Improved Hard Real-Time Scheduling of CSDF-modeled Streaming Applications

Jelena Spasic Di Liu Emanuele Cannella Todor Stefanov
Leiden Institute of Advanced Computer Science
Leiden University, Leiden, The Netherlands
Email: {j.spasic, d.liu, t.p.stefanov}@liacs.leidenuniv.nl, emanuele.cannella@gmail.com

ABSTRACT

Recently, it has been shown that hard real-time scheduling theory can be applied to streaming applications modeled as acyclic Cyclo-Static Dataflow (CSDF) graphs. However, that approach is not efficient in terms of throughput and processor utilization. Therefore, in this paper, we propose an improved hard real-time scheduling approach to schedule streaming applications modeled as acyclic CSDF graphs on a Multi-Processor System-on-Chip (MPSoC) platform. The proposed approach converts each actor in a CSDF graph to a set of real-time periodic tasks. The conversion enables application of many hard real-time scheduling algorithms which offer fast calculation of the required number of processors for scheduling the tasks. We evaluate the performance and time complexity of our approach in comparison to several existing scheduling approaches. Experiments on a set of real-life streaming applications demonstrate that our approach: 1) results in systems with higher throughput and better processor utilization in comparison to the existing hard real-time scheduling approach for CSDF graphs while requiring comparable time for the system derivation; 2) gives the same throughput as the existing periodic scheduling approach for CSDF graphs but requires much shorter time to derive the task schedule and tasks' parameters (periods, start times, etc.); and 3) gives the throughput that is equal or very close to the maximum achievable throughput of an application obtained via self-timed scheduling, but requires much shorter time to derive the schedule. The total time needed for the proposed conversion approach and the calculation of the minimum number of processors needed to schedule the tasks and the calculation of the size of communication buffers between tasks is in the range of seconds.

1. INTRODUCTION

Modern streaming applications have high computational demands and hard real-time requirements. As huge amount of data should be processed in a "short" time interval, the parallel processing comes as a natural solution. The processing power of Multi-Processor System-on-Chip (MPSoC) platforms perfectly matches the computational requirements of streaming applications. Designing such an embedded system imposes several challenges: a streaming application should be represented in a way which reveals the parallelism of the application and it should be mapped and scheduled on a platform such that hard real-time requirements are satisfied.

To address these challenges, several parallel Models-of-Computation (MoCs), e.g. Synchronous Data Flow (SDF) [12] and Cyclo-Static Dataflow (CSDF) [5], have been adopted as the parallel application specification. Within a MoC, an application is represented as a set of concurrently executing and communicating tasks. Thus, the parallelism is explicitly specified in the model. Two primary performance metrics of streaming applications are throughput and latency. Throughput is defined by the number of samples an application can produce during a given time interval, while latency is the elapsed time between the arrival of a sample to an application and the output of the processed sample by the application. Apart from guaranteeing a certain throughput and latency for each application running on a platform, modern embedded systems should be able to accept or stop applications at run-time without violating the timing requirements of the other running applications. This property is called *temporal isolation* between the applications. Many algorithms from the classical hard real-time multiprocessor scheduling theory can perform *fast* admission and scheduling decisions for the incoming applications while providing hard real-time guarantees and temporal isolation between the applications. Moreover, these algorithms enable several efficient and fast approaches to compute the number of processors required to schedule the applications instead of performing a complex design space exploration. Such an approach for computing the number of processors for scheduling the applications is given in Section 5.6 and it is, in the worst case, of a polynomial time complexity.

Recently, the authors in [2] proposed a framework to schedule streaming applications modeled as acyclic CSDF graphs as a set of real-time periodic tasks on an MPSoC platform. They also derive the minimum number of processors needed to schedule the applications on a platform. However, in that framework, the authors use one and the same worst-case execution time (WCET) value for all execution phases of a task in the CSDF graph, although a task in the CSDF graph may have a different WCET value for every phase. The authors simply take and use the maximum WCET value among the WCET values for all phases of a task. By doing this, the cyclically changing execution nature of an application modeled by the CSDF model is hidden, which leads to underestimation of the throughput, overestimation of the latency, and underutilization of processors. In another recent work [6], the authors proposed a framework to evaluate a lower bound of the maximum throughput of a periodically scheduled CSDF-modeled application. However, the authors do not provide a method to determine the number of processors required for scheduling the application. Moreover, their approach does not ensure temporal isolation among applications, i.e., the schedule of applications has to be recalculated once a new application comes in the system and hence it may be possible that the previously calculated throughput of an application can no longer be reached.

In this paper, we address the drawbacks of [2] and [6] by considering different WCET values for task's phases in an acyclic CSDF graph and enabling temporal isolation of applications while providing hard real-time guarantees. The contributions of this paper are the following:

- We prove that considering a different WCET value for each execution phase of a task we can convert the execution phases of each task in an acyclic CSDF graph to strictly periodic real-time tasks. This enables the use of many hard real-time scheduling algorithms to schedule such tasks with a certain guaranteed throughput and latency. (Theorem 3)
- We prove that our scheduling approach gives equal or higher throughput than the existing hard real-time scheduling approach for acyclic CSDF graphs. (Theorem 4)
- We show, on a set of real-life streaming applications, that scheduling each execution phase of a CSDF task as a strictly periodic task and considering different WCET per phase lead not only to tighter guarantee on the throughput of an application but also to better utilization of processor resources.
- We demonstrate, on a set of real-life streaming applications, that the total time required by our approach to derive the

schedule of the tasks, calculate the minimum number of processors needed to schedule the tasks and calculate the size of communication buffers between tasks is comparable to the time required by the existing hard real-time scheduling approach for CSDF graphs. In addition, we show that the total time needed by our approach is much shorter in comparison to the existing periodic scheduling and self-timed scheduling approaches for CSDF graphs.

The remainder of the paper is organized as follows: Section 2 gives an overview of the related work. Section 3 introduces the background necessary to understand the proposed scheduling method. Section 4 gives a motivational example. The proposed scheduling method is described in Section 5. Experimental evaluation of the approach is given in Section 6, and Section 7 concludes the paper.

2. RELATED WORK

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

[9] proposes a two parameter (σ, ρ) workload characterization to reduce the difference between the worst-case throughput, determined by the analysis, and the actual throughput of the application. They consider different execution times for task's phases and then the average worst-case execution time is used to improve the minimum guaranteed throughput/latency. Similar to them, we consider different execution times for task's phases in a CSDF graph. But in contrast to them, we convert task's phases to classical periodic hard real-time tasks, which allows us to calculate the minimum number of processors required to guarantee certain throughput and latency in a fast and analytical way for global scheduling and in a polynomial time for partitioned scheduling by using our algorithm given in Section 5.6.

In [7], the authors propose an analysis framework for hard real-time applications modeled as Affine Dataflow Graphs (ADF). The actors in an ADF graph are scheduled as periodic tasks. The ADF model proposed in [7] extends the CSDF model and hence, is more expressive than the CSDF. However, in their approach only one value is considered as the WCET value of a task, while we consider a different WCET value per each phase of a task, thereby efficiently exploiting the cyclic nature of the CSDF model and providing a tighter throughput guarantee.

[6] proposes a framework to derive the maximum throughput of a CSDF graph under a periodic schedule and to calculate the buffer sizes in the graph with a throughput constraint. Both problems are represented as LP problems and solved approximately. Similar to our work, their work considers different execution times for each phase of a task. However, it is not explicitly given in [6] how to compute the number of processors needed to schedule the graph according to the derived schedule. One possible way is to look at the derived schedules and find the maximum number of active tasks at any given point in time. However, this procedure has an exponential time complexity in the worst case. In contrast, in our case the conversion of CSDF task's phases to classical periodic hard real-time tasks enables fast and analytical calculation of the minimum number of processors for global scheduling of the tasks, and a polynomial time derivation of the number of processors for partitioned scheduling by using our algorithm given in Section 5.6.

The closest to our work, in terms of scope of work and methods proposed to schedule streaming applications modeled as acyclic CSDF graphs, is the work in [2]. The authors in [2] convert each task in a CSDF graph to a periodic task by deriving parameters such as period and start time. Then they use hard real-time schedulability analysis to determine the minimum number of processors required to execute the derived task-set. Our approach differs from [2] in the following: we use different WCET values for each execution phase of a task and each phase is converted to a periodic task, while in [2], only one WCET value is used for a task and every execution of a task is periodic with a calculated period. By considering different WCET values for each task phase and converting each phase to a periodic task, we can guarantee tighter throughput and better utilization of processor resources.

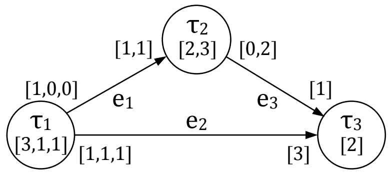

Figure 1: A CSDF graph *G*.

3. BACKGROUND

In this section, we first introduce the application model, i.e., the CSDF MoC, followed by the system model we use. After that we review the scheduling framework proposed in [2], which we use as a main reference point for comparison with our approach presented in Section 5.

3.1 Cyclo-Static Dataflow (CSDF)

An application modeled as a CSDF [5] is a directed graph $G = (V, E)$ that consists of a set of actors V which communicate with each other through a set of communication channels E. Actors represent a certain functionality of the application, while communication channels are FIFOs representing data dependency and transferring data tokens between the actors. Every actor $\tau_i \in V$ has an *execution sequence* $[f_i(1), f_i(2), \cdots, f_i(P_i)]$ of length P_i, i.e., it has P_i phases. The kth time that actor τ_i is fired, it executes the function $f_i(((k-1) \bmod P_i)+1)$. As a consequence, the execution time of actor τ_i is also a sequence $[C_i^C(1), C_i^C(2), \cdots, C_i^C(P_i)]$ consisting of the worst-case computation time values for each phase. Similarly, every output channel e_u of an actor τ_i has a predefined token *production sequence* $[x_i^u(1), x_i^u(2), \cdots, x_i^u(P_i)]$ of length P_i. Analogously, token consumption on every input channel e_u of an actor τ_i is a predefined sequence $[y_i^u(1), y_i^u(2), \cdots, y_i^u(P_i)]$, called *consumption sequence*. The total number of tokens on a channel e_u produced by τ_i during its first n invocations and the total number of tokens consumed on the same channel by τ_j during its first n invocations are $X_i^u(n) = \sum_{l=1}^{n} x_i^u(((l-1) \bmod P_i)+1)$ and $Y_j^u(n) = \sum_{l=1}^{n} y_j^u(((l-1) \bmod P_j)+1)$, respectively.

Figure 1 shows an example of a CSDF graph. For instance, actor τ_1 has 3 phases, its execution time sequence (in time units) is $[C_1^C(1), C_1^C(2), C_1^C(3)] = [3, 1, 1]$ and its token production sequence on channel e_1 is $[1, 0, 0]$.

An important property of the CSDF model is the ability to derive at design-time a schedule for the actors. In order to derive a valid static schedule for a CSDF graph at design-time, it has to be consistent and live.

THEOREM 1 (FROM [5]). *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*

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

A CSDF graph G is said to be consistent if a positive integer solution $\vec{r} = [r_1, r_2, \cdots, r_N]^T$ exists for the balance equation, Eq. (2). We call $\vec{r}$ *aggregated repetition vector*. If a deadlock-free schedule can be found, G is said to be live. An entry $q_i \in \vec{q}$ represents the number of invocations of an actor τ_i in a graph iteration of G. Similarly, an entry $r_i \in \vec{r}$ represents the number of invocations of a phase of an actor τ_i in a graph iteration of G. For graph G shown in Figure 1, the repetition vector $\vec{q}$ is $[6, 2, 2]^T$ and the aggregated repetition vector $\vec{r}$ is $[2, 1, 2]^T$. Throughout this paper, all CSDF graphs are assumed to be consistent and live.

3.2 System Model

In this work, we consider a system composed of a set $\Pi = \{\pi_1, \pi_2, \cdots, \pi_m\}$ of m identical processors. The processors execute a task-set $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_n\}$ of n periodic tasks, which can be preempted at any time. A periodic task $\tau_i \in \mathcal{T}$ is defined by a 4-tuple $\tau_i = (S_i, C_i, D_i, T_i)$, where S_i is the start time of τ_i in absolute time units, C_i is the WCET, D_i is the deadline of τ_i in relative time units, and T_i is the task period (where $T_i \geq C_i$) in relative time units. In this work, we only consider tasks which relative deadline D_i is equal to its period T_i.

The utilization of task τ_i, denoted as u_i, where $u_i \in (0, 1]$, is defined as $u_i = C_i/T_i$. For a task-set $\mathcal{T}$, $u_{\mathcal{T}}$ is the total utilization of $\mathcal{T}$ given by $u_{\mathcal{T}} = \sum_{\tau_i \in \mathcal{T}} u_i$. The total utilization of a task-set directly determines the minimum number of processors needed to schedule the task-set. Given a system Π and a task-set $\mathcal{T}$, a necessary and sufficient condition for $\mathcal{T}$ to be scheduled on Π such that all deadlines are met is given by $u_{\mathcal{T}} \leq m$. Thus, the absolute minimum number of processors needed to schedule a periodic task-set with deadlines equal to periods is given by [3]:

$$m_{\text{OPT}} = \lceil u_{\mathcal{T}} \rceil. \tag{4}$$

Scheduling $\mathcal{T}$ on m_{OPT} processors is possible only by using the optimal scheduling algorithms, which are either global or hybrid. However, global and hybrid scheduling algorithms require task migration. The other class of scheduling algorithms are partitioned algorithms which do not require task migration. With partitioned scheduling, tasks are first allocated to processors. Then, the tasks on each processor are scheduled using a uniprocessor scheduling algorithm. The minimum number of processors needed to schedule a task-set $\mathcal{T}$ assuming partitioned scheduling is given by:

$$m_{\text{PAR}} = \min_{x \in \mathbb{N}}\{x\text{-part. of } \mathcal{T} \wedge \forall i \in [1, x] : \mathcal{T}_i \text{ is sched. on } \pi_i\}. \tag{5}$$

Note that m_{OPT} is the lower bound on the number of processors m_{PAR} needed by partitioned scheduling algorithms.

3.3 Strictly Periodic Scheduling of CSDF

In [2], a real-time strictly periodic scheduling (SPS) framework for acyclic CSDF graphs is proposed. In this framework, every actor τ_i in a CSDF graph G is converted to an implicit-deadline periodic (IDP) task by computing the task parameters S_i, D_i, T_i and C_i, where C_i is computed as the maximum WCET value of actor τ_i, i.e., $C_i = \max_{1 \leq \varphi \leq P_i}\{C_i(\varphi)\}$, where $C_i(\varphi)$ contains the worst-case computation, read and write time of a phase φ of actor τ_i. To execute graph G strictly periodically, period T_i for each actor τ_i is computed as:

$$T_i = \frac{\text{lcm}(\vec{q})}{q_i} \left\lceil \frac{\max_{\tau_j \in V}\{C_j q_j\}}{\text{lcm}(\vec{q})} \right\rceil, \forall \tau_i \in V, \tag{6}$$

where $\text{lcm}(\vec{q})$ is the least common multiple of all repetition entries in $\vec{q}$. Once the actor periods are computed, the throughput of each actor τ_i can be computed as $1/T_i$, while the throughput of a graph G is equal to $1/(q_i T_i)$. The authors also provide in [2] a method for calculating the latency of a CSDF graph scheduled in a strictly periodic fashion. In addition, the framework computes the minimum buffer size for each channel in a graph such that actors, i.e., tasks, can be executed in strictly periodic fashion. Converting the actors to periodic tasks enables fast analytical calculation of the minimum number of processors needed to schedule the application. The strictly periodic schedule of all actors in G, given in Figure 1, is shown in Figure 2(a), under the assumption that read and write times are 0 (for the sake of simplicity). For example, actor τ_2 executes periodically with the calculated period $T_2 = 9$. Note that for every actor's phase one and the same WCET value is considered, i.e., for actor τ_2 we have two phases 1 and 2 and the considered WCET value C_2 for each phase is $C_2 = \max\{C_2(1), C_2(2)\} = \max\{C_2^C(1), C_2^C(2)\} = \max\{2, 3\} = 3$.

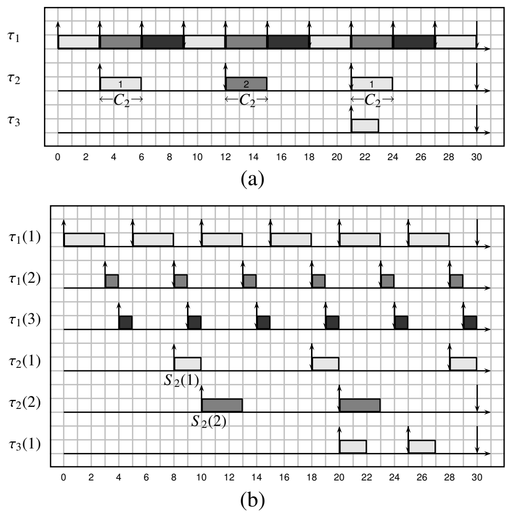

Figure 2: (a) The SPS and (b) ISPS of the CSDF graph G in Figure 1.

Table 1: Throughput, latency and number of processors for G under different scheduling schemes.

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

4. MOTIVATIONAL EXAMPLE

The goal of this section is to show that the SPS approach [2] introduced in Section 3.3 is not efficient in terms of throughput, latency and utilization of processor resources. We analyze two different schedules of the CSDF graph G in Figure 1 to demonstrate the need of considering different WCET values of actor's phases and the drawback of strictly periodic schedule between actor phases. The first schedule we consider is SPS. This schedule is visualized in Figure 2(a). Each execution of an actor is periodic with the period computed by Eq. (6). Moreover, every execution phase of an actor is assumed to have one and the same WCET value. The throughput $\mathcal{R}$, latency $\mathcal{L}$ of G and the required number of processors m are given in Table 1 under SPS.

However, by taking one and the same value as the WCET for all execution phases of an actor, the cyclic behavior of the CSDF actors is hidden. Assume that we convert each actor τ_i in G to a set of P_i tasks considering different WCET values for each execution phase and execute them as periodic tasks. The execution schedule of such task-set is given in Figure 2(b). Again, here we assume that read and write times are 0. For example, actor τ_2 is converted to 2 periodic tasks $\tau_2(1)$ and $\tau_2(2)$ where each task is executed periodically with a period equal to 10. Moreover, the WCET values of the tasks $\tau_2(1)$ and $\tau_2(2)$ are not the same but $\tau_2(1)$ has WCET $C_2(1) = 2$ and $\tau_2(2)$ has WCET $C_2(2) = 3$, as the original specification in Figure 1.

We can see from Table 1 under ISPS that by scheduling G in such a way we can obtain almost 2 times higher graph throughput and shorter graph latency while resources in terms of the required number of processors are the same compared with SPS and thus, the processor resources are better utilized in the case of ISPS. This is especially important in case of a timing constraint because it may happen that graph cannot meet the constraint when scheduled under SPS. Here the throughput and latency under ISPS are calculated by using our approach described in Section 5. The required number of processors for both SPS and ISPS is calculated by Eq. (4). Moreover, the number of processors needed for partitioned scheduling in both cases is the same as the number needed for global scheduling

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

```
Input: A CSDF graph $G = (V, E)$.
Output: For each actor $\tau_i \in V$, a set of periodic tasks $\mathcal{T}_{\tau_i} = \{\tau_i(1), \cdots, \tau_i(P_i)\}$, and for each channel $e_u \in E$, the size of the buffer $b_u$.
1  for actor $\tau_i \in V$ do
2      Compute the minimum common period $\check{T}_i$ by using Eq. (10);
3  for actor $\tau_i \in V$ do
4      for phase $\varphi$ of $\tau_i$, $1 \leq \varphi \leq P_i$ do
5          $\tau_i(\varphi) = (0, C_i(\varphi), \check{T}_i, \check{T}_i)$
6  for actor $\tau_i \in V$ do
7      Compute the start time of the first phase $S_i(1)$ by using Eq. (13);
8      for phase $\varphi$ of $\tau_i$, $2 \leq \varphi \leq P_i$ do
9          Compute the start time of the $\varphi$th phase $S_i(\varphi)$ by using Eq. (15);
10 for actor $\tau_i \in V$ do
11     for phase $\varphi$ of $\tau_i$, $1 \leq \varphi \leq P_i$ do
12         $\tau_i(\varphi) = (S_i(\varphi), C_i(\varphi), \check{T}_i, \check{T}_i)$
13 for communication channel $e_u \in E$ do
14     Compute the buffer size $b_u$ by using Eq. (18);
```

given by Eq. (4). We can see from the motivational example that the approach from [2] yields to lower throughput and larger latency of a graph by using one and the same value for the WCET of each phase of an actor and by strictly periodic scheduling of all executions of the actor. Thus, different WCET values for actor phases should be considered and the constraint on strictly periodic scheduling between the actor phases should be removed.

5. IMPROVED HARD REAL-TIME SCHEDULING OF CSDF

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

The proposed conversion procedure is given in Algorithm 1. First, the periods of tasks corresponding to actors are calculated in lines 1-2, explained in Section 5.1, and relative deadlines of the tasks are set to be equal to the corresponding task periods, lines 3-5. Then the start times for each task-set corresponding to an actor are computed in lines 6-12, for details see Section 5.2. Finally, the buffer sizes of the communication channels are derived in lines 13-14, for details see Section 5.3.

5.1 Deriving Periods of Tasks

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

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

$$C_i(\varphi) = C^R \cdot \sum_{e_r \in \text{in}(\tau_i)} y_i^r(\varphi) + C_i^C(\varphi) + C^W \cdot \sum_{e_w \in \text{out}(\tau_i)} x_i^w(\varphi), \quad (7)$$

where C^R represents the platform-dependent worst-case time needed to read a single token from an input channel e_r from the set of input channels $\text{in}(\tau_i)$ of actor τ_i; analogously, C^W is the worst-case time needed to write a single token to an output channel e_w from the set of output channels $\text{out}(\tau_i)$ of τ_i; $y_i^r(\varphi)$ and $x_i^w(\varphi)$ is the number of tokens read from e_r and written to e_w by τ_i, respectively, during its execution phase φ; and $C_i^C(\varphi)$ is the worst-case computation time of τ_i in its phase φ.

Definition 2. For an acyclic CSDF graph G, an **aggregated execution vector** $\vec{AC}$, where $\vec{AC} \in \mathbb{N}^N$, represents the aggregated WCET value of the actors in G and its elements are given by $AC_i = \sum_{\varphi=1}^{P_i} C_i(\varphi)$, where $C_i(\varphi)$ is the WCET value of τ_i's phase φ.

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

Definition 3. A task $\tau_i(\varphi)$ corresponding to a phase φ of an actor τ_i, where $1 \leq \varphi \leq P_i$, in an acyclic CSDF graph G is a **strictly periodic task** iff the time period between any two consecutive firings of that task is constant.

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

Definition 4. For an acyclic CSDF graph G, a **common period vector** $\vec{T}$, where $\vec{T} \in \mathbb{N}^N$, represents the periods, measured in time units, of periodic task-sets corresponding to actors in G. $T_i \in \vec{T}$ is common period of periodic task-set corresponding to actor $\tau_i \in V$. $\vec{T}$ is given by the solution to both

$$r_1 T_1 = r_2 T_2 = \cdots = r_{N-1} T_{N-1} = r_N T_N \quad (8)$$

and

$$\vec{T} - \vec{AC} \geq \vec{0}, \quad (9)$$

where $r_i \in \vec{r}$ and $\vec{r}$ is the aggregated repetition vector introduced in Section 3.1.

Lemma 1. *For an acyclic CSDF graph G, the minimum common period vector $\check{\vec{T}}$ is given by:*

$$\check{T}_i = \frac{\text{lcm}(\vec{r})}{r_i} \left\lceil \frac{\max_{\tau_j \in V}\{AC_j r_j\}}{\text{lcm}(\vec{r})} \right\rceil, \forall \tau_i \in V, \quad (10)$$

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

Proof. The minimum common period vector $\check{\vec{T}}$ that solves Eq. (8) is given by:

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

Ineq. (9) can be re-written as:

$$c\check{T}_1 \geq AC_1, c\check{T}_2 \geq AC_2, \cdots, c\check{T}_N \geq AC_N, c \in \mathbb{N}. \quad (11)$$

Further, Ineq. (11) can be re-written as:

$$c \geq AC_1 r_1 / \text{lcm}(\vec{r}), \cdots, c \geq AC_N r_N / \text{lcm}(\vec{r}). \quad (12)$$

From Ineq. (12), it follows that c is greater than or equal to $\max_{\tau_j \in V}\{AC_j r_j\}/\text{lcm}(\vec{r})$. However, $\max_{\tau_j \in V}\{AC_j r_j\}/\text{lcm}(\vec{r})$ is not always guaranteed to be an integer. Because of that, the value is rounded up by taking its ceiling. Thus, the minimum common period vector which satisfies both Eq. (8) and Ineq. (9) is given by Eq. (10). □

Once we derive the periods of actors in a graph using Eq. (10), actor deadlines are derived implicitly given that we convert each actor to a set of strictly periodic tasks with deadlines equal to the periods.

For the CSDF graph in Figure 1, the derived minimum common period vector is $\check{\vec{T}} = [5, 10, 5]$ time units.

Theorem 2. *For any acyclic CSDF graph G, a periodic schedule exists such that every phase of an actor $\tau_i \in V$ is strictly periodic with a constant period $T_i \in \check{\vec{T}}$ and every communication channel $e_u \in E$ has a bounded buffer capacity.*

Proof. Let us assume that graph G is partitioned into L levels in a way similar to topological sort. In that way, all input actors belong to level-1, the actors from level-2 have all immediate predecessors in level-1, the actors from level-3 have immediate predecessors in level-2 and can also have immediate predecessors in level-1, and so on. The graph iteration period is $\alpha = r_1 T_1 = \cdots = r_N T_N$. During the iteration period each phase of τ_i is executed r_i times. Assume that the first phase of level-1 actors starts at time $t = 0$. Other phases of an

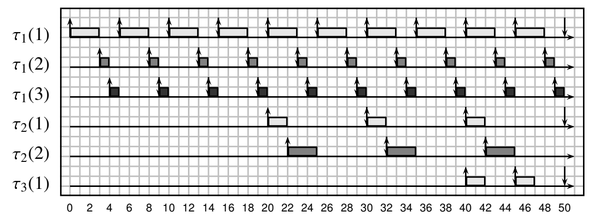

Figure 3: The periodic schedule σ for the CSDF graph G shown in Figure 1.

actor are scheduled to be fired as soon as the WCET of the previous phase elapses. Recall that every actor τ_i in graph G is converted to a set of strictly periodic tasks where a task corresponds to a phase of the actor. Consider now an actor from level-1, denoted as τ_1. By time $t = \alpha + S_1(\mathrm{P}_1)$, the last phase of τ_1 will finish its r_1th execution, where $S_1(\mathrm{P}_1)$ is the start time of the last phase of τ_1. Level-1 actors will complete a whole iteration by time $t_1 = \alpha + \max_{\tau_i \in \text{level-1}}\{S_i(\mathrm{P}_i)\}$ and will continue executing their second iteration. According to Eq. (2), level-1 actors will produce enough data on all channels to level-2 actors by time t_1 such that level-2 actors can execute a whole iteration if their first phases are started at t_1, at the earliest. Let us start the first phases of level-2 actors at time $t = 2\alpha$ and all the other phases of a level-2 actor one after the other. Similarly, by time $t_2 = 3\alpha + \max_{\tau_i \in \text{level-2}}\{S_i(\mathrm{P}_i) - S_i(1)\}$, level-3 actors will have enough data to execute one iteration. Thus, starting the first phases of level-3 actors at time $t = 4\alpha$ guarantees that the actors can execute a whole iteration. By repeating the same procedure to the actors of the last level, level L, (by starting their first phases at $t = (L-1) \cdot 2\alpha$ and all the other phases as soon as the WCET of previous phase elapses), we obtain an overlapping schedule σ where all actors execute their corresponding iterations. In the constructed schedule, the first phase of an actor τ_j corresponding to a level-i will start execution at time $t = (i-1) \cdot 2\alpha$ and once it starts it will be fired every T_j time units. The other phases start their executions one after the other and all within period T_j. Once started, each phase is re-executed every T_j time units.

Now, we will prove that the constructed schedule executes with bounded buffers. The longest delay which may happen between production and consumption of data tokens is in case when there is a dependency e_u between the first iteration of a level-1 actor and the first iteration of a level-L actor. In this case the delay is equal to $(L-1) \cdot 2\alpha$ and during that period the level-1 actor will produce on channel e_u at most $(L-1) \cdot 2X_1^u(P_1 r_1)$ data tokens, where $X_1^u(P_1 r_1)$ is the number of tokens produced during $P_1 r_1$ executions of the level-1 actor. However, starting from $L \cdot 2\alpha$ level-1 and level-L execute in parallel, so we should increase the buffer size by $2X_1^u(P_1 r_1)$ which then becomes $L \cdot 2X_1^u(P_1 r_1)$. We can now use the methodology described above to determine the buffer size of each communication channel in a graph: each channel $e_u \in E$, connecting a level-i source actor τ_k and a level-j destination actor ($j \geq i$) will store according to schedule σ at most:

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

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

For the example graph G given in Figure 1, actors in G are grouped into 3 levels such that τ_1 is level-1 actor, τ_2 level-2 and τ_3 is level-3 actor. The calculated graph iteration period α is equal to 10. The periodic schedule resulting from Theorem 2, namely schedule σ, is depicted in Figure 3.

5.2 Deriving The Earliest Actor Phase Start Times

In order to represent an actor of a CSDF graph as a set of strictly periodic tasks, we still need one parameter to be derived – the start time of each task. In Section 5.1 we already introduced the start times of phases of the actors corresponding to different levels, but those start times are not minimum. Minimizing the start times is very important since it has a direct impact on the latency of the graph and the buffer sizes of the communication channels. Therefore, the earliest start times are derived below.

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

Lemma 2. *For an acyclic CSDF graph G, the earliest start time of the first phase of an actor $\tau_j \in V$, denoted S_j, under* ISPS *is given by:*

$$S_j(1) = \begin{cases} 0 & \text{if } prec(\tau_j) = \emptyset \\ \max_{\tau_i \in prec(\tau_j)}\{S_{i\to j}(1)\} & \text{if } prec(\tau_j) \neq \emptyset \end{cases} \quad (13)$$

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

$$S_{i\to j}(1) = \min_{t \in [0, S_i(1)+\alpha+\Delta_i(P_i)]}\{t : \underset{[S_i(1),\max\{S_i(1),t\}+k)}{\mathsf{prd}}(\tau_i) \geq \underset{[t,\max\{S_i(1),t\}+k]}{\mathsf{cns}}(\tau_j),\ \forall k \in [0, \alpha + \Delta_i(P_i)]\}, \quad (14)$$

where $S_i(1)$ is the earliest start time of the first phase of a predecessor actor τ_i, $\alpha = r_i T_i = r_j T_j$, $\Delta_i(P_i) = S_i(P_i) - S_i(1)$, $\mathsf{prd}_{[t_s,t_e)}(\tau_i)$ is the number of tokens produced by τ_i during the time interval $[t_s, t_e)$, and $\mathsf{cns}_{[t_s,t_e]}(\tau_j)$ is the number of tokens consumed by τ_j during the time interval $[t_s, t_e]$. The earliest start times of the other phases of $\tau_j \in V$ are then given by:

$$S_j(\varphi) = S_j(\varphi - 1) + C_j(\varphi - 1),\ \forall \varphi \in [2, \cdots, P_j]. \quad (15)$$

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

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

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

$$S_j(1) = \max_{\tau_i \in prec(\tau_j)}\{S_i(1) + \alpha + \Delta_i(\mathrm{P}_i)\}.$$

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

$$S_j(1) = \max_{\tau_i \in prec(\tau_j)}\{S_{i\to j}(1)\}$$
$$\text{where } S_{i\to j}(1) = t',\ t' \in [0, S_i(1) + \alpha + \Delta_i(\mathrm{P}_i)].$$

A valid start time candidate $S_{i\to j}(1)$ must guarantee that the number of tokens available on channel $e_u = (\tau_i, \tau_j)$ at any time instant $t \geq t'$ is greater than or equal to the number of consumed tokens at the same instant such that τ_j can be executed as a set of strictly periodic tasks. Here, we have two cases: ***Case 1***: $\underline{t' \geq S_i(1)}$: In order to guarantee that τ_j can fire its first phase at times $t = t', t' + T_j, \cdots, t' + \alpha$ and each its other phase φ as early as possible at times $t = t' + \Delta_j(\varphi), t' + \Delta_j(\varphi) + T_j, \cdots, t' + \Delta_j(\varphi) + \alpha - T_j$, where $\Delta_j(\varphi) = \sum_{l=1}^{\varphi-1} C_j(l)$, t' must satisfy:

$$\forall k \in [0, \alpha + \Delta_i(\mathrm{P}_i)] : \underset{[S_i(1),t'+k)}{\mathsf{prd}}(\tau_i) \geq \underset{[t',t'+k]}{\mathsf{cns}}(\tau_j). \quad (16)$$

Thus, a valid value of t' guarantees that once τ_j starts, it always finds enough data to fire for one iteration. ***Case 2***: $\underline{t' < S_i(1)}$: This case happens when τ_j consumes zero tokens in the interval $[S_i(1), t']$ or there are initial tokens on the channel. It is sufficient to check the cumulative production and consumption over the interval $[S_i(1), S_i(1) + \alpha + \Delta_i(\mathrm{P}_i)]$ since by time $t = S_i(1) + \alpha + \Delta_i(\mathrm{P}_i)$ both τ_i and τ_j are guaranteed to have finished one iteration:

$$\forall k \in [0, \alpha + \Delta_i(\mathrm{P}_i)] : \underset{[S_i(1),S_i(1)+k)}{\mathsf{prd}}(\tau_i) \geq \underset{[t',S_i(1)+k]}{\mathsf{cns}}(\tau_j). \quad (17)$$

By merging Eq. (16) and Eq. (17) and then selecting among valid start times t' the minimum one, we obtain Eq. (14). Start times for the tasks corresponding to the actor phases other than the first phase are obtained by adding the WCET value of the previous phase to the derived start time of the previous phase, which is given by Eq. (15). The start times derived in such a way enable the serialized execution of tasks corresponding to actor phases, when it is needed, by careful allocation and certain scheduling algorithms, which will be explained in more detail in Section 5.4. □

For example, the derived earliest start times for phases of actor τ_2 in G, shown in Figure 1, are $S_2(1) = 8$ and $S_2(2) = S_2(1) + C_2(1) = 10$, as illustrated in Figure 2(b).

5.3 Deriving Channel Buffer Sizes

We proved that ISPS has bounded buffer sizes in Section 5.1. Those bounds are sufficient but not minimum. Therefore, we want to derive the minimum buffer sizes that guarantee periodic execution of tasks corresponding to actor phases.

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

Lemma 3. *For an acyclic CSDF graph G, the minimum buffer size b_u of a communication channel $e_u = (\tau_i, \tau_j)$ under ISPS is given by:*

$$b_u = \max_{k\in[0,\alpha+\Delta_j(P_j)]}\{\underset{[S_i(1),\max\{S_i(1),S_j(1)\}+k]}{\mathsf{prd}}(\tau_i) - \underset{[S_j(1),\max\{S_i(1),S_j(1)\}+k)}{\mathsf{cns}}(\tau_j)\}, \quad (18)$$

where $S_i(1)$ is the earliest start time of the first phase of a predecessor actor τ_i, $\alpha = r_iT_i = r_jT_j$, $\Delta_j(P_j) = S_j(P_j) - S_j(1)$, $\mathsf{prd}_{[t_s,t_e]}(\tau_i)$ is the number of tokens produced by τ_i during the time interval $[t_s, t_e]$, and $\mathsf{cns}_{[t_s,t_e)}(\tau_j)$ is the number of tokens consumed by τ_j during the time interval $[t_s, t_e)$.

Proof. Equation (18) tracks the maximum cumulative number of unconsumed tokens on channel e_u during one iteration of τ_i and τ_j. We have two cases: ***Case 1***: $\underline{S_j(1) \geq S_i(1)}$: Here we have two intervals $[S_i(1), S_j(1))$ and $\overline{[S_j(1), S_j(1) + \alpha + \Delta_j(P_j)]}$. During the first interval only phases of actor τ_i are executing, so tokens are only produced and buffer size should be large enough to accommodate all produced tokens in that interval. During the second interval phases of both actors execute in parallel. Thus, the minimum number of tokens that needs to be stored is given by the maximum number of unconsumed tokens on e_u at any time over this interval. At time $t = S_j(1)+\alpha+\Delta_j(P_j)$, both τ_i and τ_j have completed one iteration and the number of tokens on e_u is the same as at time $t = S_j(1) + \Delta_j(P_j)$ [5]. Due to the periodicity of τ_i and τ_j, their execution pattern repeats. Thus, b_u given by Eq. (18) is the minimum buffer size which guarantees periodic execution of τ_i and τ_j. ***Case 2***: $\underline{S_j(1) < S_i(1)}$: Here we have three intervals $[S_j(1), S_i(1))$, $[S_i(1), S_j\overline{(1)+\alpha+\Delta_j(P_j)}]$ and $(S_j(1) + \alpha + \Delta_j(P_j), S_i(1) + \alpha + \Delta_j(P_j)]$. During the first interval there is no production nor consumption or there are initial tokens on the channel, and hence the b_u during that interval is equal to the number of initial tokens. During the second interval phases of both actors execute in parallel and b_u gives the maximum number of unconsumed tokens on e_u. During the third interval phases of actor τ_j executes their second iteration, again either there is no consumption, which means that e_u has to accommodate all the tokens produced during this interval or there is consumption and b_u gives the maximum number of unconsumed tokens on e_u. At time $t = S_i(1) + \alpha + \Delta_j(P_j)$, both τ_i and τ_j have completed one iteration and the number of tokens on e_u is the same as at time $t = S_i(1) + \Delta_j(P_j)$ [5]. Due to the periodicity of τ_i and τ_j, their execution pattern repeats. Thus, b_u given by Eq. (18) is the minimum buffer size which guarantees periodic execution of τ_i and τ_j. □

For the example graph G given in Figure 1, the calculated buffer sizes are $[b_1, b_2, b_3] = [4, 15, 4]$ tokens.

5.4 Hard Real-Time Schedulability

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

Theorem 3. *For an acyclic CSDF graph G, let $\mathcal{T}_G$ be a set of periodic task-sets $\mathcal{T}_{\tau_i}$ such that $\mathcal{T}_{\tau_i}$ corresponds to $\tau_i \in V$. $\mathcal{T}_{\tau_i}$ consists of P_i periodic tasks given by:*

$$\tau_i(\varphi) = (S_i(\varphi), C_i(\varphi), D_i, T_i),\ 1 \leq \varphi \leq P_i, \quad (19)$$

where $S_i(\varphi)$ is the earliest start time of a phase φ of actor τ_i given by Eq. (13) and Eq. (15), $C_i(\varphi)$ is the WCET value of a phase φ given by Eq. (7), D_i is the implicit deadline, and T_i is the period of $\mathcal{T}_{\tau_i}$ given by Eq. (10). $\mathcal{T}_G$ is schedulable on m processors using a hard real-time scheduling algorithm A for implicit-deadline periodic tasks if:

1. *A is partitioned Earliest Deadline First, partitioned Rate Monotonic, partitioned Deadline Monotonic or hierarchical global hard real-time scheduling algorithm,*
2. *$\mathcal{T}_G$ satisfies the schedulability test of A on m processors,*
3. *every communication channel $e_u \in E$ has a capacity of at least b_u tokens, where b_u is given by Eq. (18).*

Proof. According to Theorem 2, the graph is converted into strictly periodic tasks. The relative deadline of these strictly periodic tasks is set to be equal to their period. If all actors in a graph have phases which are not data-dependent (*stateless* actors) then the corresponding tasks become implicit-deadline periodic (IDP) tasks, hence the tasks can be scheduled by any hard real-time scheduling algorithm for IDP tasks which satisfies the schedulability test of task-sets on the corresponding platform. If a graph contains an actor τ_i which phases are data-dependent (*stateful* actor) then the corresponding task-set $\mathcal{T}_{\tau_i}$ of this actor should be scheduled in a way which preserves the dependency between the actor phases. The hard real-time scheduling algorithms which can do this are partitioned Earliest Deadline First (EDF), Rate Monotonic (RM) [15] and Deadline Monotonic (DM) [13], or hierarchical [10], [14]. In case of the partitioned algorithms, tasks which correspond to phases of an actor with data-dependent phases should be allocated to the same processor and scheduled by EDF or DM because the deadlines of the phases are in the same order as the phases themselves, or by RM fixed priority scheduler where ties should be broken in favor of jobs arrived earlier in a system. In hierarchical scheduling a set of tasks are grouped together and scheduled as a single entity - server task or supertask. When the entity is scheduled, one of its tasks is selected to execute according to an internal scheduling policy. Hence, the supertasks/servers are scheduled globally, while the scheduling of the tasks within a supertask/server is done locally, i.e., it is analogous to scheduling on uniprocessor. By grouping the tasks which correspond to phases of an actor with data-dependent phases into a supertask/server and scheduling them by a scheduler which preserves their order (e.g. EDF) the synchronization problem of such dependent tasks is solved.

If tasks which correspond to different phases of a stateless actor are scheduled to execute on different processors, it may happen that they produce/consume tokens out-of-order. In that case, every task corresponding to a phase must know where to write or read from the communication buffer. A writing/reading pattern which ensures correct token production/token consumption follows the one which would be obtained if the tasks corresponding to different phases of an actor wrote/read in-order to a buffer that is implemented as a circular buffer. The buffer sizes derived in Section 5.3 are large enough to guarantee that tokens produced by tasks corresponding to different phases of an actor will never overwrite the unconsumed tokens of each other. □

5.5 Performance Analysis

Once an acyclic CSDF graph has been converted to a set of strictly periodic tasks, the calculated task parameters are used for performance analysis of the graph.

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

$$\mathcal{R}(G) = \frac{1}{\alpha} = \frac{1}{r_i\check{T}_i}, \tau_i \in V, \quad (20)$$

where $\breve{T}_i$ is calculated by Eq. (10). Given that during one graph iteration every actor $\tau_i \in V$ is executed q_i times, the throughput of each actor is calculated as:

$$\mathcal{R}_i = \frac{q_i}{\alpha} = \frac{P_i}{\breve{T}_i}, \tau_i \in V. \tag{21}$$

Theorem 4. *For any acyclic CSDF graph G scheduled by* ISPS*, the throughput of the graph is never less than the graph throughput when G is scheduled by* SPS*.*

Proof. The throughput of a graph scheduled under SPS is $1/\alpha^{\mathrm{SPS}} = 1/(q_i T_i^{\mathrm{SPS}})$, $\tau_i \in V$. If the same graph is scheduled under our ISPS, then its throughput is $1/\alpha^{\mathrm{ISPS}} = 1/(r_i T_i^{\mathrm{ISPS}})$, $\tau_i \in V$. By using Eq. (6) and Eq. (10) and denoting $u = \max_{\tau_j \in V}\{r_j \sum_{\varphi=1}^{P_j} C_j(\varphi)\}$ and $w = \max_{\tau_j \in V}\{q_j \max_{1 \le \varphi \le P_j}\{C_j(\varphi)\}\}$, we can write the relation which we want to prove, i.e., $\alpha^{\mathrm{ISPS}} \le \alpha^{\mathrm{SPS}}$, as follows:

$$\mathrm{lcm}(\vec{r})\left\lceil \frac{u}{\mathrm{lcm}(\vec{r})} \right\rceil \le \mathrm{lcm}(\vec{q})\left\lceil \frac{w}{\mathrm{lcm}(\vec{q})} \right\rceil. \tag{22}$$

We have that $u \le w$. Now, we want to analyze the relation between $\mathrm{lcm}(\vec{r})$ and $\mathrm{lcm}(\vec{q})$. Given that the least common multiple of positive integer numbers can be found using prime factorization, and the relation between vectors $\vec{r} = [r_1, \cdots, r_N]^T$ and $\vec{q} = \mathbf{P} \cdot \vec{r} = [P_1 r_1, \cdots, P_N r_N]^T$, we can write:

$$\mathrm{lcm}(\vec{r}) = \prod_i p_i^{\max_{1 \le j \le N}\{a_j^i\}} \text{and}\, \mathrm{lcm}(\vec{q}) = \prod_i p_i^{\max_{1 \le j \le N}\{a_j^i + b_j^i\}},$$

where p_i is a prime number, a_j^i is a power of p_i in the representation of r_j and b_j^i is a power of p_i in the representation of P_j. For each prime number p_i, $\max_{1 \le j \le N}\{a_j^i\}$ is not grater than $\max_{1 \le j \le N}\{a_j^i + b_j^i\}$, which means that we can write $\mathrm{lcm}(\vec{q})/\,\mathrm{lcm}(\vec{r}) = \prod_i p_i^{k_i}$, with $k_i = \max_{1 \le j \le N}\{a_j^i + b_j^i\} - \max_{1 \le j \le N}\{a_j^i\}$, and $k_i \ge 0$. Thus, $\mathrm{lcm}(\vec{q})$ is divisible by $\mathrm{lcm}(\vec{r})$.

Finally, to prove relation (22) we consider the following cases (with regard to divisibility by the corresponding *lcm* term): ***Case 1***: workloads on both sides of inequality (22) are divisible by the corresponding *lcm* terms. By removing the ceiling operation we obtain inequality $u \le w$, which always holds. ***Case 2***: u is divisible by $\mathrm{lcm}(\vec{r})$. We can represent the ceiling operation on the right-hand side as $(w + \mathrm{lcm}(\vec{q}) - w \bmod (\mathrm{lcm}(\vec{q})))/\,\mathrm{lcm}(\vec{q})$. In the worst case $w \bmod (\mathrm{lcm}(\vec{q}))$ is equal to $\mathrm{lcm}(\vec{q}) - 1$. By putting this into Ineq. (22) we obtain $u \le w + 1$, which holds. ***Case 3***: w is divisible by $\mathrm{lcm}(\vec{q})$ (also divisible by $\mathrm{lcm}(\vec{r})$). We can represent u and w as $k_u\,\mathrm{lcm}(\vec{r}) + u \bmod (\mathrm{lcm}(\vec{r}))$ and $k_w\,\mathrm{lcm}(\vec{r})$, respectively, for some integer constants k_u and k_w, $k_u < k_w$. We represent the ceiling operation as in *Case 2*, so Ineq. (22) becomes $u + \mathrm{lcm}(\vec{r}) - u \bmod (\mathrm{lcm}(\vec{r})) \le w$. Now, by putting the k_u-representation of u and k_w-representation of w, the inequality becomes $k_u + 1 \le k_w$, which is true and thus, Ineq. (22) holds. ***Case 4***: workloads on both sides of Ineq. (22) are not divisible by the corresponding *lcm* terms. Similarly to *Case 2* and *Case 3*, we can represent the ceiling operation through the modulo operation. In the worst case, we have on the right-hand side the smallest possible value which is $w + 1$, which means that this value now is divisible by both $\mathrm{lcm}(\vec{q})$ and $\mathrm{lcm}(\vec{r})$. In the worst case $u = w$, which means that u also needs only 1 unit to be rounded up to a value divisible by $\mathrm{lcm}(\vec{r})$. Thus, Ineq. (22) becomes $w + 1 \le w + 1$, which holds. $\square$

The other performance metric of a graph is the latency. The latency of G scheduled by ISPS is given by:

$$\mathcal{L}(G) = \max_{w_{\mathrm{in}\to\mathrm{out}} \in \mathcal{W}} \{S_{\mathrm{out}}(g_{\mathrm{out}}^C) + T_{\mathrm{out}} - S_{\mathrm{in}}(g_{\mathrm{in}}^P)\}, \tag{23}$$

where $\mathcal{W}$ is the set of all paths from any input actor τ_{in} to any output actor τ_{out}, and $w_{\mathrm{in}\to\mathrm{out}}$ is one path of the set. $S_{\mathrm{out}}(g_{\mathrm{out}}^C)$ and $S_{\mathrm{in}}(g_{\mathrm{in}}^P)$ are the earliest start times of the first phase of τ_{out} with non-zero token consumption (phase g_{out}^C) and the first phase of τ_{in} with non-zero token production (phase g_{in}^P) on a path $w_{\mathrm{in}\to\mathrm{out}} \in \mathcal{W}$, respectively. T_{out} is the common period of τ_{out}.

Algorithm 2: Procedure to derive the number of processors.

```
Input: A CSDF graph $G = (V, E)$, a partitioned scheduling algorithm $A$,
       an allocation heuristic $H$.
Output: Number of processors $m_{\mathrm{PAR}}$, task allocation alloc.
1  for actor $\tau_i$ in $V$ do
2      Compute the minimum common period $\breve{T}_i$ by using Eq. (10);
3  $u_{\mathrm{total}} = 0$;
4  for actor $\tau_i \in V$ do
5      $u_i = 0$;
6      for phase $\varphi$ of $\tau_i$, $1 \le \varphi \le P_i$ do
7          $u_i(\varphi) = \frac{C_i(\varphi)}{\breve{T}_i}$;
8          $u_i = u_i + u_i(\varphi)$;
9          $u_{\mathrm{total}} = u_{\mathrm{total}} + u_i(\varphi)$;
10 Find $V_s = \{\tau_i : \tau_i \in V \land \tau_i$ is stateful$\}$;
11 $U \leftarrow \emptyset$; (the set of allocation units, initially empty)
12 for actor $\tau_i \in V_s$ do
13     $U = U \cup u_i$;
14 for actor $\tau_i \in V - V_s$ do
15     for phase $\varphi$ of $\tau_i$, $1 \le \varphi \le P_i$ do
16         $U = U \cup u_i(\varphi)$;
17 $m_{\mathrm{PAR}} = m_{\mathrm{OPT}} = \lceil u_{\mathrm{total}} \rceil$;
18 Reorder elements of $U$ if required by an allocation heuristic $H$;
19 for $u \in U$ do
20     $\Pi = \{\pi_1, \pi_2, \cdots, \pi_{m_{\mathrm{PAR}}}\}$;
21     Apply bin-packing allocation heuristic $H$ to $u$ on $\pi_j \in \Pi$ and check
       a schedulabiility test of algorithm $A$ on $\pi_j$;
22     if $u$ is not allocated to any $\pi_j \in \Pi$ then
23         Allocate $u$ on a new processor $\pi_{m_{\mathrm{PAR}}+1}$;
24         $m_{\mathrm{PAR}} = m_{\mathrm{PAR}} + 1$;
25 return $m_{\mathrm{PAR}}$, alloc;
```

5.6 Deriving the Number of Processors

As introduced in Section 3.2, by using Eq. (4) one can compute the absolute minimum number of processors m_{OPT} needed to schedule the tasks with deadlines equal to the periods. The tasks can be scheduled on m_{OPT} if an optimal scheduling algorithm is used. The optimal scheduling algorithms are either global or hybrid, and hence, they require the task migration. On the other hand, the partitioned scheduling algorithms do not require the task migration. In that case the tasks are first allocated to the processors, and then the tasks on each processor are scheduled using a uniprocessor scheduling algorithm. The problem of allocating tasks onto processors is similar to the bin-packing problem and can be solved using either exact or approximate allocation algorithms. The disadvantage of using an exact algorithm is its high computational complexity. Therefore, many heuristics exist for task partitioning such as First-Fit, Best-Fit, Worst-Fit, etc. [8] which have, in the worst case, a polynomial time complexity.

The procedure to calculate the number of processors required for the partitioned scheduling of the task-set obtained by the conversion procedure described in Section 5.1-5.2 is given in Algorithm 2. The minimum common period for each actor is calculated in lines 1-2 of the algorithm. Once the periods are calculated, then the total utilization of the converted task-set and the utilization per task-set corresponding to an actor are calculated in lines 3-9. Note that deadlines of tasks are equal to their periods. Lines 10-16 in Algorithm 2 ensure that the task-set of an actor which phases are data-dependent (*stateful* actor) is considered as one scheduling entity, i.e., one allocation unit. The absolute minimum number of processors m_{OPT} for scheduling the tasks with deadlines equal to the periods is computed in line 17. Some allocation heuristics require a preprocessing step to be performed on the tasks before applying the heuristic. This preprocessing step is usually sorting the tasks based on some criteria, such as their utilization. That step is done in Algorithm 2 in line 18. The following lines find the number of processors and the allocation of tasks to processors. Given that m_{OPT} is the lower bound on the number of processors m_{PAR} needed by partitioned scheduling algorithms, Algorithm 2 starts with the task partitioning on m_{OPT} processors. If the tasks pass the schedulability test on all m_{PAR} processors, i.e., the utilization of the tasks allocated to a processor is not greater than the corresponding utilization bound (e.g., for EDF utilization bound is 1), then the algorithm returns m_{PAR} and the corresponding allocation of the tasks to the processors *alloc*.

Let us now analyze the time complexity of Algorithm 2 in the worst case. The first **for loop** in lines 1-2 takes linear time to calculate the minimum common period of each actor, i.e., its time complexity is $O(|V|)$. The second **for loop** in lines 4-9 has a nested for loop and hence, its time complexity in the worst case is given by $O(|V|P)$, where P is the maximum number of execution phases per actor, i.e., $P = \max_{\tau_i \in V}\{P_i\}$. Lines 10-13 of Algorithm 2 run in linear time in the worst case, with complexity $O(|V|)$. The **for loop** in lines 14-16 is similar to the loop in lines 4-9 and thus, it takes $O(|V|P)$ time to construct a set of scheduling entities. If the task sorting in line 18 should be performed prior to performing the task allocation, it will have $O(|V|P \log(|V|P))$ time complexity given that the maximum number of tasks is $|V|P$. The **for loop** in lines 19-24 implements the allocation of the tasks to the processors by applying certain allocation heuristic and scheduling algorithm. Given that the maximum number of tasks is $|V|P$ and the maximum number of processors needed to allocate and schedule an CSDF graph is equal to the number of actors in the graph $|V|$, the time complexity of finding the number of processors m_{PAR} and the feasible task allocation is $O(|V|P \log |V|)$ [21]. Thus, we can conclude that the running time of Algorithm 2 is polynomial and its complexity is $O(|V|P \log |V|)$ or $O(|V|P \log(|V|P))$ if the preprocessing step is performed.

6. EVALUATION

We evaluate our approach in terms of its performance and time complexity by performing experiments on the benchmarks given in Table 2. Columns 3, 4 and 5 in Table 2 give for each benchmark the number of actors $|V|$, the number of channels $|E|$ in the corresponding CSDF graph of a benchmark, and the number of periodic tasks $|\mathcal{T}|$ obtained after converting the actors of the CSDF graph by our approach to a set of periodic tasks $\mathcal{T}$. The execution times of the benchmarks are given in clock cycles [6] or in time units [4], [20]. If the execution times of a benchmark are not given [5], [17], [16], certain values based on a static analysis are assumed. The execution times of benchmark [23] are obtained from the measurements of the benchmark running on a MicroBlaze processor.

Our approach is evaluated by comparison to 3 related scheduling approaches - *strictly periodic scheduling*, SPS, proposed in [2], *periodic scheduling*, PS, presented in [6], and *self-timed scheduling*, STS, given in [19]. We implemented our approach in Python. The SPS approach was implemented in Python within the *darts* tool-set [1]. The approach in [19] was implemented in C++ within the SDF^3 tool-set [18]. In addition, we implemented the approach from [6] in Python as well. We formulated both LP problems [6] for finding the period of a graph, and for finding the start times and the buffer sizes as integer linear programming (ILP) problems, and we added the constraint that the periods of all actors in a graph have to be integers. We used CPLEX Optimization Studio [11] to solve the ILP problems. We run all the experiments on a Dell PowerEdge T710 server running Ubuntu 11.04 (64-bit) Server OS.

6.1 Performance of the ISPS approach

The main objective of the evaluation is to compare the throughput of streaming applications and the required number of processors to guarantee the throughput when scheduled by our ISPS with the throughput and the number of processors under SPS [2], PS [6] and STS [19]. In addition, we compare our ISPS and the other scheduling approaches in terms of latency and memory resources needed to implement the communication channels.

We used the `sdf3analysis-csdf` tool from SDF^3 [18] to obtain the maximum achievable throughput of a graph, which is the throughput under STS, and to compute the minimum buffer sizes required to achieve that throughput. Unfortunately, the `sdf3-analysis-csdf` tool does not support the latency calculation and the calculation of the number of processors. Thus, we were not able

Table 2: Benchmarks used for evaluation.

Domain	Benchmark	$\|V\|$	$\|E\|$	$\|\mathcal{T}\|$	Source
Medical	Heart pacemaker	4	3	67	[17]
Communication	Reed Solomon Decoder (RSD)	6	6	904	[4]
Financial	BlackScholes	41	40	261	[6]
Computer Vision	Disparity map	5	6	11	[23]
	Pdetect	58	76	4045	[6]
Audio processing	CELP algorithm	9	10	167	[5]
	CD2DAT rate converter	6	5	22	[16]
	MP3 Playback	4	3	8	[20]
Video processing	JPEG2000	240	703	639	[6]

to compare them with our approach. We were also not able to obtain the number of processors for a graph scheduled under PS, because the calculation of the number of processors was not considered in [6].

Results of the performance evaluation are given in Table 3. We report the throughput of the output actors under ISPS, calculated by Eq. (21), in the second column of Table 3. Here t.u. denotes the corresponding time unit of a benchmark. Columns 7, 12 and 15 show the ratio between the throughput of the output actors under our ISPS and SPS, PS and STS, respectively. For processor requirements in case of ISPS and SPS, we compute the minimum number of processors under optimal and partitioned First-Fit Decreasing EDF (FFD-EDF) schedulers by using Eq. (4) and Algorithm 2 for ISPS, and Eq. (4) and Eq. (5) for SPS - see columns 4, 5, 9 and 10. By comparing the throughputs under ISPS and SPS, we can see that for the majority of the benchmarks the throughput under our ISPS is higher than the corresponding throughput under SPS. Only in two cases the throughputs are the same for both schedules. The first case is MP3 Playback, which bottleneck actor (the actor with the biggest workload over one iteration period) is the same under both SPS and ISPS, and that actor has only one phase, so the influence of different WCET for actor phases on throughput cannot be seen. However, the influence can be seen from the required number of processors needed for scheduling of MP3 Playback, which is smaller in the case of our ISPS. The second case is CD2DAT. For this benchmark $\text{lcm}(\vec{q})$ and $\text{lcm}(\vec{r})$ are equal and much higher than the maximum workload of actors over an iteration period for both SPS and ISPS, which leads to the same iteration period for both schedules. However, the WCET-awareness of ISPS leads to smaller number of processors. Note that if we want to schedule a task-set on smaller number of processors than the one calculated by Eq. (4) or Eq. (5)/Algorithm 2, we should scale up the computed actor periods by the same scaling factor [22]. Hence, to schedule CD2DAT by SPS on the same number of processors required by ISPS, we need to scale up actor periods by 2, which will lead to decrease in throughput by 2. Thus, ISPS outperforms SPS in terms of throughput when CD2DAT is scheduled on 1 processor. Benchmarks JPEG2000 and RSD can achieve much better throughput when scheduled under ISPS, but in that case they require larger number of processors to be scheduled. Note that the throughputs of these two benchmarks cannot be increased under SPS even when the number of processors is increased. If we apply the period scaling technique [22] for these two benchmarks to schedule them under ISPS on the same number of processors as required under SPS the throughput values for JPEG2000 and RSD under our ISPS are 3.93 and 11.2 times higher, as given in column 7 in parenthesis, than the corresponding values under SPS. Therefore, we can conclude that in all cases the minimum number of processors required to guarantee certain throughput under ISPS is smaller than or equal to the minimum number of processors under SPS while the throughput under ISPS is increased in most cases, thus, processors are better utilized.

Column 12 in Table 3 shows the ratio of the maximum throughput of the output actors achieved by our ISPS to the maximum throughput of the output actors achieved by PS. We can see that both approaches give the same throughput for all benchmarks, which is expected given that PS schedules phases of an actor in a CSDF graph statically within a period of the actor, hence the scheduling granularity is similar between these two approaches.

Table 3 shows in column 15 the ratio of the maximum throughput of the output actors achieved by our approach to the absolute maximum throughput of the output actors achieved by self-timed scheduling of actor firings, which is the optimal scheduling in terms of throughput. We can see that throughput under ISPS is equal or very close to the throughput under STS for the majority of the benchmarks. Difference in throughput appears as a result of the ceiling operation during calculation of actor common periods in Eq. (10). The biggest difference is in the case of CD2DAT benchmark. For this benchmark lcm($\vec{r}$) is much higher than the maximum workload of actors over an iteration period, and thus, the calculated actor periods are underutilized, which leads to lower throughput. The throughput value N/A for JPEG2000 indicates that the SDF^3 tool-set [18] returned an infeasible throughput (most likely related to an integer overflow).

Let us now analyze the latency and the memory resources needed to implement the communication channels of the benchmarks. The graph latency under ISPS is calculated by Eq. (23) and shown in column 3 of Table 3. As we can see from columns 4, 5, 7-10 in Table 3: for 4 benchmarks (highlighted in the table) under ISPS we obtain higher throughput and smaller latency than under SPS without increasing the number of processors (with JPEG2000 and RSD scheduled on the same number of processors as in case of the SPS); for the other 3 benchmarks (BlackScholes, Disp. map, Pdetect) the obtained increase in throughput is less than the increase in latency on a platform with the same (or 1 less for BlackScholes under ISPS, partitioned scheduling) number of processors; for the rest 2 benchmarks we obtained the same throughput with the increase in latency, but also with the decrease in the number of processors. For the tested benchmarks, the calculated buffer sizes under ISPS are never smaller than the buffer sizes under SPS, see column 11 in Table 3. The highest ratio in buffer sizes between ISPS and SPS is obtained for BlackScholes and CD2DAT. However, the actual increase in communication memory resources is 215 KB and less than 1 KB, respectively, which is acceptable given the size of memory available in modern embedded systems. Note that both latency and buffer sizes can be reduced by carefully selecting deadlines for individual actors (actors phases). However, in that case the calculation of the number of processors is more complex and most likely, bigger number of processors might be needed to schedule the benchmarks.

Column 13 gives the ratio of the maximum latency of benchmarks under ISPS to the latency of benchmarks under PS. Although [6] does not provide the latency calculation, we were able to extract the latency information from the start times obtained by solving the ILP problem. However, for benchmarks JPEG2000 and Pdetect we could not get a solution from the ILP solver after more than 1 day, so we could not calculate the latency for these two benchmarks. As we can see, the average latency of benchmarks under ISPS is four times the latency under PS. As mentioned above, reducing the latency under ISPS can be done by carefully selecting deadlines for individual actors (actors phases). Moreover, ISPS reports the maximum latency while PS reports the actual latency under a certain schedule. The ratio of the calculated buffer sizes under ISPS to the calculated buffer sizes under PS and STS is given in columns 14 and 16, respectively. Again, for benchmarks JPEG2000 and Pdetect under PS we could not get a solution from the ILP solver after more than 1 day. Similarly, for benchmarks RSD, BlackScholes, Pdetect and CELP under STS we could not get a solution for longer than 1 day. As mentioned before, value N/A for JPEG2000 indicates that SDF^3 tool-set returned an infeasible throughput, and hence the buffer sizes were not calculated. As we can see, the buffer sizes under PS and STS are always smaller than the buffer sizes under ISPS. The highest ratio in buffer sizes between ISPS and PS is obtained for BlackScholes and CD2DAT, with the actual increase in communication memory resources of 232 KB and less than 1 KB, respectively. The highest increase in buffer sizes under ISPS when compared to STS is less than 1 KB. The reason of difference in the buffer sizes is that in both PS and STS approaches it is assumed that the production of tokens happens at the end of the actor firing, while the consumption happens at the start of the firing, while in our case (and in SPS case) the worst-case scenario is considered, i.e., the production of tokens happens at the earliest possible start of the actor firing (at start times), while the consumption happens at the latest possible end of actor firing (at deadlines). Note that in an implementation of a dataflow application, data may be consumed from input channels and produced to output channels at arbitrary points in time during an actor firing. To guarantee that buffer overflow/underflow does not occur, buffer sizes have to be sufficiently large. Thus, the assumption in PS and STS limits the actual implementation of reading and writing of tokens, while the buffers calculated in our case are valid regardless of the actual point in time where reading and writing of tokens happens and thus, our approach does not limit the implementation of the reading and writing of tokens. Moreover, the buffer sizes calculated in PS and STS are valid for that specific schedule and the specific production/consumption pattern, while in the case of our ISPS the computed buffer sizes are valid for any schedule of actor firings during its period and for any production/consumption pattern during its firing.

6.2 Time complexity of the ISPS approach

In this section we evaluate the efficiency of our ISPS approach in terms of the execution time of our algorithms to calculate the throughput of an application, and to find a schedule and buffer sizes of communication channels. The execution times are given in Table 4. We compare these execution times with the corresponding execution times of related approaches – SPS, PS and STS.

Let us first analyze the time needed to calculate the throughput of an application. The execution times needed to find the application throughput under ISPS, SPS, PS and STS are given in columns 2, 4, 6 and 8, respectively. As we can see, the times spent on calculating the throughput of an application under ISPS and SPS are similar and much shorter than the time needed for solving the ILP problem to find the application throughput under PS and the time spent on finding the maximum achievable throughput of the application, i.e., the throughput under STS. Thus, our approach outperforms PS and STS in terms of time required to calculate the throughput of an application. Given that in most cases ISPS gives higher throughput of an application than SPS within almost the same time, we can say that ISPS outperforms SPS as well.

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

We report in Table 5 the execution time of calculating the minimum number of processors needed to temporally schedule the tasks, obtained by the conversion of an application by using our ISPS approach, under global optimal and partitioned FFD-EDF schedulers. In the case of global optimal scheduling, the minimum number of processors is calculated by Eq. (4), while the calculation procedure for FFD-EDF partitioned scheduling is presented in Algorithm 2 in Section 5.6. As we can see, the number of processors in the case of optimal scheduling can be calculated within a millisecond for most of the benchmarks, while in the case of partitioned scheduling the calculation is done within less than 12 milliseconds for most cases and within less than 420 milliseconds in two cases. Thus, the calculation of the number of processors required to schedule an application under our ISPS is very efficient. We obtained similar times for the calculation of the number of processors under SPS and global and partitioned FFD-EDF schedulers. We could not numerically compare the time complexity of our approach with

Table 3: Comparison of different scheduling approaches.

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

Table 4: Time complexity (in seconds) of different scheduling approaches.

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

Table 5: Time complexity (in seconds) for the calculation of number of processors.

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

regard to the PS approach because the calculation of the number of processors was not considered in [6]. As mentioned already in Section 2, one possible way to find the minimum number of processors under PS is to trace the schedules, but that procedure has an exponential time complexity in the worst case, whereas our Algorithm 2 for finding the minimum number of processors under ISPS has a polynomial time complexity, see Section 5.6. Finding the minimum number of processors under STS requires complex Design Space Exploration (DSE) procedures, with an exponential time complexity in the worst case, to find the best allocation which delivers the maximum achievable throughput. The SDF^3 tool-set used to compute the self-timed scheduling parameters does not support such design space exploration for self-timed scheduling. Thus, we could not numerically compare the time complexity of ISPS with the time complexity of STS. However, given that our approach finds the minimum number of processors for scheduling an application in polynomial time in the worst case, as shown in Section 5.6, we can conclude that our ISPS is faster than STS.

7. CONCLUSIONS

In this paper, we presented a scheduling approach which converts each actor in a CSDF graph, by considering different WCET value for each actor phase, to a set of strictly periodic tasks. As a result, a variety of hard real-time scheduling algorithms can be applied to temporally schedule the graph on a platform with calculated number of processors with a certain guaranteed throughput and latency. The experiments on a set of real-life applications showed that our approach gives tighter guarantee on the throughput and better processor utilization with acceptable increase in terms of communication memory requirements when compared with the existing hard real-time scheduling approach. When compared with the existing periodic scheduling approach for CSDF graphs, our proposed approach gives the same throughput with increased communication memory, but takes much shorter time for deriving the schedule, the calculation of the minimum number of processors and the calculation of the size of communication buffers. Finally, our approach gives the throughput that is equal or very close to the absolute maximum throughput achieved by self-timed scheduling of actor firings, but requires much shorter time to derive the schedule.

8. REFERENCES

[1] M. Bamakhrama. http://daedalus.liacs.nl/darts.

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

[3] S. K. Baruah et al. Proportionate progress: A notion of fairness in resource allocation. In *STOC*, 1993.

[4] M. Benazouz et al. A new method for minimizing buffer sizes for cyclo-static dataflow graphs. In *ESTIMedia*, 2010.

[5] G. Bilsen et al. Cyclo-static dataflow. *IEEE Trans. Signal Process.*, 44(2):397–408, 1996.

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

[7] A. Bouakaz, J.-P. Talpin, and J. Vitek. Affine data-flow graphs for the synthesis of hard real-time applications. In *ACSD*, 2012.

[8] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: A survey. In *Approximation algorithms for NP-hard problems*, pages 46–93. 1996.

[9] J. P. H. M. Hausmans et al. Two parameter workload characterization for improved dataflow analysis accuracy. In *RTAS*, 2013.

[10] P. Holman and J. H. Anderson. Group-based Pfair scheduling. *Real-Time Systems*, 32(1–2):125–168, 2006.

[11] IBM. IBM ILOG CPLEX Optimization Studio V12.4.

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

[13] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[14] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *ECRTS*, 2003.

[15] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[16] H. Oh and S. Ha. Fractional rate dataflow model for efficient code synthesis. *J. of VLSI Signal Process.*, 37(1):41–51, 2004.

[17] R. Pellizzoni et al. Handling mixed-criticality in SoC-based real-time embedded systems. In *EMSOFT*, 2009.

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

[19] S. Stuijk, M. Geilen, and T. Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Trans. on Computers*, 57(10):1331–1345, 2008.

[20] M. Wiggers et al. Efficient computation of buffer capacities for cyclo-static real-time systems with back-pressure. In *RTAS*, 2007.

[21] O. U. P. Zapata and P. M. Alvarez. EDF and RM multiprocessor scheduling algorithms: Survey and performance evaluation. Technical Report CINVESTAV-CS-RTG-02, 2004.

[22] J. T. Zhai, M. Bamakhrama, and T. Stefanov. Exploiting just-enough parallelism when mapping streaming applications in hard real-time systems. In *DAC*, 2013.

[23] C. L. Zitnick and T. Kanade. A cooperative algorithm for stereo matching and occlusion detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(7):675–684, 2000.