On the Improved Hard Real-Time Scheduling of Cyclo-Static Dataflow

JELENA SPASIC, DI LIU, EMANUELE CANNELLA, and TODOR STEFANOV,
Leiden University

Recently, it has been shown that the hard real-time scheduling theory can be applied to streaming applications modeled as acyclic Cyclo-Static Dataflow (CSDF) graphs. However, this recent approach is not always efficient in terms of throughput and processor utilization. Therefore, in this article, we propose an improved hard real-time scheduling approach to schedule streaming applications modeled as acyclic CSDF graphs on a Multiprocessor 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 that offer fast calculation of the required number of processors for scheduling the tasks. In addition, we propose a method to reduce the graph latency when the converted tasks are scheduled as real-time periodic 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) delivers shorter application latency by applying the proposed method for graph latency reduction while providing better throughput and processor utilization when compared to the existing hard real-time scheduling approach; (3) 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, and so on); and (4) gives the throughput that is equal to 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.

CCS Concepts: • **Theory of computation → Models of computation**; • **Computer systems organization → Embedded systems**; **Real-time systems**; • **Software and its engineering → Scheduling;**

Additional Key Words and Phrases: Streaming applications, cyclo-static dataflow, multiprocessor system-on-chip, hard real-time scheduling

ACM Reference Format:
Jelena Spasic, Di Liu, Emanuele Cannella, and Todor Stefanov. 2016. On the improved hard real-time scheduling of cyclo-static dataflow. ACM Trans. Embed. Comput. Syst. 15, 4, Article 68 (August 2016), 26 pages.
DOI: http://dx.doi.org/10.1145/2932188

1. INTRODUCTION

Modern streaming applications have high computational demands, and should deliver high-quality output. As a huge amount of data should be processed in a "short" time interval, parallel processing is a natural solution. The processing power of Multiprocessor

Authors' addresses: J. Spasic, D. Liu, E. Cannella, and T. Stefanov, Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands, 2333CA, Niels Bohrweg 1; emails: {j.spasic, d.liu, t.p.stefanov}@liacs.leidenuniv.nl, emanuele.cannella@gmail.com.

DOI: http://dx.doi.org/10.1145/2932188

System-on-Chip (MPSoC) platforms perfectly matches the computational requirements of streaming applications. In modern MPSoCs, it is desirable to execute multiple applications, a mixture of streaming applications and control (hard) real-time applications, simultaneously in order to efficiently utilize the resources in an MPSoC. To deliver high-quality output of multiple running applications, together with the ability to dynamically start/stop applications without affecting other already running applications, streaming applications have tight timing requirements that often make it necessary to treat them as hard real-time applications [Moreira et al. 2005]. Designing such an embedded system imposes several challenges: a streaming application should be represented in a way that reveals the parallelism of the application, and it should be mapped and scheduled on a platform such that the timing requirements are satisfied.

To address these challenges, several parallel Models-of-Computation (MoCs), for example, Synchronous Data Flow (SDF) [Lee and Messerschmitt 1987] and Cyclo-Static Dataflow (CSDF) [Bilsen et al. 1996], have been adopted as the parallel application specification. Within an 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 that an application can produce during a given time interval. 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 runtime 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; it is, in the worst case, of a polynomial time complexity.

Recently, Bamakhrama and Stefanov [2013] 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 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, Bodin et al. [2013], the authors proposed a framework to evaluate a lower bound of the maximum throughput of a periodically scheduled CSDF-modeled application. However, the authors do not provide a method to determine the number of processors required for scheduling the application. Moreover, their approach does not ensure temporal isolation among applications, that is, the schedule of applications has to be recalculated once a new application comes in the system; thus, it may be possible that the previously calculated throughput of an application can no longer be reached.

In this article, we address the drawbacks of Bamakhrama and Stefanov [2013] and Bodin et al. [2013] by considering different WCET values for a task's phases in an

acyclic CSDF graph and enabling temporal isolation of applications while providing hard real-time guarantees. The contributions of this article 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 5.10).
—We prove that our scheduling approach gives equal or higher throughput than the existing hard real-time scheduling approach for acyclic CSDF graphs (Theorem 5.11).
—We propose a method for reducing the latency of an acyclic CSDF graph scheduled as a set of strictly periodic real-time tasks (Section 5.5.2).
—We show, on a set of real-life streaming applications, that scheduling each execution phase of a CSDF task as a strictly periodic task and considering different WCET per phase lead not only to tighter guarantee on the throughput of an application but also to better utilization of processor resources (Section 6.1).
—We demonstrate, on a set of real-life streaming applications, that the total time required by our approach to derive the schedule of the tasks, 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 (Section 6.2).
—We show, on a set of real-life streaming applications, that the latency of the applications scheduled by our scheduling approach can be reduced by our proposed latency reduction method in most cases to the desirable latency values while keeping higher or equal application throughput and requiring an equal or smaller number of processors in comparison to the existing scheduling approaches (Section 6.3).

Scope of the work. In this work, we assume that a given CSDF graph is acyclic. However, even with this limitation, our approach is still applicable to many real-life streaming applications because Thies and Amarasinghe [2010] have shown that around 90% of streaming applications can be modeled as acyclic CSDF graphs. We assume a homogeneous MPSoC platform with predictable communication infrastructure, that is, the communication infrastructure provides guaranteed communication latency. We use the worst-case communication latency to compute the WCET of a task, which, in our approach, includes the worst-case time needed for the tasks' computation and the worst-case time needed to perform intertask data communication on the considered platform.

The remainder of the article 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 presented in Section 6, and Section 7 contains our conclusions.

2. RELATED WORK

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

Hausmans et al. [2013] 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 a task's phases; then, they use the average worst-case execution time to improve the minimum guaranteed throughput/latency. Similar to their approach, we consider different execution times for a task's phases in a CSDF graph. But, in contrast to their approach, we convert a 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, presented in Section 5.6.

Bouakaz et al. [2012] propose an analysis framework for hard real-time applications, modeled as Affine Dataflow Graphs (ADFs). The actors in an ADF graph are scheduled as periodic tasks. The ADF model proposed in Bouakaz et al. [2012] extends the CSDF model, thus 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.

Bodin et al. [2013] propose 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 Bodin et al. [2013] 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 the 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, presented in Section 5.6.

The closest to our work, in terms of scope and methods proposed to schedule streaming applications modeled as acyclic CSDF graphs, is the work in Bamakhrama and Stefanov [2013]. Bamakhrama and Stefanov [2013] 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 Bamakhrama and Stefanov [2013] as follows: we use different WCET values for each execution phase of a task and each phase is converted to a periodic task; in Bamakhrama and Stefanov [2013], only one WCET value is used for a task and every execution of a task is periodic with a calculated period. By considering different WCET values for each task phase and converting each phase to a periodic task, we can guarantee tighter throughput and better utilization of processor resources.

3. BACKGROUND

In this section, we first introduce the application model, that is, the CSDF MoC, followed by the system model that we use. We then review the scheduling framework proposed in Bamakhrama and Stefanov [2013], 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 [Bilsen et al. 1996] is a directed graph $G = (V, E)$ that consists of a set of actors V that communicate with each other through a set of communication channels E. Actors represent a certain functionality of the application,

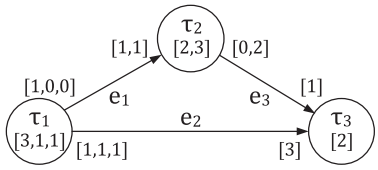

Fig. 1. A CSDF graph G.

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* $[f_i(1), f_i(2), \ldots, f_i(P_i)]$ of length P_i, that is, 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 $[C_i^C(1), C_i^C(2), \ldots, C_i^C(P_i)]$ 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* $[x_i^u(1), x_i^u(2), \ldots, x_i^u(P_i)]$. Analogously, token consumption on every input channel e_u of an actor τ_i is a predefined sequence $[y_i^u(1), y_i^u(2), \ldots, y_i^u(P_i)]$, 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 acyclic CSDF graph can be partitioned into a number of levels, denoted by L, 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.

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

THEOREM 3.1 (FROM BILSEN ET AL. [1996]). *In a CSDF graph G, a repetition vector* $\vec{q} = [q_1, q_2, \ldots, q_N]^T$ *is given by*

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

where $\vec{r} = [r_1, r_2, \ldots, r_N]^T$ *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*

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

A CSDF graph G is said to be consistent if a positive integer solution $\vec{r} = [r_1, r_2, \ldots, r_N]^T$ exists for the balance equation, Equation (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 article, 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 $\Pi = \{\pi_1, \pi_2, \ldots, \pi_m\}$ of m identical processors. The processors execute a task set $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ 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 in relative time units, where $C_i \leq D_i \leq T_i$. If $D_i = T_i$, then τ_i is said to have an *implicit-deadline*. Otherwise, if $D_i < T_i$, then τ_i is said to have a *constrained deadline*. If all the tasks in a task set $\mathcal{T}$ are implicit-deadline periodic tasks, then we say that $\mathcal{T}$ is an *implicit-deadline periodic (IDP) task set*. Otherwise, we say that $\mathcal{T}$ is a *constrained-deadline periodic (CDP) task set*.

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 $u_\mathcal{T} = \sum_{\tau_i \in \mathcal{T}} u_i$. Similarly, the density of task τ_i is $\delta_i = C_i/D_i$ and the total density of $\mathcal{T}$ is $\delta_\mathcal{T} = \sum_{\tau_i \in \mathcal{T}} \delta_i$. The total utilization of a task set directly determines the minimum number of processors needed to schedule the task set. Given a system Π of m identical processors and a task set $\mathcal{T}$, a necessary condition for $\mathcal{T}$ to be scheduled on Π such that all deadlines are met is given by $u_\mathcal{T} \leq m$. This condition is also a sufficient condition for scheduling an IDP task set $\mathcal{T}$ on Π. Thus, the absolute minimum number of processors needed to schedule a periodic task set $\mathcal{T}$ with deadlines equal to periods is given by Baruah et al. [1993]:

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

Scheduling such $\mathcal{T}$ on m_{OPT} processors is possible only by using optimal scheduling algorithms [Davis and Burns 2011], which are either global or hybrid. However, global and hybrid scheduling algorithms require task migration. Several sufficient tests for global scheduling of a CDP task set are given in Davis and Burns [2011]. The other class of scheduling algorithms for IDP and CDP task sets are partitioned algorithms that 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 the following:

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

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 Bamakhrama and Stefanov [2013], 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 a real-time periodic 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, that is, $C_i = \max_{1 \leq \varphi \leq P_i} \{C_i(\varphi)\}$, where $C_i(\varphi)$ contains the worst-case computation, the worst-case data read, and the worst-case data write times of a phase φ of actor τ_i. To execute graph G strictly periodically, period T_i for each actor τ_i is computed as

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

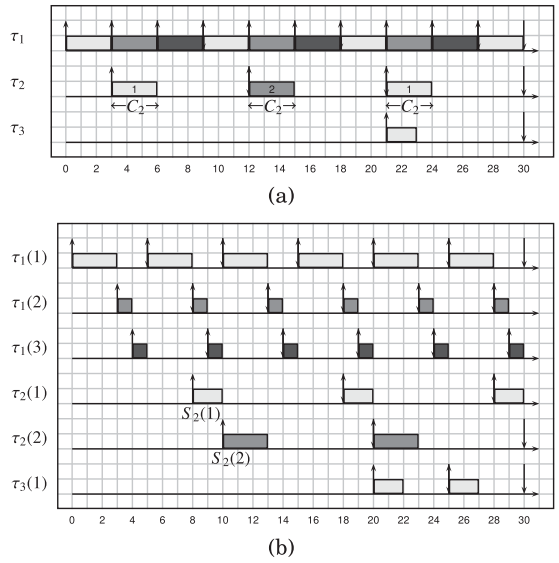

Fig. 2. (a) The SPS and (b) ISPS of graph G in Figure 1.

Table I. 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

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 $1/(q_i T_i)$. Bamakhrama and Stefanov [2013] also provide 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, that is, 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 data 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 phase, the same WCET value is considered, that is, for actor τ_2, we have two phases 1 and 2 and the considered WCET value C_2 for each phase is $C_2 = \max\{C_2(1), C_2(2)\} = \max\{C_2^C(1), C_2^C(2)\} = \max\{2, 3\} = 3$.

4. MOTIVATIONAL EXAMPLE

The goal of this section is to show that the SPS approach [Bamakhrama and Stefanov 2013] 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 to consider different WCET values of actor phases and the drawback of strictly periodic schedule between actor phases. The first schedule that we consider is SPS. This schedule is visualized in Figure 2(a). Each execution of an actor is periodic, with the period computed by Equation (6) and the relative deadline equal to the period. Moreover, every execution phase of an actor is assumed to have the same WCET value. The throughput $\mathcal{R}$, latency $\mathcal{L}$ of G, and the required number of processors m are given in Table I under SPS.

However, by taking 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 IDP tasks considering different WCET values for each execution

phase, and execute them as periodic tasks. The execution schedule of this task set is given in Figure 2(b). Again, here we assume that data read and write times are 0. For example, actor τ_2 is converted to 2 IDP tasks $\tau_2(1)$ and $\tau_2(2)$, in which 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 I under ISPS (improved strictly periodic scheduling) that, by scheduling G this 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; thus, the processor resources are better utilized in the case of ISPS. This is especially important in the case of a timing constraint because it may happen that the graph cannot meet the constraint when scheduled under SPS. Here, the throughput and latency under ISPS are calculated by using our approach described in Section 5. The required number of processors for both SPS and ISPS is calculated by Equation (4). Moreover, the number of processors needed for partitioned scheduling in both cases is the same as the number needed for global scheduling given by Equation (4). We can see from the motivational example that the SPS approach from Bamakhrama and Stefanov [2013] yields to lower throughput and larger latency of a graph by using 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 our scheduling framework, called *improved strictly periodic scheduling* (ISPS), which enables a conversion of every actor of an acyclic CSDF graph to a set of periodic tasks. Each set of periodic tasks corresponding to an actor has as many elements as the number of phases of that actor. By taking into account the WCET value of each phase of an actor in a graph, the proposed approach computes the parameters S_i and T_i of tasks corresponding to the actor and the minimum buffer sizes of the communication channels such that ISPS is guaranteed to exist.

The proposed conversion procedure is given in Algorithm 1. First, the periods of tasks corresponding to actors are calculated in Lines 1 and 2, explained in Section 5.1. Then, relative deadlines D_i of the tasks corresponding to an actor τ_i are selected from the range $D_i \in [\max_{1 \le \varphi \le P_i}\{C_i(\varphi)\}, \check{T}_i]$, Lines 3 through 6. For example, to minimize the number of processors needed to schedule the converted tasks, one should select relative deadlines of the tasks to be equal to the corresponding task periods, that is, $D_i = \check{T}_i$. On the other hand, to reduce the graph latency, one should use our latency reduction method proposed in Section 5.5.2. The start times for each task set corresponding to an actor are computed in Lines 7 through 12; for details, see Section 5.2. Finally, the buffer sizes of the communication channels are derived in Lines 13 and 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 5.1. For each actor τ_i in an acyclic CSDF graph G, the **WCET sequence** $C_i = [C_i(1), C_i(2), \ldots, C_i(P_i)]$ 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

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 $\mathcal{T}_{\tau_i} = \{\tau_i(1), \ldots, \tau_i(P_i)\}$, and for each channel $e_u \in E$, the size of the buffer b_u.

```
for actor τ_i ∈ V do
    Compute the minimum common period Ť_i by using Equation (10);
for actor τ_i ∈ V do
    Select deadline D_i, where D_i ∈ [max_{1≤φ≤P_i}{C_i(φ)}, Ť_i];
    for phase φ of τ_i, 1 ≤ φ ≤ P_i do
        τ_i(φ) = (0, C_i(φ), D_i, Ť_i);
for actor τ_i ∈ V do
    Compute the start time of the first phase S_i(1) by using Equation (14);
    τ_i(1) = (S_i(1), C_i(1), D_i, Ť_i);
    for phase φ of τ_i, 2 ≤ φ ≤ P_i do
        Compute the start time of the φth phase S_i(φ) by using Equation (12);
        τ_i(φ) = (S_i(φ), C_i(φ), D_i, Ť_i);
for communication channel e_u ∈ E do
    Compute the buffer size b_u by using Equation (18);
```

given by

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

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 5.2. For each actor τ_i in an acyclic CSDF graph G, the **maximum WCET value** MC_i is given by $MC_i = \max_{1 \le \varphi \le P_i}\{C_i(\varphi)\}$.

Definition 5.3. For an acyclic CSDF graph G, an **aggregated execution vector** $\vec{AC}$, where $\vec{AC} \in \mathbb{N}^N$, represents the aggregated WCET values of the actors in G and its elements are given by $AC_i = \sum_{\varphi=1}^{P_i} C_i(\varphi)$, 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 $\mathcal{T}_{\tau_i} = \{\tau_i(1), \ldots, \tau_i(P_i)\}$.

Definition 5.4. A task $\tau_i(\varphi)$ corresponding to a phase φ of an actor τ_i, where $1 \le \varphi \le P_i$, in an acyclic CSDF graph G is a **strictly periodic task** if and only if 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 the *common period*.

Definition 5.5. For an acyclic CSDF graph G, a **common period vector** $\vec{T}$, where $\vec{T} \in \mathbb{N}^N$, represents the periods, measured in time units, of periodic task sets corresponding to actors in G. $T_i \in \vec{T}$ is common period of a periodic task set corresponding

to actor $\tau_i \in V$. $\vec{T}$ is given by the solution to both

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

and

$$\vec{T} - \vec{AC} \geq \vec{0}, \tag{9}$$

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

LEMMA 5.6. *For an acyclic CSDF graph G, the minimum common period vector* $\vec{\check{T}}$ *is given by*

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

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

PROOF. The proof can be found in Spasic et al. [2015]; see proof of Lemma 1. □

For the CSDF graph in Figure 1, the derived minimum common periods in time units are $[\check{T}_1, \check{T}_2, \check{T}_3] = [5, 10, 5]$.

THEOREM 5.7. *For any acyclic CSDF graph G, for which G has L topological sort levels, a periodic schedule exists with start times* $S_i(\varphi)$, $\varphi \in [1, P_i]$, *for each level-k actor* $\tau_i \in V$ *given by*

$$S_i(1) = (k-1) \cdot 2\alpha \tag{11}$$

and

$$S_i(\phi) = S_i(\phi - 1) + C_i(\phi - 1), \ \forall\ \phi \in [2, P_i], \tag{12}$$

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

$$b_u = (l - k + 1) \cdot 2X_i^u(P_i r_i), \tag{13}$$

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

PROOF. The proof can be found in Spasic et al. [2015]; see proof of Theorem 2. □

5.2. Deriving the Earliest Start Time of Actor's First Phase

In order to represent an actor of a CSDF graph as a set of strictly periodic tasks, in Theorem 5.7, we already introduced the start times of phases of the actors corresponding to different levels. However, although start times given by Equation (12) are minimal relative to the start time of the corresponding first phase $S_i(1)$, start times $S_i(1)$ given by Equation (11) are not minimal. 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 (minimal) start times of an actor's first phase $S_i(1)$ are derived here.

LEMMA 5.8. *For an acyclic CSDF graph G, the earliest start time of the first phase of an actor* $\tau_j \in V$, *denoted* $S_j(1)$, *under ISPS is given by*

$$S_j(1) = \begin{cases} 0 & \text{if } prec(\tau_j) = \emptyset \\ \max_{\tau_i \in prec(\tau_j)}\{S_{i \to j}(1)\} & \text{if } prec(\tau_j) \neq \emptyset \end{cases}, \tag{14}$$

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

$$S_{i\to j}(1) = \min_{t\in[0,S_i(1)+\alpha+\Delta_i(P_i)]} \left\{ t : \mathop{\mathsf{prd}^{\mathsf{S}}}_{[S_i(1),\max\{S_i(1),t\}+k)}(\tau_i,e_u) \geq \mathop{\mathsf{cns}^{\mathsf{S}}}_{[t,\max\{S_i(1),t\}+k]}(\tau_j,e_u), \forall k\in[0,\alpha+\Delta_i(P_i)] \right\}, \tag{15}$$

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

Proof. The proof can be found in Spasic et al. [2015]; see proof of Lemma 2. □

NOTE. We derive the earliest start times assuming that token production happens as late as possible (at the deadlines) and token consumption happens as early as possible (at the beginning of the execution of each phase). The cumulative production and the cumulative consumption functions can be computed efficiently by

$$\mathop{\mathsf{prd}^{\mathsf{S}}}_{[t_s,t_e)}(\tau_i,e_u) = \begin{cases} X_i^u\left(\left(\left\lfloor \frac{t_e-t_s}{T_i}\right\rfloor - 1 + \left\lfloor \frac{\Delta}{T_i}\right\rfloor\right)\cdot P_i + k_1\right) & \text{if } t_e - t_s \geq T_i \\ X_i^u(k_2) & \text{if } D_i \leq t_e - t_s \leq T_i \\ 0 & \text{if } t_e - t_s < D_i \end{cases}, \tag{16}$$

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

$$\mathop{\mathsf{cns}^{\mathsf{S}}}_{[t_s,t_e]}(\tau_i,e_u) = \begin{cases} Y_i^u\left(\left\lfloor \frac{t_e-t_s}{T_i}\right\rfloor + k\right) & \text{if } t_e \geq t_s \\ 0 & \text{if } t_e < t_s \end{cases}, \tag{17}$$

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

For example, the derived earliest start times for phases of actor 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

Equation (13) in Theorem 5.7 shows that ISPS has bounded buffer sizes b_u. These buffer sizes b_u are sufficient, but not minimal. Therefore, we want to derive the minimum buffer sizes that guarantee periodic execution of tasks corresponding to actor phases.

Lemma 5.9. *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*

$$b_u = \max_{k\in[0,\alpha+\Delta_j(P_j)]} \left\{ \mathop{\mathsf{prd}^{\mathsf{B}}}_{[S_i(1),\max\{S_i(1),S_j(1)\}+k]}(\tau_i,e_u) - \mathop{\mathsf{cns}^{\mathsf{B}}}_{[S_j(1),\max\{S_i(1),S_j(1)\}+k)}(\tau_j,e_u) \right\}, \tag{18}$$

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

Proof. The proof can be found in Spasic et al. [2015]; see proof of Lemma 3. □

NOTE. 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 token production happens as early as possible (at the beginning of the execution of each phase) and token consumption happens as late as possible (at the deadlines). The corresponding cumulative production and consumption functions can be computed efficiently by

$$\underset{[t_s,t_e)}{\mathsf{prd}^{\mathsf{B}}}(\tau_i, e_u) = \begin{cases} X_i^u\left(\left\lfloor \frac{t_e - t_s}{T_i} \right\rfloor + k\right) & \text{if } t_e \geq t_s \\ 0 & \text{if } t_e < t_s \end{cases}, \tag{19}$$

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

$$\underset{[t_s,t_e)}{\mathsf{cns}^{\mathsf{B}}}(\tau_i, e_u) = \begin{cases} Y_i^u\left(\left(\left\lfloor \frac{t_e - t_s}{T_i} \right\rfloor - 1 + \left\lfloor \frac{\Delta}{T_i} \right\rfloor\right) \cdot P_i + k_1\right) & \text{if } t_e - t_s \geq T_i \\ Y_i^u(k_2) & \text{if } D_i \leq t_e - t_s \leq T_i \\ 0 & \text{if } t_e - t_s < D_i \end{cases}, \tag{20}$$

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

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

5.4. Hard Real-Time Schedulability

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

THEOREM 5.10. *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*

$$\tau_i(\varphi) = (S_i(\varphi), C_i(\varphi), D_i, T_i),\ 1 \leq \varphi \leq P_i, \tag{21}$$

where $S_i(\varphi)$ is the earliest start time of a phase φ of actor τ_i given by Equations (14) and (12), $C_i(\varphi)$ is the WCET value of a phase φ given by Equation (7), D_i is the relative deadline, $\max_{1 \leq \varphi \leq P_i}\{C_i(\varphi)\} \leq D_i \leq T_i$, and T_i is the period of $\mathcal{T}_{\tau_i}$ given by Equation (10). $\mathcal{T}_G$ is schedulable on m processors using a hard real-time scheduling algorithm A for periodic tasks if

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

PROOF. According to Theorem 5.7, the graph is converted into strictly periodic tasks. The task set $\mathcal{T}_{\tau_i}$ corresponding to an actor τ_i, should be scheduled in a way that preserves the dependency between the actor phases. The hard real-time scheduling algorithms that can do this are partitioned Earliest Deadline First (EDF), Rate Monotonic (RM) [Liu and Layland 1973] and Deadline Monotonic (DM) [Leung and Whitehead 1982], or hierarchical [Holman and Anderson 2006; Lipari and Bini 2003]. In the case of the partitioned algorithms, tasks that correspond to phases of an actor should be allocated to the same processor and scheduled by EDF or DM because the deadlines of the phases are in the same order as the phases themselves, thereby preserving the

data dependencies between the phases, or by RM fixed priority scheduler, in which ties should be broken in favor of jobs arriving earlier in a system. In hierarchical scheduling, a set of tasks are grouped together and scheduled as a single entity, called *server task* or *supertask*. When the entity is scheduled, one of its tasks is selected to execute according to an internal scheduling policy. Hence, the supertasks/servers are scheduled globally, while the scheduling of the tasks within a supertask/server is done locally, that is, it is analogous to scheduling on a uniprocessor. By grouping the tasks that correspond to phases of an actor with data-dependent phases into a supertask/server and scheduling them by a scheduler that preserves their order (e.g., EDF) the synchronization problem of such dependent tasks is solved. □

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, that is, for analysis of the graph's throughput and latency.

5.5.1. Throughput Analysis Under ISPS. The throughput of a graph G scheduled by ISPS is given by

$$\mathcal{R}(G) = \frac{1}{\alpha} = \frac{1}{r_i \check{T}_i}, \tau_i \in V, \tag{22}$$

where $\check{T}_i$ is calculated by Equation (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

$$\mathcal{R}_i = \frac{q_i}{\alpha} = \frac{P_i}{\check{T}_i}, \tau_i \in V. \tag{23}$$

THEOREM 5.11. *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 $1/\alpha^{\text{SPS}} = 1/(q_i T_i^{\text{SPS}})$, $\tau_i \in V$. If the same graph is scheduled under our ISPS, then its throughput is $1/\alpha^{\text{ISPS}} = 1/(r_i T_i^{\text{ISPS}})$, $\tau_i \in V$. By using Equations (6) and (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 that we want to prove, that is, $\alpha^{\text{ISPS}} \le \alpha^{\text{SPS}}$, as follows:

$$\text{lcm}(\vec{r})\left\lceil \frac{u}{\text{lcm}(\vec{r})} \right\rceil \le \text{lcm}(\vec{q})\left\lceil \frac{w}{\text{lcm}(\vec{q})} \right\rceil. \tag{24}$$

We have that $u \le w$. Given that the least common multiple of positive integer numbers can be found using prime factorization, and the relation between vectors $\vec{r} = [r_1, \ldots, r_N]^T$ and $\vec{q} = \mathbf{P} \cdot \vec{r} = [P_1 r_1, \ldots, P_N r_N]^T$, we have that $\text{lcm}(\vec{q})$ is divisible by $\text{lcm}(\vec{r})$.

Finally, to prove the relation in Equation (24), we consider the following cases (with regard to divisibility by the corresponding *lcm* term): ***Case 1***: workloads on both sides of Inequality (24) 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 $\text{lcm}(\vec{r})$. We can represent the ceiling operation on the right-hand side as $(w + \text{lcm}(\vec{q}) - w \bmod (\text{lcm}(\vec{q})))/\text{lcm}(\vec{q})$. In the worst case, $w \bmod (\text{lcm}(\vec{q}))$ is equal to $\text{lcm}(\vec{q}) - 1$. By putting this into Inequality (24), we obtain $u \le w+1$, which holds. ***Case 3***: w is divisible by $\text{lcm}(\vec{q})$ (also divisible by $\text{lcm}(\vec{r})$). We can represent u and w as $k_u \text{lcm}(\vec{r}) + u \bmod (\text{lcm}(\vec{r}))$ and $k_w \text{lcm}(\vec{r})$, respectively, for some integer constants k_u and k_w, $k_u < k_w$. We represent the ceiling operation as in *Case 2*; thus, Inequality (24) becomes $u + \text{lcm}(\vec{r}) - u \bmod$

$(\text{lcm}(\vec{r})) \leq w$. Now, by putting the k_u-representation of u and k_w-representation of w, the inequality becomes $k_u + 1 \leq k_w$, which is true; thus, Inequality (24) holds. ***Case 4***: workloads on both sides of Inequality (24) are not divisible by the corresponding *lcm* terms. Similar 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, $w + 1$, which means that this value now is divisible by both $\text{lcm}(\vec{q})$ and $\text{lcm}(\vec{r})$. In the worst case, $u = w$, which means that u also needs only 1 unit to be rounded up to a value divisible by $\text{lcm}(\vec{r})$. Thus, Inequality (24) becomes $w + 1 \leq w + 1$, which holds. □

5.5.2. Latency Analysis Under ISPS. The latency of G scheduled by ISPS is given by

$$\mathcal{L}(G) = \max_{w_{\text{in}\rightarrow\text{out}} \in \mathcal{W}} \left\{ S_{\text{out}}(g^C_{\text{out}}) + D_{\text{out}} - S_{\text{in}}(g^P_{\text{in}}) \right\}, \tag{25}$$

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

From Equation (25), we can see that the latency of a graph depends on start times and deadlines of the graph's actors. Given that actor start times are dependent on deadlines (see Section 5.2), in order to reduce the latency, we should reduce actor deadlines, that is, we should change the token production times. However, given that reducing the deadlines increases the number of processors required to schedule the graph, we are interested in selecting the deadlines that lead to required graph latency while the number of processors needed to obtain that latency is minimized. To select deadlines properly, we devise the solution approach presented in this section that formulates the problem of selecting task deadlines under a given latency constraint while the number of processors is minimized when a CSDF graph is converted to real-time periodic tasks by using our ISPS approach as a mathematical programming problem. In order to formulate our problem as a mathematical programming problem, we need to rewrite the start-time computation in a proper form.

LEMMA 5.12. *For an acyclic CSDF graph G, the earliest start time of the first phase of an actor $\tau_j \in V$, denoted $S_j(1)$, under ISPS is given by*

$$S_j(1) = \begin{cases} 0 & \text{if } prec(\tau_j) = \emptyset \\ \max_{\tau_i \in prec(\tau_j)} \{S_i(1) + (S^{min}_{i\rightarrow j}(1) - S^{min}_i(1) - MC_i) + D_i\} & \text{if } prec(\tau_j) \neq \emptyset \end{cases}, \tag{26}$$

where $prec(\tau_j)$ is the set of predecessors of τ_j; and $S_i(1)$, MC_i, and D_i are the earliest start time of the first phase, the maximum WCET (Definition 5.2), and deadline of the predecessor actor τ_i, respectively. $S^{min}_i(1)$ is the earliest start time of the first phase of τ_i given by Equation (14) when $D_k = MC_k, \forall \tau_k \in V$, and $S^{min}_{i\rightarrow j}(1)$ is given by Equation (15) when $D_k = MC_k, \forall \tau_k \in V$.

PROOF. Let us consider an arbitrary channel $e_u = (\tau_i, \tau_j)$ in a CSDF graph $G = (V, E)$. Actor τ_j starts execution of its first phase after τ_i has started and fired a certain number of times. This number of firings is independent from the execution speed of the actors and depends only on the production and consumption rates of τ_i and τ_j on e_u, where cumulative production and cumulative consumption functions are given by Equations (16) and (17). Suppose that $D_k = MC_k, \forall \tau_k \in V$. The production ($\text{prd}^S$) and consumption ($\text{cns}^S$) curves of τ_i and τ_j are shown in Figure 3. Interval Δ in Figure 3 can be calculated as

$$\Delta = S^{\min}_{i\rightarrow j}(1) - S^{\min}_i(1) - MC_i. \tag{27}$$

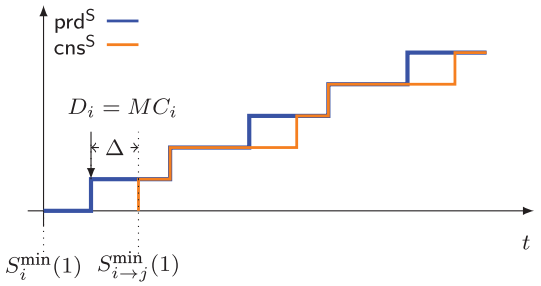

Fig. 3. Production and consumption curves on edge $e_u = (\tau_i, \tau_j)$.

Now, suppose that $D_k > MC_k, \forall \tau_k \in V$. The production curve will move to the right for certain time units, and the new start time of the first phase of τ_i is $S_i(1)$. If the consumption curve does not move, the relation between production and consumption given by Equation (15) will be violated, that is, it will happen in some point in time that cumulative consumption is greater than cumulative production. This means that we have to move the consumption curve to the right by the same number of time units such that the new start time $S_{i\to j}(1)$ satisfies Equation (15). Hence, interval Δ will stay the same, and it is given by

$$\Delta = S_{i\to j}(1) - S_i(1) - D_i. \tag{28}$$

By rewriting Equations (27) and (28), we obtain

$$S_{i\to j}(1) = S_i(1) + (S_{i\to j}^{\min}(1) - S_i^{\min}(1) - MC_i) + D_i. \quad \square \tag{29}$$

We can derive from Equation (26) the following set of linear inequality constraints, in which the number of linear inequality constraints is equal to the number of edges in the CSDF:

$$S_i(1) + \left(S_{i\to j}^{\min}(1) - S_i^{\min}(1) - MC_i\right) + D_i \le S_j(1), \forall e_u \in E. \tag{30}$$

In addition, we can rewrite Equation (25) as follows:

$$\mathcal{L}(G) = \max_{w_{\text{in}\to\text{out}} \in \mathcal{W}} \left\{ S_{\text{out}}(1) + \sum_{k=1}^{g_{\text{out}}^C - 1} C_{\text{out}}(k) + D_{\text{out}} - S_{\text{in}}(1) - \sum_{k=1}^{g_{\text{in}}^P - 1} C_{\text{in}}(k) \right\}. \tag{31}$$

Since the number of processors needed to schedule CDP tasks depends on the total density δ_{sum} of the tasks [Davis and Burns 2011], our objective is to minimize δ_{sum} in order to minimize the number of processors. Therefore, we formulate our optimization problem as follows:

$$\text{Minimize} \quad \delta_{\text{sum}} = \sum_{\tau_k \in V} \frac{AC_k}{D_k} \tag{32a}$$

$$\text{subject to} \quad S_{\text{out}}(1) + D_{\text{out}} - S_{\text{in}}(1) \le \mathcal{L} - \sum_{k=1}^{g_{\text{out}}^C - 1} C_{\text{out}}(k) + \sum_{k=1}^{g_{\text{in}}^P - 1} C_{\text{in}}(k), \quad \forall w_{\text{in}\to\text{out}} \in \mathcal{W} \tag{32b}$$

$$S_i(1) + D_i - S_j(1) \le -\left(S_{i\to j}^{\min}(1) - S_i^{\min}(1) - MC_i\right), \quad \forall e_u \in E \tag{32c}$$

$$-D_k \le -MC_k, D_k \le T_k, \quad \forall \tau_k \in V, \tag{32d}$$

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_{PAR}, task allocation *alloc*.

```
for actor τ_i in V do
    Compute the minimum common period T̆_i by using Equation (10);
u_total = 0;
U ← ∅; (the set of allocation units, initially empty)
for actor τ_i ∈ V do
    u_i = 0;
    for phase φ of τ_i, 1 ≤ φ ≤ P_i do
        u_i(φ) = C_i(φ) / T̆_i;
        u_i = u_i + u_i(φ);
        u_total = u_total + u_i(φ);
    U = U ∪ u_i;
m_PAR = m_OPT = ⌈u_total⌉;
Reorder elements of U if required by an allocation heuristic H;
for u ∈ U do
    Π = {π_1, π_2, ..., π_{m_PAR}};
    Apply bin-packing allocation heuristic H to u on π_j ∈ Π and check the schedulability
    test of algorithm A on π_j;
    if u is not allocated to any π_j ∈ Π then
        Allocate u on a new processor π_{m_PAR+1};
        m_PAR = m_PAR + 1;
return m_PAR, alloc;
```

where Equation (32a) is the objective function and D_k is an optimization variable. The objective function Equation (32a) has $|V|$ optimization variables and is subject to a latency constraint $\mathcal{L}$. Therefore, Equation (32b) comes from Equation (31). For each channel in a graph, we have Equation (30), which can be rewritten as Equation (32c). In addition, Equation (32d) bounds all optimization variables in the objective function. $S_i(1)$ and $S_j(1)$ (including $S_{\text{in}}(1)$, $S_{\text{out}}(1)$) are implicit variables that are not in the objective function Equation (32a), but still need to be considered in the optimization procedure. $\mathcal{L}$, g^P_{in}, g^C_{out}, $S^{\min}_{i\to j}(1)$, $S^{\min}_i(1)$, MC_k, and T_k are constants. Given that all variables are integers and both the objective function and the constraints are convex, problem Equation (32) is an integer convex programming (ICP) problem [Liu et al. 2014], which can be solved by using existing convex programming solvers, for example, CVX solver [Grant and Boyd 2014].

5.6. Deriving the Number of Processors

As introduced in Section 3.2, by using Equation (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; thus, they require task migration. On the other hand, the partitioned scheduling algorithms do not require task migration. In that case, the tasks are first allocated to the processors; 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, and so on [Coffman, Jr. et al. 1996], 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 (see Algorithm 1) is given in Algorithm 2. Algorithm 2 takes as input a CSDF graph G, a partitioned scheduling algorithm A, and an allocation heuristic H. The minimum common period for each actor is calculated in Lines 1 and 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 through 10. Line 11 in Algorithm 2 ensures that the task set corresponding to an actor is considered as one scheduling entity, that is, one allocation unit. The absolute minimum number of processors m_{OPT} for scheduling the tasks is computed in Line 12. Some allocation heuristics require a preprocessing step to be performed on the tasks before applying the heuristic. This preprocessing step is usually sorting the tasks based on some criteria, such as their utilization. That step is done in Algorithm 2 in Line 13. The following lines find the number of processors and the allocation of tasks to processors. Given that m_{OPT} is the lower bound on the number of processors m_{PAR} needed by partitioned scheduling algorithms, Algorithm 2 starts with the task partitioning on m_{OPT} processors. If the tasks pass the schedulability test on all m_{PAR} processors—for example, in the case of IDP tasks and EDF scheduler, the utilization of the tasks allocated to a processor is not greater than 1—then, the algorithm returns m_{PAR} and the corresponding allocation of the tasks to the processors *alloc*.

Let us now analyze the time complexity of Algorithm 2 in the worst case. The first **for loop** in Lines 1 and 2 takes linear time to calculate the minimum common period of each actor, that is, its time complexity is $O(|V|)$. The second **for loop** in Lines 5 through 11 has a nested for loop; thus, its time complexity in the worst case is given by $O(|V|P)$, where P is the maximum number of execution phases per actor, that is, $P = \max_{\tau_i \in V}\{P_i\}$. If the task sorting in Line 13 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 14 through 19 implements the allocation of the tasks to the processors by applying a particular 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 a 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 $O(|V|P \log |V|)$ [Pereira Zapata and Mejía Alvarez 2004; Baruah and Fisher 2005]. Thus, we can conclude that the runtime 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 II. Columns 3, 4, and 5 in Table II 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 WCETs of actors in the benchmarks are given in clock cycles [Bodin et al. 2013] or in time units [Benazouz et al. 2010; Wiggers et al. 2007]. If the execution times of a benchmark are not given [Bilsen et al. 1996; Pellizzoni et al. 2009; Oh and Ha 2004], certain values based on a static analysis are assumed. The execution times of the benchmark [Zitnick and Kanade 2000] are obtained from the measurements of the benchmark running on a MicroBlaze processor.

Table II. Benchmarks Used for Evaluation

Domain	Benchmark	$\|V\|$	$\|E\|$	$\|\mathcal{T}\|$	Source
Medical	Heart pacemaker	4	3	67	[Pellizzoni et al. 2009]
Communication	Reed Solomon Decoder (RSD)	6	6	904	[Benazouz et al. 2010]
Financial	BlackScholes	41	40	261	[Bodin et al. 2013]
Computer Vision	Disparity map	5	6	11	[Zitnick and Kanade 2000]
	Pdetect	58	76	4045	[Bodin et al. 2013]
Audio processing	CELP algorithm	9	10	167	[Bilsen et al. 1996]
	CD2DAT rate converter	6	5	22	[Oh and Ha 2004]
	MP3 Playback	4	3	8	[Wiggers et al. 2007]
Image processing	JPEG2000	240	703	639	[Bodin et al. 2013]

Our approach is evaluated by comparison to 3 related scheduling approaches—*strictly periodic scheduling* (SPS), proposed in Bamakhrama and Stefanov [2013]; *periodic scheduling* (PS), presented in Bodin et al. [2013]; and *self-timed scheduling* (STS), given in Stuijk et al. [2008]. We implemented our approach in Python. The SPS approach was implemented in Python within the *darts* tool set [Bamakhrama 2012]. The approach in Stuijk et al. [2008] was implemented in C++ within the SDF^3 tool set [Stuijk et al. 2006]. In addition, we implemented the approach in Bodin et al. [2013] in Python as well. We formulated both LP problems [Bodin et al. 2013] 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 [IBM 2012] to solve the ILP problems and mixed-integer disciplined convex programming (MIDCP) in CVX [Grant and Boyd 2014] to solve our latency reduction problem. We ran all the experiments on a Dell PowerEdge T710 server running Ubuntu 11.04 (64b) 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 [Bamakhrama and Stefanov 2013], PS [Bodin et al. 2013], and STS [Stuijk et al. 2008]. In addition, we compare our ISPS and the other scheduling approaches in terms of application latency and memory resources needed to implement the communication channels.

We used the `sdf3analysis-csdf` tool from SDF^3 [Stuijk et al. 2006] to obtain the maximum achievable throughput of a graph, which is the throughput under STS, and to compute the minimum buffer sizes required to achieve that throughput. Unfortunately, the `sdf3analysis-csdf` tool does not support the latency calculation and the calculation of the number of processors. Thus, we were not able to compare them with our approach. We were also not able to obtain the number of processors for a graph scheduled under PS, because the calculation of the number of processors was not considered in Bodin et al. [2013].

Results of the performance evaluation are given in Table III. We report the throughput of the output actors under ISPS, calculated by Equation (23), in the second column of Table III. 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, and PS and STS, respectively. Given that the main objective of this experiment is to evaluate the throughput of the benchmarks scheduled under ISPS and the minimum number of processors needed to obtain that throughput, our ISPS approach converts the CSDF graphs of the benchmarks to IDP tasks, which minimizes

Table III. 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	4	3860	1	1.84	4	4	1.48	1	2.02	1.76	0.91	1.66
JPEG2000	1/811008 (1/14598144)	27255343 (497471535)	18 (1)	18 (1)	9625878 (10006530)	70.65 (3.93)	0.02 (0.3)	1	1	1.17 (1.21)	1	–	–	N/A	N/A

the number of processors required to schedule the benchmarks. For processor requirements in the case of ISPS and SPS, we compute the minimum number of processors for IDP tasks under optimal and partitioned First-Fit Decreasing (Utilization) EDF (FFD-EDF) schedulers by using Equation (4) and Algorithm 2 for ISPS, and Equations (4) and (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 are the throughputs the same for both schedules. The first case is MP3 Playback, in which the 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; thus, the influence of a different WCET for actor phases on throughput cannot be seen. However, the influence can be seen from the required number of processors needed for scheduling of MP3 Playback by optimal schedulers, which is smaller in the case of our ISPS. The second case is CD2DAT. For this benchmark, lcm($\vec{q}$) and lcm($\vec{r}$) are equal and much higher than the maximum workload of actors over an iteration period for both SPS and ISPS, which leads to the same iteration period for both schedules. However, the WCET awareness of ISPS leads to a smaller number of processors. Note that if we want to schedule a task set on a smaller number of processors than the one calculated by Equation (4) or Equation (5)/Algorithm 2, we should scale up the computed actor periods by the same scaling factor [Zhai et al. 2013]. 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 a 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 a 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 [Zhai et al. 2013] 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, respectively, as given in Column 7 in parentheses, than the corresponding values under SPS. Therefore, we can conclude that, in all cases, the minimum number of processors required to guarantee certain throughput under our ISPS is smaller than or equal to the minimum number of processors under SPS while the throughput under ISPS is increased in most cases; thus, processors are better utilized.

Column 12 in Table III 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; thus, the scheduling granularity is similar between these two approaches.

Table III shows, in Column 15, the ratio of the maximum throughput of the output actors achieved by our approach to the absolute maximum throughput of the output actors achieved by self-timed scheduling of actor firings, which is the optimal scheduling in terms of throughput. We can see that the throughput under ISPS is equal to or very close to the throughput under STS for the majority of the benchmarks. Differences in the throughput appear as a result of the ceiling operation during the calculation of actor common periods in Equation (10). The biggest difference is in the case of the CD2DAT benchmark. For this benchmark, lcm($\vec{r}$) is much higher than the maximum workload of actors over an iteration period; 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 [Stuijk et al. 2006] returned an infeasible throughput (most likely related to an integer overflow).

Let us now analyze the latency and the memory resources needed to implement the communication channels of the benchmarks. The graph latency under our ISPS is calculated by Equation (25) for IDP tasks and shown in Column 3 of Table III. Column 8 shows the ratio between the graph latency under our ISPS and SPS. As we can see from Columns 4, 5, and 7 through 10 in Table III: 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 the 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 other 2 benchmarks, we obtained the same throughput with an increase in latency, but also with a 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 III). 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 215KB and less than 1KB, respectively, which is acceptable given the size of the memory available in modern embedded systems. Note that both latency and buffer sizes under our ISPS can be reduced by carefully selecting deadlines for individual actors (actor phases). This will be shown in Section 6.3.

Column 13 gives the ratio of the maximum latency of benchmarks under our ISPS to the latency of benchmarks under PS. Although Bodin et al. [2013] do not provide the latency calculation for their PS, we were able to extract the latency information from the start times obtained by solving the ILP problem. However, for benchmarks JPEG2000 and Pdetect, we could not get a solution from the ILP solver after more than 1 day; thus, we could not calculate the latency for these two benchmarks. As we can see, the latency of benchmarks under ISPS is always larger than the latency under PS. As mentioned earlier, reducing the latency under ISPS can be done by carefully selecting deadlines for individual actors (actor phases), as shown in Section 6.3. Moreover, ISPS reports the maximum latency while PS reports the actual latency under a certain schedule. The ratio of the calculated buffer sizes under ISPS to the calculated buffer sizes under PS and STS is given in Columns 14 and 16, respectively. 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 the SDF^3 tool set returned an infeasible throughput; thus, 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 232KB and less than 1KB, respectively. The highest increase in buffer sizes under ISPS when compared to STS is less than 1KB. The reason for the difference in the buffer sizes is that, in both PS and STS approaches, it is assumed that the production of tokens happens at the end of the actor firing and the consumption happens at the start of the firing, while in our case (and in SPS case) the worst-case scenario is considered, that is, the production of tokens happens at the earliest possible start of the actor firing (at start times), while the consumption happens at the latest possible end of the 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 when reading and writing of tokens happens. 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 IV. We compare these execution times with the corresponding execution times of related approaches—SPS, PS, and STS.

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

Next, we compare the time needed to derive the start times of actor firings, that is, the schedule, and the buffer sizes of communication channels. These times are given in Columns 3, 5, 7, and 9, for ISPS, SPS, PS, and STS, respectively. By comparing the times under ISPS and SPS, we can see that both approaches find the start times and the buffer sizes within less than 4s in most cases, and within 1min 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; thus, they can be solved very fast. As shown in Table IV, ISPS gives a solution for those two cases within 1s, and within 1min. On the other hand, for benchmarks Pdetect and JPEG2000, we could not get a solution from the ILP solver for PS after more than 1d, while our ISPS produced the results in a couple of seconds and within 1min. 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 1d.

In Table V, we report 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 Equation (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 1ms for most of the benchmarks, while in the case of partitioned scheduling, the calculation is done within less than 12ms for most cases and within less than 420ms 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 regard to the

Table IV. Time Complexity (In Seconds) of Different Scheduling Approaches

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

Table V. Time Complexity (In Seconds) for the Calculation of Number of Processors

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

PS approach because the calculation of the number of processors was not considered in Bodin et al. [2013]. 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 that delivers the maximum achievable throughput. The SDF^3 tool set used to compute the self-timed scheduling parameters does not support such DSE for self-timed scheduling. Thus, we could not numerically compare the time complexity of ISPS with the time complexity of STS. However, given that ISPS finds the minimum number of processors for scheduling an application in polynomial time in the worst case, as shown in Section 5.6, we can conclude that our ISPS is faster than STS.

6.3. Reducing Latency Under ISPS

We have shown in the previous experiments that, when compared to the SPS approach, our ISPS delivers larger graph latency in 5 out of 9 cases. When compared to the PS approach, our ISPS approach always results in a graph schedule with larger graph latency. If we want to reduce graph latency under ISPS, we could use the latency reduction method presented in Section 5.5.2. We would like to see how close we are in graph latency in comparison to the SPS and PS approaches after applying our latency reduction method. Therefore, in this section, we present results obtained after applying our latency reduction method introduced in Section 5.5.2 on the benchmarks given in

Table VI. Performance of the ISPS Approach under Different Latency Constraints

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

Table II. The results are given in Table VI. In order to apply our latency reduction method, we should set a latency constraint. To compare our ISPS approach to the SPS approach, we set the latency constraint to be equal to the graph latency obtained under SPS, $\mathcal{L}^{\text{SPS}}$, and we apply our method for latency reduction. We can see from Column 3 in Table VI that we significantly reduce latency for the benchmarks that had higher latency under ISPS than SPS (see Column 8 in Table III), and that we were able to meet the latency constraint $\mathcal{L}^{\text{SPS}}$ for all benchmarks. Moreover, we see that reduction in graph latency does not influence graph throughput, that is, the ratio of graph throughput under ISPS to graph throughput under SPS in Column 2 is the same as the corresponding ratio given in Column 7 in Table III with the period scaling technique applied for benchmarks RSD and JPEG2000 under ISPS. Columns 4 through 6 give the results on resources in terms of the number of processors required by ISPS and SPS, and the ratio between the ISPS and SPS approaches in buffer sizes needed to implement communication channels in a graph. We find the minimum number of processors under the partitioned First-Fit Increasing Deadlines EDF (FFID-EDF) [Baruah and Fisher 2005] scheduler by using Algorithm 2 for ISPS and Equation (5) for SPS and the FFD-EDF scheduler. We can see from Columns 2 through 5 that our ISPS approach with our latency reduction method is able to schedule almost all benchmarks on the same number of processors as the SPS approach while obtaining better graph throughput and shorter graph latency. Only in one case, for benchmark BlackScholes, does our approach need one processor more than the SPS approach. However, our approach delivers better throughput for benchmark BlackScholes than the SPS. Although the ratio between the buffer sizes under ISPS and the buffer sizes under SPS, given in Column 6 in Table VI, is smaller than the corresponding ratio in Table III, Column 11, the buffer sizes under ISPS are still always bigger than the corresponding buffer sizes under SPS.

Columns 7 through 10 of Table VI give the results when our latency reduction method is applied with the latency constraint dictated by the PS approach, $\mathcal{L}^{\text{PS}}$. Since we could not obtain the solution from the ILP solver in the case of the PS approach after 1d for Pdetect and JPEG2000 benchmarks (see Table III), we could not provide latency and buffer sizes ratios for these two benchmarks. We can see from Column 8 in Table VI that we significantly reduce latency for all benchmarks (see Column 13 in Table III). However, in four cases—for benchmarks BlackScholes, CELP, CD2DAT, and MP3 playback—our latency reduction method was not able to meet the latency constraint $\mathcal{L}^{\text{PS}}$. The reason is that the PS approach gives the actual latency under a static schedule, while our ISPS approach calculates the maximum latency for a CSDF graph converted into real-time periodic tasks. For these 4 benchmarks, Column 8 gives the shortest achievable latency under ISPS obtained by applying our latency reduction

method. The ratio of the graph throughput under ISPS to the graph throughput under PS is given in Column 7, and is the same as the corresponding ratio given in Column 12 in Table III. We report in Column 9 the minimum number of processors under ISPS and FFID-EDF found by Algorithm 2. We can see that the number of processors needed by all the benchmarks with reduced latency under ISPS is higher than the corresponding number of processors given in Table III, Column 5, which is expected. The number of processors for a graph scheduled under PS is not given because the calculation of the number of processors was not considered in Bodin et al. [2013]. Although the ratio between the buffer sizes under ISPS and the buffer sizes under PS, given in Column 10 in Table VI, is smaller than the corresponding ratio in Table III, Column 14, the buffer sizes under ISPS are still always bigger than the buffer sizes under PS. As explained previously, the reason for the difference in the buffer sizes is that the PS approach considers a 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 their deadlines and for any production/consumption pattern during its firing.

We also measured the execution times of our ISPS approach enhanced with the latency reduction method to find tasks' deadlines and a schedule, that is, tasks' start times, such that the latency constraint is satisfied. In most cases, our latency reduction method needed less than 1s, and in three cases less than 1min, to find tasks' deadlines and a schedule that meets the latency constraint.

7. CONCLUSIONS

In this article, we presented a scheduling approach that converts each actor in a CSDF graph by considering different WCET values 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 a calculated number of processors with a certain guaranteed throughput and latency. In addition, we presented a method to reduce the graph latency when the converted tasks are scheduled as real-time periodic tasks. The experiments on a set of real-life applications showed that our ISPS approach gives a tighter guarantee on the throughput and better processor utilization with an acceptable increase in terms of communication memory requirements when compared with the SPS hard real-time scheduling approach. By applying our proposed latency reduction method, the ISPS delivers shorter graph latency while providing better throughput and processor utilization than the SPS approach. When compared with the PS approach, our proposed approach gives the same throughput with increased communication memory, but takes much shorter time for deriving the schedule, 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 to or very close to the absolute maximum throughput achieved by the STS of actor firings, but requires much shorter time to derive the schedule.

REFERENCES

M. Bamakhrama. 2012. Retrieved July 2, 2016 from http://daedalus.liacs.nl/darts.

M. Bamakhrama and T. Stefanov. 2013. On the hard-real-time scheduling of embedded streaming applications. *International Journal on Design Automation for Embedded Systems* 17, 2, 221–249.

S. Baruah and N. Fisher. 2005. The partitioned multiprocessor scheduling of sporadic task systems. In *Proceedings of RTSS*. 321–329.

S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. 1993. Proportionate progress: A notion of fairness in resource allocation. In *Proceedings of STOC*. 345–354.

M. Benazouz, O. Marchetti, A. Munier Kordon, and T. Michel. 2010. A new method for minimizing buffer sizes for cyclo-static dataflow graphs. In *Proceedings of ESTIMedia*. 11–20.

G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. 1996. Cyclo-static dataflow. *IEEE Transactions on Signal Processing* 44, 2, 397–408.

B. Bodin, A. Munier-Kordon, and B. D. de Dinechin. 2013. Periodic schedules for cyclo-static dataflow. In *Proceedings of ESTIMedia*. 105–114.

A. Bouakaz, J.-P. Talpin, and J. Vitek. 2012. Affine data-flow graphs for the synthesis of hard real-time applications. In *Proceedings of ACSD*. 183–192.

E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. 1996. Approximation algorithms for bin packing: A survey. In *Approximation Algorithms for NP-Hard Problems*. 46–93.

R. I. Davis and A. Burns. 2011. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys* 43, 4, 35.

M. Grant and S. Boyd. 2014. CVX:MATLAB software for disciplined convex programming, version 2.1. Retrieved from http://cvxr.com/cvx.

J. P. H. M. Hausmans, S. J. Geuns, M. H. Wiggers, and M. J. G. Bekooij. 2013. Two parameter workload characterization for improved dataflow analysis accuracy. In *Proceedings of RTAS*. 117–126.

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

IBM. 2012. IBM ILOG CPLEX Optimization Studio V12.4. Retrieved from https://www.ibm.com/developer works/downloads/ws/ilogcplex.

E. A. Lee and D. G. Messerschmitt. 1987. Synchronous data flow. *Proceedings of IEEE* 75, 9, 1235–1245.

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

G. Lipari and E. Bini. 2003. Resource partitioning among real-time applications. In *Proceedings of ECRTS*. 151–158.

C. L. Liu and J. W. Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM* 20, 1, 46–61.

D. Liu, J. Spasic, J. T. Zhai, T. Stefanov, and G. Chen. 2014. Resource optimization for CSDF-modeled streaming applications with latency constraints. In *Proceedings of DATE*. 188:1–188:6.

O. Moreira, J.-D. Mol, M. Bekooij, and J. van Meerbergen. 2005. Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix. In *Proceedings of RTAS*. 332–341.

H. Oh and S. Ha. 2004. Fractional rate dataflow model for efficient code synthesis. *Journal of VLSI Signal Processing* 37, 1, 41–51.

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

O. U. Pereira Zapata and P. Mejía Alvarez. 2004. *EDF and RM Multiprocessor Scheduling Algorithms: Survey and Performance Evaluation*. Technical Report CINVESTAV-CS-RTG-02.

J. Spasic, D. Liu, E. Cannella, and T. Stefanov. 2015. Improved hard real-time scheduling of CSDF-modeled streaming applications. In *Proceedings of CODES+ISSS*. 65–74.

S. Stuijk, M. Geilen, and T. Basten. 2006. SDF^3: SDF for free. In *Proceedings of ACSD*. 276–278.

S. Stuijk, M. Geilen, and T. Basten. 2008. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers* 57, 10, 1331–1345.

W. Thies and S. Amarasinghe. 2010. An empirical characterization of stream programs and its implications for language and compiler design. In *Proceedings of PACT*. 365–376.

M. Wiggers, M. Bekooij, P. G. Jansen, and G. J. M. Smit. 2007. Efficient computation of buffer capacities for cyclo-static real-time systems with back-pressure. In *Proceedings of RTAS*. 281–292.

J. T. Zhai, M. Bamakhrama, and T. Stefanov. 2013. Exploiting just-enough parallelism when mapping streaming applications in hard real-time systems. In *Proceedings of DAC*. 170:1–170:8.

C. L. Zitnick and T. Kanade. 2000. A cooperative algorithm for stereo matching and occlusion detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22, 7, 675–684.

Received December 2015; revised April 2016; accepted April 2016