to dynamically start/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 [7]. 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,

EMSOFT'11, October 9–14, 2011, Taipei, Taiwan.

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 implicit-deadline 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 consider streaming applications modeled as acyclic CSDF graphs with periodic input streams. Typically, embedded streaming applications receive their input samples from hardware sensors/devices which generate these samples periodically. 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 [28]. 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 with periodic input streams, determine whether it is possible to execute the graph actors as implicit-deadline periodic tasks. An implicit-deadline periodic task τ_i is defined by a 3-tuple $\tau_i = (C_i, T_i, S_i)$. 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)* [21] to avoid confusion with the term *periodic scheduling* used in the dataflow scheduling theory to refer to a repetitive 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 Contributions of this Paper

Given a streaming application modeled as an acyclic CSDF graph, we analytically prove that it is possible to execute the graph actors as implicit-deadline 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 implicit-deadline periodic tasks. Upon empirical evaluation, we demonstrate that our strictly periodic scheduling approach yields the maximum achievable throughput for a class of applications called *matched I/O rates applications* which represents the majority of streaming applications. Applying our approach to matched I/O rates applications enables using a plethora of schedulability tests developed in the real-time scheduling theory [7] 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: Section 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 Section 4. Finally, Section 6 ends the paper with conclusions.

2. RELATED WORK

Parks and Lee [24] 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 [11], and 2) they considered SDF graphs which are a subset of the more general CSDF graphs.

Goddard [10] 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 [10] is not applicable to CSDF graphs.

Bekooij et al. presented a dataflow analysis for embedded real-time multiprocessor systems [3]. 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 Homogeneous SDF (HSDF) graphs in [19, 20, 21]. They formulated a resource allocation heuristic [19] and a TDM scheduler combined with static allocation policy [20]. Their TDM scheduler improves the one proposed in [3]. In [21], 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 HSDF graph in order to enforce a periodic execution of every actor as a Linear Programming (LP) problem.

Our approach differs from [3, 19, 20] in: 1) using the periodic task model which allows applying a variety of proven hard-real-time scheduling algorithms for multiprocessors, 2) providing direct analysis of acyclic graphs for which the maximum cycle mean analysis can not be applied without converting the graph into a cyclic one, and 3) using the CSDF model which is more expressive than SDF graphs. Compared to [21], our approach differs in: 1) we derive an equation which always finds the actors periods that ensure periodic execution and scheduling. The authors in [21] indicated that deriving the schedule for the original SDF graph from the schedule of the HSDF graph requires adding extra constraints to the LP problem which can result in an infeasible problem, and 2) we perform the analysis directly on the CSDF graph instead of the equivalent SDF/HSDF graphs. Thus, our approach is faster and more general since it avoids the exponentially complex conversion from (C)SDF to HSDF.

3. BACKGROUND

We introduce in this section the CSDF model, system model, and hard-real-time scheduling algorithms for periodic tasks. The material presented is this section is essential for understanding our contribution in Section 4.

3.1 The Cyclo-Static Dataflow (CSDF) Model

Cyclo-Static Dataflow (CSDF) is a MoC for describing applications in the digital signal processing domain [4]. It extends the popular Synchronous Data Flow (SDF) model proposed in [14]. The main difference between CSDF and SDF is that CSDF supports algorithms with a *cyclically* changing but predefined behavior.

3.1.1 Formal Definition

Bilsen et al. defined the CSDF model in [4] as a directed graph $G = \langle V, E \rangle$, where V is a set of actors and E is a set of communication channels. Actors represent functions that transform the input data streams into output streams. The communication channels carry streams of data. An atomic data object is called a *token*. A communication channel is a first-in, first-out (FIFO) queue with unbounded capacity.

In CSDF, every actor $v_j \in V$ has an execution sequence $[f_j(1), f_j(2), \cdots, f_j(P_j)]$ of length P_j. The interpretation of this sequence is: The *n*th 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 $[x_j^u(1), x_j^u(2), \cdots, x_j^u(P_j)]$. The *n*th 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 $[y_k^u(1), y_k^u(2), \cdots, y_k^u(P_j)]$. The firing rule of a CSDF actor v_k is evaluated as "true" for its *n*th firing iff all its input channels contain at least $y_k^u(((n-1) \bmod P_j)+1)$ tokens.

Example 1. Figure 1 shows a CSDF graph consisting of four actors and four communication channels.

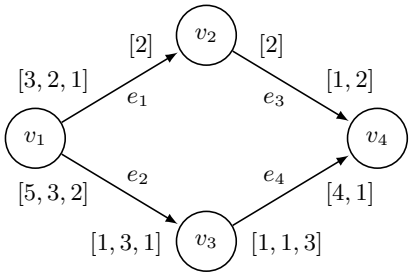

Figure 1: Example CSDF graph

3.1.2 Compile-Time Scheduling

An important property of the CSDF model is the ability to derive at compile-time a schedule for the actors. Compile-time scheduling has been an attractive property of dataflow models because it removes the need for a run-time scheduler [13]. In order to derive a compile-time schedule for a CSDF graph, it has to be *consistent* and *live*. We use the consistency property of the CSDF graph and extend it later in Section 4 to allow periodic scheduling of CSDF graphs. In this paper, we use the notations relating to CSDF graphs as shown in Table 1.

Definition 1. Given a connected CSDF graph G, a *valid static schedule* for G is a schedule that can be repeated infinitely on the incoming sample stream and where the amount of data in the buffers remains bounded. A vector $\vec{q} = [q_1, q_2, \cdots, q_N]^T$, 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.

Table 1: Notation for CSDF graphs (Same as [4])

N	Number of actors in a CSDF graph G
P_j	Length of execution sequence of actor v_j
$x_j^u(n)$	Number of tokens produced by actor v_j on channel e_u during its *n*th invocation
$y_k^u(n)$	Number of tokens consumed by actor v_k from channel e_u during its *n*th invocation
$X_j^u(n)$	Number of tokens produced by actor v_j on channel e_u during the first *n* invocations $= \sum_{l=1}^{n} x_j^u(l)$
$Y_k^u(n)$	Number of tokens consumed by actor v_k from channel e_u during the first *n* invocations $= \sum_{l=1}^{n} y_k^u(l)$

The repetition vector of G in which all the elements are relatively prime[1] is called the *basic repetition vector* of G, denoted $\vec{q}$. 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 (FROM [4]). *In a CSDF graph G, a repetition vector $\vec{q} = [q_1, q_2, \cdots, q_N]^T$ is given by*

$$\vec{q} = \mathbf{P} \cdot \vec{r}, \qquad \textit{with} \qquad P_{jk} = \begin{cases} P_j & \textit{if } j = k \\ 0 & \textit{otherwise} \end{cases} \tag{1}$$

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

$$\mathbf{\Gamma} \cdot \vec{r} = \vec{0} \tag{2}$$

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

$$\Gamma_{uj} = \begin{cases} X_j^u(P_j) & \textit{if actor } v_j \textit{ produces on ch. } e_u \\ -Y_j^u(P_j) & \textit{if actor } v_j \textit{ consumes from ch. } e_u \\ 0 & \textit{Otherwise.} \end{cases} \tag{3}$$

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 $q_i \in \vec{q}$.

COROLLARY 1 (FROM [4]). *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 [4] 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. □

Example 2. For the CSDF graph shown in Figure 1

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

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

3.2 System Model and Scheduling Algorithms

In this section, we introduce the system model and hard-real-time scheduling algorithms for periodic tasks.

[1] i.e. $\gcd(q_1, q_2, \cdots, q_N) = 1$.

3.2.1 System Model

A system Ψ consists of a set $\Lambda = \{\pi_1, \pi_2, \cdots, \pi_M\}$ of M homogeneous processors. The processors execute a task set $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_N\}$ of N periodic tasks, and a task may be preempted at any time. A periodic task $\tau_i \in \Gamma$ is defined by a 4-tuple $\tau_i = (C_i, T_i, D_i, S_i)$, where C_i is the worst-case execution time, T_i is the task period (where $T_i \geq C_i$), D_i is the deadline of τ_i, and S_i is the start time 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 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 $k \in \mathbb{N}_0$. τ_i is said to have *implicit-deadline* if $D_i = T_i$. In this case, the deadline of the current invocation is the time of the next invocation. If $D_i \leq T_i$, then τ_i is said to have *constrained-deadline*. For a task set Γ, Γ is said to be *synchronous* if all the tasks in Γ have the same start time. Otherwise, Γ is said to be *asynchronous*.

The utilization of task τ_i, denoted U_i where $U_i \in (0, 1]$, is defined as $U_i = C_i/T_i$. For a task set Γ, U_Γ is the total utilization of Γ given by $U_\Gamma = \sum_{\tau_i \in \Gamma} U_i$. The maximum utilization factor in Γ, denoted $U_\Gamma^{\max}$, is defined as $U_\Gamma^{\max} = \max_{\tau_i \in \Gamma}(U_i)$.

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

3.2.2 Scheduling Asynchronous Set of Implicit Deadline Periodic Tasks

Given a system Ψ and a task set Γ, a *periodic schedule* is one that allocates a processor to a task $\tau_i \in \Gamma$ 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 Γ to be scheduled on Ψ to meet all the deadlines (i.e. Γ is *feasible*) is [7]:

$$U_\Gamma \leq M \tag{4}$$

The problem of constructing a periodic schedule for Γ can be solved using several algorithms [7]. 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 Γ on Ψ. Several global and hybrid algorithms were proven optimal for scheduling asynchronous sets of implicit-deadline periodic tasks, e.g. [1, 2, 6, 15]. The minimum number of processors needed to schedule Γ using an optimal scheduling algorithm, denoted M_{OPT}, is given by:

$$M_{\mathsf{OPT}} = \lceil U_\Gamma \rceil \tag{5}$$

Partitioned algorithms are known to be non-optimal for scheduling implicit-deadline periodic tasks [5]. However, they have the advantage of not requiring task migration. One prominent example of partitioned scheduling is the Partitioned Earliest-Deadline-First (PEDF) algorithm. EDF is known to be optimal for scheduling arbitrary task sets on a uniprocessor system [8]. In a multiprocessor system, EDF can be combined with different processor allocation algorithms (e.g. Bin-packing heuristics such as First-Fit (FF), Worst-Fit (WF)). López et al. derived in [16] the worst-case utilization bounds for a task set Γ to be schedulable using PEDF. 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 Γ under PEDF, denoted M_{PEDF}:

$$M_{\mathsf{PEDF}} \geq \begin{cases} 1 & \text{if } U_\Gamma \leq 1 \\ \min\left(\left\lceil \frac{N}{\beta} \right\rceil, \left\lceil \frac{(\beta+1)U_\Gamma - 1}{\beta} \right\rceil\right) & \text{if } U_\Gamma > 1, \end{cases} \tag{6}$$

where $\beta = \lfloor 1/U_\Gamma^{\max} \rfloor$. A task set Γ with total utilization U_Γ and maximum utilization factor $U_\Gamma^{\max}$ is always guaranteed to be schedulable on M_{PEDF} processors. Since M_{PEDF} is derived based on a sufficient test, it is important to note that Γ may be schedulable on less number of processors. We define M_{PAR} as the minimum number of processors on which Γ 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:

$$M_{\mathsf{PAR}} = \min_{x \in \mathbb{N}} \{x : B \text{ is } x\text{-partition of } \Gamma \text{ and } U_y \leq 1 \forall y \in B\} \tag{7}$$

M_{PAR} is specific to the task set Γ for which it is computed. Another task set $\hat{\Gamma}$ with the same total utilization and maximum utilization factor as Γ 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 implicit-deadline periodic tasks. We prove the existence of strictly periodic schedules for acyclic CSDF graphs with a single periodic input stream. Nevertheless, the presented results are applicable also to acyclic CSDF graphs with multiple periodic input streams.

4.1 Notations

We introduce the following notations:

M	Number of processors
G	A consistent and live CSDF graph with a single periodic input stream
V	Set of actors in G
E	Set of communication channels in G
N	Number of actors in G

4.2 Existence of a Strictly Periodic Schedule

Definition 3. For G, an *execution vector* $\vec{\mu}$, where $\vec{\mu} \in \mathbb{N}^N$, 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

$$\mu_j = \max_{k=1}^{P_j} \left(T^R \sum_{l \in I_j} y_j^l(k) + T^W \sum_{r \in O_j} x_j^r(k) + T_j^C(k) \right) \tag{8}$$

where T^R is the worst-case time needed to read a single token from an input channel, I_j is the set of input channels of v_j, T^W is the

worst-case time needed to write a single token to an output channel, O_j is the set of output channels of v_j, and $T_j^C(k)$ is the worst-case computation time of v_j in firing k.

Definition 4. An actor $v_i \in V$ is *strictly periodic* iff the time period between any two consecutive firings is constant.

Definition 5. For G, a *period vector* $\vec{\lambda}$, where $\vec{\lambda} \in \mathbb{N}^N$, represents the periods, measured in time-units, of the actors in G. $\lambda_j \in \vec{\lambda}$ is the period of actor $v_j \in V$. $\vec{\lambda}$ is given by the solution to both

$$q_1\lambda_1 = q_2\lambda_2 = \cdots = q_{N-1}\lambda_{N-1} = q_N\lambda_N \tag{9}$$

and

$$\vec{\lambda} - \vec{\mu} \geq \vec{0}, \tag{10}$$

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

LEMMA 2. *The minimum period vector of G, denoted $\vec{\lambda}^{min}$, is given by*

$$\lambda_i^{min} = \frac{L}{q_i}\left\lceil \frac{\max_{v_j \in V}(\mu_j q_j)}{L} \right\rceil \text{ for } v_i \in V, \tag{11}$$

where $L = \mathsf{lcm}(q_1, q_2, \cdots, q_N)$ (lcm stands for least-common-multiple).

PROOF. Equation 9 can be re-written as:

$$\mathbf{Q}\vec{\lambda} = \vec{0}, \tag{12}$$

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

$$Q_{ij} = \begin{cases} q_1 & \text{if } j = 1 \\ -q_j & \text{if } j = i+1 \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

Observe that $\text{nullity}(\mathbf{Q}) = 1$. Thus, there exists a single vector which forms the basis of the null-space of $\mathbf{Q}$. 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:

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

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

$$\lambda_k = \mathsf{lcm}(q_1, q_2, \cdots, q_N)/q_k$$

Thus, the minimum $\vec{\lambda} \in \mathbb{N}$ that solves Equation 9 is given by

$$\lambda_i = \mathsf{lcm}(q_1, q_2, \cdots, q_N)/q_i \text{ for } v_i \in V \tag{14}$$

Let $L = \mathsf{lcm}(q_1, q_2, \cdots, q_N)$, and let $\hat{\vec{\lambda}}$ be the solution given by Equation 14. Equations 9 and 10 can be re-written as:

$$\mathbf{Q}(c\hat{\vec{\lambda}}) = \vec{0} \tag{15}$$

$$c\hat{\lambda}_1 \geq \mu_1, c\hat{\lambda}_2 \geq \mu_2, \cdots, c\hat{\lambda}_N \geq \mu_N \tag{16}$$

where $c \in \mathbb{N}$. Equation 16 can be re-written as:

$$c \geq \mu_1 q_1/L, c \geq \mu_2 q_2/L, \cdots, c \geq \mu_N q_N/L \tag{17}$$

It follows from Equation 17 that c must be greater than or equal to $\max_{v_i \in V}(\mu_i q_i)/L$. However, $\max_{v_i \in V}(\mu_i q_i)/L$ is not always guaranteed to be an integer. As a result, the value is rounded by taking the ceil. It follows that the minimum $\vec{\lambda}$ which satisfies both of Equation 9 and Equation 10 is given by

$$\lambda_i = L/q_i\lceil \max_{v_j \in V}(\mu_j q_j)/L \rceil \text{ for } v_i \in V$$

□

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

PROOF. We use Algorithm 1 to find the levels of G.

Algorithm 1 ACYCLIC-CSDF-GRAPH-LEVELS(G)

Require: Acyclic CSDF graph $G = \langle V, E \rangle$
1: $i \leftarrow 1$
2: **while** $V \neq \emptyset$ **do**
3: $A_i = \{v_j \in V \mid v_j \text{ has no incoming channels}\}$
4: $Z_i = \{e_u \in E \mid \exists v_k \in A_i : v_k \text{ is the source of } e_u\}$
5: $V \leftarrow V \setminus A_i$
6: $E \leftarrow E \setminus Z_i$
7: $i \leftarrow i + 1$
8: **end while**
9: $d \leftarrow i - 1$
10: **return** d disjoint sets $\{A_1, A_2, ..., A_d\}$, where $\bigcup_{i=1}^{d} A_i = V$

An actor $v_i \in A_j$ is said to be a level-j actor. Since G has a single input stream, it follows that A_1 contains a single actor.

Let $\alpha = q_1\lambda_1 = q_2\lambda_2 = \cdots = q_N\lambda_N$ (See Equation 9), and let v_1 denote the level-1 actor. Suppose that the input stream to v_1 has a period equal to λ_1. It follows that v_1 can execute periodically since its input is always available when it fires. By Definition 2, v_1 will complete one iteration when it fires q_1 times. Assume that v_1 starts executing at time $t = 0$. Then, by time $t = q_1\lambda_1 = \alpha$, v_1 is guaranteed to finish one iteration. According to Theorem 1, v_1 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 Equation 9, firing v_k for q_k times with a period λ_k takes α time-units. Thus, starting level-2 actors at time $t = q_1\lambda_1 = \alpha$ guarantees that they can execute periodically with their periods given by Definition 5 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 d levels, a schedule $\mathcal{S}_1$ (shown in Figure 2) is constructed in which all the actors that belong to A_i are started at *start time*, denoted ϕ_i, given by

$$\phi_i = (i-1)\alpha \tag{18}$$

time	$[0,\alpha)$	$[\alpha,2\alpha)$	$[2\alpha,3\alpha)$	$\cdots$	$[(d-1)\alpha, d\alpha)$
level	$A_1(1)$	$A_2(1)$	$A_3(1)$	$\cdots$	$A_d(1)$

Figure 2: Schedule $\mathcal{S}_1$

$A_j(k)$ denotes level-j actors executing their kth iteration. For example, $A_2(1)$ denotes level-2 actors executing their first iteration. At time $t = d\alpha$, G completes one iteration. It is trivial to observe from $\mathcal{S}_1$ that as soon as v_1 finishes one iteration, it can immediately start executing the next iteration since its input stream arrives periodically. If v_1 starts its second iteration at time $t = \alpha$, its 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 $\mathcal{S}_2$ shown in Figure 3.

Now, the overlapping can be applied d times on schedule $\mathcal{S}_1$ to yield a schedule $\mathcal{S}_d$ as shown in Figure 4.

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

Figure 3: Schedule $\mathcal{S}_2$

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

Figure 4: Schedule $\mathcal{S}_d$

Starting from time $t = d\alpha$, a schedule $\mathcal{S}_\infty$ can be constructed as shown in Figure 5.

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

Figure 5: Schedule $\mathcal{S}_\infty$

In schedule $\mathcal{S}_\infty$, every actor v_i is fired every λ_i time-unit once it starts. The start time defined in Equation 18 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 $\mathcal{S}_\infty$ 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 $\mathcal{S}_\infty$ executes with bounded memory buffers. In $\mathcal{S}_\infty$, the largest delay in consuming the tokens occurs for a channel $e_u \in E$ connecting a level-1 actor and a level-d actor. This is illustrated in Figure 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-d destination actor after $(d-1)\alpha$ time-units. In this case, e_u must be able to store at least $(d-1)X_1^u(q_1)$ tokens. However, starting from time $t = d\alpha$, both of the level-1 and level-d 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 $dX_1^u(q_1)$ tokens. Similarly, if a level-2 actor, denoted v_2, is connected directly to a level-d actor via channel e_v, then e_v must be able to store at least $(d-1)X_2^v(q_2)$ 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 $\mathcal{S}_\infty$ at most:

$$b_u = (j - i + 1)X_k^u(q_k) \tag{19}$$

tokens, where v_k is the level-i actor, and $q_k \in \vec{q}$. 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 Figure 1. Assume that the CSDF graph has an execution vector $\vec{\mu} = [5, 2, 3, 2]^T$. Given $\vec{q} = [3, 3, 6, 4]^T$ as computed in Example 2, we use Equation 11 to find $\vec{\lambda}^{\min} = [8, 8, 4, 6]^T$. Figure 6 illustrates the periodic schedule of the actors for the first 71 time-units. Applying Algorithm 1 on the graph results in 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.

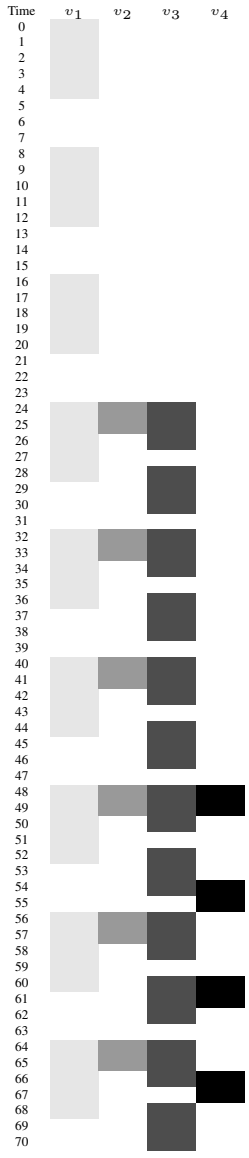

Figure 6: Strictly periodic schedule for the CSDF graph shown in Figure 1

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 ini-

tial 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.

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

$$\phi_j = \begin{cases} 0 & \text{if } \Omega(v_j) = \emptyset \\ \max_{v_i \in \Omega(v_j)} (\phi_{i \to j}) & \text{if } \Omega(v_j) \neq \emptyset \end{cases} \tag{20}$$

where $\Omega(v_j)$ is the set of predecessors of v_j, and $\phi_{i\to j}$ is given by

$$\phi_{i \to j} = \min_{t \in [\phi_i, \phi_i + \alpha]} \{t : \underset{[\phi_i, t+k)}{\mathsf{prd}}(v_i) \geq \underset{[\phi_i, t+k]}{\mathsf{cns}}(v_j)\ \forall k \in [0, \alpha]\}, \tag{21}$$

where ϕ_i is the earliest start time of a predecessor actor v_i, $\alpha = q_i\lambda_i = q_j\lambda_j$, $\mathsf{prd}_{[t_s,t_e)}(v_i)$ is the number of tokens produced by v_i during the time interval $[t_s, t_e)$, and $\mathsf{cns}_{[t_s,t_e]}(v_j)$ is the number of tokens consumed by v_j during the time interval $[t_s, t_e]$.

PROOF. Theorem 2 proved that starting a level-k actor v_j at a start time given by:

$$\phi_j = (k-1)\alpha \tag{22}$$

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 22 can be re-written as:

$$\phi_j = \begin{cases} 0 & \text{if } \Omega(v_j) = \emptyset \\ \max_{v_i \in \Omega(v_j)}(\phi_i) + \alpha & \text{if } \Omega(v_j) \neq \emptyset \end{cases} \tag{23}$$

The equivalence is immediate by observing that Algorithm 1 guarantees that a level-k actor, where $k > 1$, has a level-$(k-1)$ predecessor. Hence, applying Equation 23 to a level-k actor, where $k > 1$, yields:

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

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

$$\phi_j \leq \max_{v_i \in \Omega(v_j)}(\phi_i) + \alpha \tag{24}$$

ϕ_j has also a lower-bound by observing that in order to have a periodic execution, an actor v_j can not start before all its predecessors start. That is:

$$\max_{v_i \in \Omega(v_j)}(\phi_i) \leq \phi_j \leq \max_{v_i \in \Omega(v_j)}(\phi_i) + \alpha \tag{25}$$

Equation 25 can be re-written as:

$$\phi_j = \max_{v_i \in \Omega(v_j)}(\phi_i) + c, c \in [0, \alpha] \tag{26}$$

The choice of c in Equation 26 has to be made such that v_j is guaranteed to start only after all its predecessors have generated enough data to let v_j execute in a strictly periodic way. Thus, Equation 26 can be re-written as:

$$\begin{aligned} \phi_j &= \max_{v_i \in \Omega(v_j)}(\phi_{i\to j}), \phi_{i\to j} = \phi_i + c, c \in [0, \alpha] \\ &= \max_{v_i \in \Omega(v_j)}(\phi_{i\to j}), \phi_{i\to j} = \hat{t}, \hat{t} \in [\phi_i, \phi_i + \alpha] \end{aligned} \tag{27}$$

If we select $\hat{t}$ in Eq. 27 such that $\mathsf{prd}_{[\phi_i,\hat{t})}(v_i) \geq \mathsf{cns}_{[\phi_i,\hat{t}]}(v_j)$, then this guarantees that v_j can fire once at time $t = \hat{t}$. If $\hat{t}$ is also selected such that $\mathsf{prd}_{[\phi_i,\hat{t}+k)}(v_i) \geq \mathsf{cns}_{[\phi_i,\hat{t}+k]}(v_j)$ for all $k \in (0, \alpha]$, then this guarantees that v_j can fire at times $t = \hat{t} + \lambda_j, \hat{t} + 2\lambda_j, \cdots, \hat{t} + \alpha$. Thus, the value of $\hat{t}$ given by Equation 21 guarantees that once v_j is started, it always finds enough data to fire. As a result, v_j executes in a strictly periodic way. Since $\hat{t}$ is selected in Equation 21 as the minimum value that satisfies the inequality, using a smaller value than $\hat{t}$ will cause the inequality of Equation 21 to not hold. If the inequality does not hold for any $k \in [0, \alpha]$, then v_j can not execute in a strictly periodic manner since at some time point in $[\hat{t}, \hat{t} + \alpha]$ it will not have enough tokens to fire and it will be blocked on reading. As a result, $\hat{t}$ given by Equation 21 is the earliest start time to guarantee strictly periodic execution of the actors. □

4.3.2 Minimum Buffer Sizes

THEOREM 4. *For an acyclic 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 periodic schedule is given by*

$$b_u = \max_{k \in [0,\alpha]} \left(\underset{[\phi_i, \phi_j + k)}{\mathsf{prd}}(v_i) - \underset{[\phi_i, \phi_j + k)}{\mathsf{cns}}(v_j) \right), \tag{28}$$

where $\mathsf{prd}_{[t_s,t_e)}(v_i)$ is the number of tokens produced by v_i during the time interval $[t_s, t_e)$, and $\mathsf{cns}_{[t_s,t_e)}(v_j)$ is the number of tokens consumed by v_j during the time interval $[t_s, t_e)$.

PROOF. Equation 28 tracks the maximum cumulative number of unconsumed tokens in e_u during the time interval $[\phi_i, \phi_j + \alpha)$.

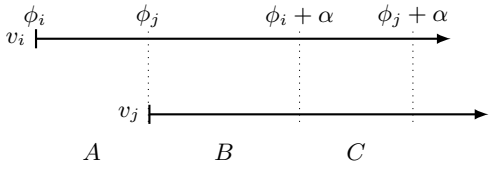

Figure 7: Execution time-lines of v_i and v_j

Figure 7 illustrates the execution time-lines of v_i and v_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 $\mathsf{prd}_{[\phi_i,\phi_j)}(v_i)$. 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 Figure 7). 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 Figure 7 repeats. Thus, b_u is also sufficient and minimum for any $t \geq \phi_j + \alpha$. □

COROLLARY 2. *For an acyclic G, let Γ_G be a task set such that $\tau_i \in \Gamma_G$ corresponds to $v_i \in V$. τ_i is given by:*

$$\tau_i = (\mu_i, \lambda_i, \phi_i), \tag{29}$$

where $\mu_i \in \vec{\mu}$, $\lambda_i \in \vec{\lambda}^{min}$ given by Equation 11, and ϕ_i is the earliest start time of v_i given by Equation 20. Γ_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 Equation 28*
2. *Γ_G satisfies the schedulability test of $\mathcal{A}$*

PROOF. Follows from Theorems 2, 3, and 4. □

Example 4. This is an example to illustrate Theorems 3, 4, and Corollary 2. First, we calculate the earliest start times and the corresponding minimum buffer sizes for the CSDF graph shown in Figure 1. Applying Theorems 3 and 4 on the CSDF graph results in:

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

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 $\vec{\mu}$ and $\vec{\lambda}^{\min}$ computed in Example 3, we construct a task set $\Gamma_G = \{(5, 8, 0), (2, 8, 8), (3, 4, 8), (2, 6, 20)\}$. We compute the minimum number of required processors to schedule Γ_G according to Equations 5, 6, and 7:

$$M_{\mathsf{OPT}} = \lceil 5/8 + 2/8 + 3/4 + 2/6 \rceil = \lceil 47/24 \rceil = 2$$
$$M_{\mathsf{PEDF}} = \min\{\lceil 4/1 \rceil, \lceil (2 \times 47/24 - 1)/1 \rceil\} = 3$$
$$M_{\mathsf{PAR}} = \min_{x \in \mathbb{N}}\{x : B \text{ is } x\text{-partition of } \Gamma \text{ and } U_y \leq 1 \forall y \in B\} = 3$$

Γ_G is schedulable using an optimal scheduling algorithm on 2 processors, and is schedulable using PEDF on 3 processors.

5. EVALUATION RESULTS

We evaluate our proposed framework in Section 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 Section 4 and the throughput comparison experiment. For 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 which come 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 [28] which contributes 11 streaming applications. The second source is the SDF^3 benchmark [27] which contributes 5 streaming applications. The third source is individual research articles which contain real-life CSDF graphs such as [23, 18, 25]. In total, 19 applications are considered as shown in Table 2. 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 [23, 25], 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 2.

Table 2: Benchmarks used for evaluation

Domain	No.	Application	Source
Signal Processing	1	Multi-channel beamformer	[28]
	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	[27]
Audio processing	11	MP3 audio decoder	[23]
	12	CD-to-DAT rate converter (SDF)[1]	
	13	CD-to-DAT rate converter (CSDF)	
	14	Vocoder	[28]
Communication	15	Software FM radio with equalizer	[27]
	16	Data modem	
	17	Satellite receiver	
	18	Digital Radio Mondiale receiver	[18]
Medical	19	Heart pacemaker[2]	[25]

[1] 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 $\vec{\mu} = [5, 2, 3, 1, 4, 6]^T$ μseconds.

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

We use SDF^3 tool-set [27] 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 Comparison

In this experiment, we compare the throughput resulting from our SPS approach to the maximum achievable throughput of a streaming application. The maximum achievable throughput of a streaming application modeled as a CSDF graph is its throughput under STS schedule [26]. We measure the throughput of the actors producing the output streams of the applications (i.e. *sink* actors). Let ρ_i^{SPS} be the throughput of actor $v_i \in V$ with period λ_i under SPS. It follows that:

$$\rho_i^{\mathsf{SPS}} = 1/\lambda_i, \tag{30}$$

where $\lambda_i \in \vec{\lambda}^{\min}$ given by Equation 11. In order to compute the throughput under STS, we use the SDF^3 tool-set. SDF^3 defines ρ_G^{STS} as the graph throughput under STS, and $\rho_i^{\mathsf{STS}} = q_i \rho_G^{\mathsf{STS}}$, where $q_i \in \vec{q}$ (The repetition vector of G).

Computing the throughput of the STS using SDF^3 is done using the `sdf3analysis-(c)sdf` tool with the following parameters: 1) *algorithm used*: `throughput`, 2) *auto-concurrency*: disabled (i.e. each actor has a self back-edge with one initial token, and 3) *channel size*: unbounded.

We define some notations that help in understanding the results. Let $L = \mathsf{lcm}(\vec{q})$ and $H = \max_{v_i \in V}(\mu_i q_i)$. Moreover, let $H = Lp + r$, where $p = H \div L$ ($\div$ is the integer division operator),

and $r = H \bmod L$ (mod is the integer modulus operator). Now, Table 3 shows the results of comparing the throughput of the sink 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 STS schedule throughput to the SPS schedule throughput ($\rho_{\text{snk}}^{\text{STS}}/\rho_{\text{snk}}^{\text{SPS}}$), where snk denotes the sink actor. We clearly see that our SPS delivers the same throughput as STS for 16 out of 19 applications. An SPS schedule that delivers the same throughput as an STS one is called *Rate-Optimal Strictly Periodic Schedule (ROSPS)* [21]. Only three applications (CD2DAT-(S,C) and Satellite) have lower throughput under our SPS. To understand the impact of the results, we introduce the concept of *matched I/O applications* which according to [28] is the class of applications with a small value of L. The authors in [28] reported recently an interesting finding: *Neighboring actors often have matched I/O rates. This reduces the opportunity and impact of advanced scheduling strategies proposed in the literature*. According to [28], the advanced scheduling strategies proposed in the literature (e.g. [26]) are suitable for *mis-matched* I/O rates applications (i.e. with large L such as CD2DAT and Satellite in Table 3). Looking into the results in Table 3, we see that our SPS performs very-well for matched I/O applications.

Table 3: Results of Throughput Comparison. snk denotes the sink actor.

Application	$\dot{q}_{\text{snk}}$	$\rho_{\text{snk}}^{\text{STS}}$	H	L	$\rho_{\text{snk}}^{\text{SPS}}$	$\rho_{\text{snk}}^{\text{STS}}/\rho_{\text{snk}}^{\text{SPS}}$
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	24.5
CD2DAT-C	160	1.361×10^{-1}	1176	23520	1/147	20.0
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	5.0
Receiver	3840	4.76×10^{-2}	80640	3840	1/21	1.0
Pacemaker	64	2.0×10^{-1}	320	320	1/5	1.0

To further quantify the effect of our SPS, ρ_i^{SPS} can be re-written by substituting $H = Lp + r$ in Equation 11 which results in:

$$\rho_i^{\text{SPS}} = \begin{cases} \frac{q_i}{L} & \text{if } L \nmid H \wedge L > H \\ \frac{q_i}{(p+1)L} & \text{if } L \nmid H \wedge L < H \\ \frac{q_i}{H} & \text{if } L \mid H \end{cases} \quad (31)$$

Equation 31 highlights that the throughput under SPS depends solely on the relationship between L and H. If $L \mid H$, then ρ_i^{SPS} is exactly the same as ρ_i^{STS} for SDF graphs and CSDF graphs where all the firings of an actor v_i require the same execution time (in these two cases, $\rho_i^{\text{STS}} = q_i/H$). If $L \nmid H$ and/or the actor execution time differs per firing, then ρ_i^{SPS} is lower than ρ_G^{STS}. 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.

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 Section 4 provides a very fast and accurate answer, thanks to Corollary 2. It 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 providing the maximum achievable throughput. Figure 8 illustrates the ability to easily compute the minimum number of processors required to schedule the benchmarks in Table 2 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 simply M_{OPT} computed using Equation 5. For partitioned algorithms, we choose PEDF algorithm combined with First-First (FF) allocation, denoted PEDF-FF. For PEDF-FF, the minimum number of processors is computed using Eq. 6 (M_{PEDF}) and Eq. 7 (M_{PAR}). For matched I/O applications (i.e. applications where $L \mid H$), it is easy to show (by substituting L in Equation 11) that β defined in Section 3.2.2 is equal to 1. This implies that for matched I/O applications, $M_{\text{PEDF}} = \lceil 2U_\Gamma - 1 \rceil$ which is approximately twice as M_{OPT} for large values of U_Γ. M_{PAR} provides less resource usage compared to M_{PEDF} with the restriction that it is valid only for the specific task set Γ_G for which it is computed. Another task set $\hat{\Gamma}_G$ with the same total utilization and maximum utilization factor as Γ_G may not be schedulable on M_{PAR} due to the partitioning issues. Comparing M_{PAR} to M_{OPT}, we see that PEDF-FF requires in around 44% of the cases an average of 14% more processors than an optimal algorithm due to the bin-packing effects.

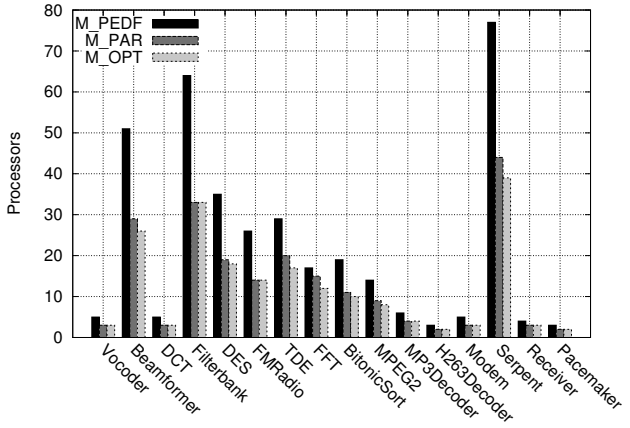

Figure 8: Number of processors required by an optimal algorithm and PEDF-FF

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 M_{STS}, can not be easily computed with equations such as Equations 5, 6, and 7. 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.

6. CONCLUSIONS

We prove that the actors of a streaming application, modeled as an acyclic CSDF graph, can be scheduled as implicit-deadline 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. Based on empirical evaluations, we demonstrate that our framework for strictly periodic scheduling is very suitable for matched I/O rates applications since it provides the maximum achievable throughput of the applications together with the ability to analytically determine the minimum number of processors needed to schedule the applications.

7. ACKNOWLEDGMENT

This work has been supported by CATRENE/MEDEA+ 2A718 project (TSAR: Tera-scale multi-core processor architecture). We would like to thank William Thies and Sander Stuijk for their support with StreamIt and SDF^3 benchmarks respectively. We also would like to thank Taleb Alkurdi and Onno van Gaans from Leiden Mathematics Institute for their valuable comments. Finally, we thank our colleagues at Leiden Embedded Research Center for reviewing early drafts of this manuscript.

8. REFERENCES

[1] J. H. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. In *Proc. of ECRTS*, pages 76–85, 2001.

[2] B. Andersson and E. Tovar. Multiprocessor Scheduling with Few Preemptions. In *Proc. of RTCSA*, pages 322–334, 2006.

[3] M. Bekooij, R. Hoes, O. Moreira, P. Poplavko, M. Pastrnak, B. Mesman, J. Mol, S. Stuijk, V. Gheorghita, and J. Meerbergen. Dataflow Analysis for Real-Time Embedded Multiprocessor System Design. In *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, volume 3, pages 81–108. Springer Netherlands, 2005.

[4] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-Static Dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, 1996.

[5] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms. In J. Y. T. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, 2004.

[6] H. Cho, B. Ravindran, and E. D. Jensen. T-L plane-based real-time scheduling for homogeneous multiprocessors. *Journal of Parallel and Distributed Computing*, 70(3):225–236, 2010.

[7] R. I. Davis and A. Burns. A Survey of Hard Real-Time Scheduling for Multiprocessor Systems. Accepted for publication in ACM Computing Surveys. Pre-print available at: `http://www-users.cs.york.ac.uk/~robdavis/papers/MPSurveyv5.0.pdf`.

[8] M. L. Dertouzos. Control Robotics: The Procedural Control of Physical Processes. In *Proc. of IFIP Congress*, pages 807–813, 1974.

[9] A. Gerstlauer, C. Haubelt, A. D. Pimentel, T. P. Stefanov, D. D. Gajski, and J. Teich. Electronic System-Level Synthesis Methodologies. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1517–1530, 2009.

[10] S. Goddard. *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, 1998.

[11] K. Jeffay, D. Stanat, and C. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proc. of RTSS*, pages 129–139, 1991.

[12] L. Karam, I. AlKamal, A. Gatherer, G. Frantz, D. Anderson, and B. Evans. Trends in multicore DSP platforms. *IEEE Signal Processing Magazine*, 26(6):38–49, 2009.

[13] E. A. Lee and S. Ha. Scheduling strategies for multiprocessor real-time DSP. In *Proc. of GLOBECOM*, volume 2, pages 1279–1283, 1989.

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

[15] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt. DP-FAIR: A Simple Model for Understanding Optimal Multiprocessor Scheduling. In *Proc. of ECRTS*, pages 3–13, 2010.

[16] J. M. López, J. L. Díaz, and D. F. García. Utilization Bounds for EDF Scheduling on Real-Time Multiprocessor Systems. *Real-Time Systems*, 28:39–68, 2004.

[17] G. Martin. Overview of the MPSoC design challenge. In *Proc. of DAC*, pages 274–279, 2006.

[18] A. Moonen, M. Bekooij, R. van den Berg, and J. van Meerbergen. Cache aware mapping of streaming applications on a multiprocessor system-on-chip. In *Proc. of DATE*, pages 300–305, 2008.

[19] O. Moreira, J.-D. Mol, M. Bekooij, and J. van Meerbergen. Multiprocessor Resource Allocation for Hard-Real-Time Streaming with a Dynamic Job-Mix. In *Proc. of RTAS*, pages 332–341, 2005.

[20] O. Moreira, F. Valente, and M. Bekooij. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *Proc. of EMSOFT*, pages 57–66, 2007.

[21] O. M. Moreira and M. J. G. Bekooij. Self-Timed Scheduling Analysis for Real-Time Applications. *EURASIP Journal on Advances in Signal Processing*, 2007:1–15, 2007.

[22] V. Nollet, D. Verkest, and H. Corporaal. A Safari Through the MPSoC Run-Time Management Jungle. *Journal of Signal Processing Systems*, 60:251–268, 2010.

[23] H. Oh and S. Ha. Fractional Rate Dataflow Model for Efficient Code Synthesis. *The Journal of VLSI Signal Processing*, 37:41–51, 2004.

[24] T. Parks and E. Lee. Non-preemptive real-time scheduling of dataflow systems. In *Proc. of ICASSP*, volume 5, pages 3235–3238, 1995.

[25] R. Pellizzoni, P. Meredith, M.-Y. Nam, M. Sun, M. Caccamo, and L. Sha. Handling mixed-criticality in SoC-based real-time embedded systems. In *Proc. of EMSOFT*, pages 235–244, 2009.

[26] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, 2nd edition, 2009.

[27] S. Stuijk, M. Geilen, and T. Basten. SDF^3: SDF For Free. In *Proc. of ACSD*, pages 276–278, 2006.

[28] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proc. of PACT*, pages 365–376, 2010.