Resource Optimization for Real-Time Streaming Applications using Task Replication

Sobhan Niknam, Peng Wang, and Todor Stefanov, *Member, IEEE*

***Abstract*—In this paper, we study the problem of exploiting parallelism in a hard real-time streaming application modeled as an acyclic Synchronous Data Flow (SDF) graph and scheduled on a heterogeneous Multi-Processor System-on-Chip (MPSoC) platform to alleviate the capacity fragmentation due to partitioned scheduling algorithms and reduce the number of required processors when a throughput requirement is satisfied. As the main contribution in this paper, we propose a method to determine a replication factor for each task in an acyclic SDF graph such that by distributing the workloads among more parallel tasks with lower utilization in the obtained transformed graph, the left capacity on the processors can be efficiently exploited, hence reducing the number of required processors. The experimental results, on a set of real-life streaming applications, demonstrate that our approach can reduce the minimum number of processors required to schedule an application and considerably improve the memory requirements and application latency compared to related approaches while meeting the same throughput constraint.**

I. Introduction

STREAMING applications is an important group of embedded software that spans different application domains such as image processing, video/audio processing, and digital signal processing. The ever-increasing computational demand and hard real-time constraints of these applications push the system designers toward using Multiprocessor System-on-Chip (MPSoCs) in modern embedded systems to benefit from parallel execution. Nowadays, heterogeneous MPSoCs are becoming increasingly common due to their capability to balance the performance and energy efficiency by employing relatively slower but low-power processors along with faster but high-power ones, e.g., ARM big.LITTLE [1]. To efficiently exploit the computational capacity of such MPSoCs, however, streaming applications must be expressed primarily in a parallel fashion. The common practice for expressing the parallelism in an application is to use parallel Models of Computation (MoCs). The main benefits of the MoCs are the explicit representation of important properties in the application, e.g., parallelism, and the enhanced design-time analyzability of certain system properties, e.g., throughput. Within a parallel MoC, a streaming application is represented as a task graph with concurrently executing and communicating tasks. Two well-known MoCs are Synchronous Dataflow (SDF) [2] and its generalization, Cyclo-Static Dataflow (CSDF) [3].

Although parallel MoCs resolve the problem of explicitly exposing the available parallelism in an application, the main challenge is then how to allocate and schedule the tasks of the application on an MPSoC such that all hard real-time constraints are guaranteed. To address this challenge, a large body of research exists in the classical real-time scheduling theory for scheduling different *real-time task models*, e.g., periodic and sporadic task models, on multiprocessors [4]. However, since these theories typically assume sets of independent tasks, they are not directly applicable to modern embedded streaming applications which have data-dependent tasks. Recently, a scheduling framework has been presented in [5] that shows how streaming applications modeled as acyclic (C)SDF graphs can be scheduled as a set of real-time implicit-deadline periodic tasks. This framework, thus, enables a designer to reuse many well-developed algorithms from the classical hard real-time multiprocessor scheduling theory to guarantee hard real-time constraints and temporal isolation among different concurrently running applications on a multiprocessor system, using fast schedulability analysis. Moreover, these algorithms provide fast analytical calculation of the minimum number of processors needed to schedule the tasks in an application. Therefore, because of the advantages of [5] over conventional static scheduling, we adopt [5] in this paper as a primary technique for scheduling streaming applications.

In real-time systems, tasks can be scheduled on multiprocessors using three main classes of algorithms, i.e., global, partitioned, and hybrid scheduling algorithms based on whether a task can migrate between processors [4]. Under global scheduling algorithms, all the tasks can migrate between all the processors. Such scheduling guarantees optimal utilization of the available processors but at the expense of high scheduling overheads due to extreme task preemptions and migrations. More importantly, implementing global scheduling algorithms in distributed-memory MPSoCs imposes a large memory overhead due to replicating the code of each task on every processor [6]. Under partitioned scheduling algorithms, however, no task migration is allowed and the tasks are allocated statically to the processors, hence they have low run-time overheads. The tasks on each processor are scheduled separately by a uniprocessor (hard) real-time scheduling algorithm, e.g., Earliest Deadline First (EDF) [7]. The third class of scheduling algorithms is hybrid scheduling that is a mix of global and partitioned approaches to take advantages of both classes. However, since hybrid scheduling

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis 2018 and appears as part of the ESWEEK-TCAD special issue.

The authors are with the Leiden Institute of Advanced Computer Science, Leiden University, Leiden, The Netherlands, Email: {s.niknam, p.wang, t.p.stefanov}@liacs.leidenuniv.nl. This research is supported by the Dutch Technology Foundation STW under the Robust Cyber Physical Systems program (Project 12695).

algorithms allow task migration, they still introduce additional run-time task migration/preemption overheads and memory overhead on distributed-memory MPSoCs. By performing an extensive empirical comparison of global, clustered (hybrid) and partitioned algorithms for EDF scheduling, the authors in [8] concluded that the partitioned algorithms outperform all the other algorithms when hard real-time systems are considered. Thus, in this paper, we consider partitioned scheduling algorithms.

Although partitioned scheduling algorithms do not impose any migration and memory overheads, they are known to be non-optimal for scheduling real-time tasks [4]. This is because the partitioned scheduling algorithms fragment the processors' computational capacity such that no single processor has sufficient remaining capacity to schedule any other task in spite of the existence of a total large amount of unused capacity on the platform. Therefore, more processors are needed to schedule a set of real-time tasks using partitioned scheduling algorithms compared to optimal (global) scheduling algorithms.

However, for better resource usage and energy efficiency in a real-time embedded system while taking advantages of partitioned scheduling algorithms, the number of processors needed to guarantee a performance constraint, i.e., throughput, in an application should be minimized. This can be difficult because often the given initial application task graph is not the most suitable one for the given MPSoC platform because the application developers typically focus on realizing certain application behavior while neglecting the efficient utilization of the available resources on MPSoC platforms. Therefore, to better utilize the resources on an underlying MPSoC platform while using partitioned scheduling algorithms, the initial application task graph should be transformed to an alternative one that exposes more parallelism while preserving the same application behavior and performance. This is mainly because by replicating a task, its workload is distributed among more parallel task's replicas in the obtained transformed graph. Therefore, the task's required capacity is split up in multiple smaller chunks that can more likely fit into the left capacity on the processors and alleviate the capacity fragmentation due to partitioned scheduling algorithms. However, having more parallelism, i.e., tasks' replicas, than necessary introduces significant overheads in code and data memory, scheduling and inter-tasks communication. Thus, the right amount of parallelism should be determined in a parallel application specification to achieve the required performance while minimizing the number of required processors.

Therefore, considering partitioned scheduling algorithms, in this paper, we address the problem of finding a proper replication factor for each task in an initial application graph, such that the obtained alternative one requires processors while guaranteeing a given throughput constraint. More specifically, the main novel contributions of this paper are summarized as follows:

- We propose a novel heuristic algorithm to allocate the tasks in a hard real-time streaming application modeled as an acyclic SDF graph, which is subject to a throughput constraint, onto a heterogeneous MPSoC such that the number of required processors is reduced under partitioned scheduling algorithms. The main innovation in this algorithm is that by using the unfolding graph transformation in [9], we propose a method to determine a replication factor for each task in the SDF graph such that the distribution of the workloads among more parallel tasks, in the obtained graph after the transformation, results in a better resource utilization, which can alleviate the capacity fragmentation introduced by partitioned scheduling algorithms, hence reducing the number of required processors.
- We show, on a set of real-life benchmarks, that our approach significantly reduces the number of required processors compared to the related approach in [10], called First-Fit Decreasing (FFD) allocation algorithm, with slightly increasing the memory requirements and application latency while maintaining the same application throughput. We also show that our approach can still reduce the number of required processors compared to the related approaches in [11], [12], [9], [13] with significantly improving the memory requirements and application latency while maintaining the same application throughput.

Scope of work. In this paper, we consider SDF graphs that are acyclic. This restriction comes from the adopted hard real-time scheduling framework [5] to schedule an SDF graph. Although this restriction may seem to limit the scope of our approach, our approach is still applicable to the majority of real-life streaming applications. This is because, the authors in [14] have shown that around 90% of streaming applications can be modeled as acyclic SDF graphs. In this paper, we also consider heterogeneous multi-processor systems with distributed program and data memory to ensure predictability of the execution at runtime and scalability. We assume that the communication infrastructure used for inter-processor communication is predictable, i.e., it provides guaranteed communication latency. We use the worst-case communication latency to compute the worst-case execution time of a task, which in our approach includes the worst-case time needed for the task's computation and the worst-case time needed to perform inter-task data communication on the considered platform. Finally, we adopt a partitioned scheduling algorithm, i.e., Partitioned EDF, in this paper. Partitioned EDF outperforms the global EDF scheduling algorithm for hard real-time task sets, as empirically studied and shown in [8].

Organization. The remainder of the paper is organized as follows: Section II gives an overview of the related work. Section III introduces the background material needed for understanding the contributions of this paper. Section IV gives a motivational example. Section V presents the proposed approach. Section VI presents the results of the evaluation of the proposed approach. Finally, Section VII ends the paper with conclusions.

II. Related Work

In order to overcome the scheduling problems in global and partitioned scheduling algorithms, a restricted-migration semi-partitioned scheduling algorithm, called EDF-fm, in the class of hybrid scheduling algorithms, is proposed in [11] for homogeneous platforms. In this scheduling algorithm, the

tasks can be either fixed or migrating between only two processors at job boundaries. The purpose of this migration is to utilize the remaining capacity on the processors where a migrating task cannot be entirely allocated. However, this scheduler provides hard real-time guarantees only for migrating tasks and soft real-time guarantees for fixed tasks, i.e., fixed tasks can miss their deadlines by a bounded value called tardiness. In [13], another semi-partitioned scheduling algorithm, called EDF-sh, is proposed that, in contrast to EDF-fm, supports heterogeneous platforms and allows the tasks to migrate between more than two processors. In EDF-sh, however, both migrating and fixed tasks may miss their deadlines.

Similarly, [15] proposes the C=D approach to split real-time tasks on homogeneous multiprocessor systems while on each processor a normal EDF scheduler is used. In this approach, if a task cannot be entirely allocated to a processor, the C=D approach splits the task into two parts. However, since the task splitting is performed in every job execution, this approach requires transferring the internal state of the splitted tasks between processors at run-time, thereby imposing high task migration overhead. Moreover, these approaches in [11], [13], [15] only consider sets of independent tasks. In contrast, in this paper, we consider a more realistic application model which consists of tasks with data dependencies. In addition, we use partitioned scheduling to allocate the tasks statically on the processors. Therefore, since task migration is not allowed in partitioned scheduling, no extra run-time overhead is imposed to the system by our approach in comparison to [15] and no task is subjected to a deadline miss in comparison to [11], [13]. Compared to the approaches in [11], [15] that only support homogeneous platforms, our proposed approach also supports heterogeneous platforms.

To allocate data-dependent application tasks to a multiprocessor platform, many techniques have already been devised [16]. Existing approaches which are close to our work are [5], [12], [9]. The authors in [5] propose a scheduling framework to only convert each task in an acyclic (C)SDF graph to an implicit-deadline periodic task by deriving parameters such as period and start time to enable the usage of all well-developed real-time theories. In [5], however, no optimization technique for different system design metrics, such as, throughput, latency, memory, number of processors, etc. is proposed. In contrast, in this paper, we propose a heuristic approach on top of the scheduling framework in [5] to optimize the number of required processors when scheduling a hard real-time streaming application with a throughput constraint onto a heterogeneous MPSoC under partitioned scheduling algorithms.

Using the framework in [5], the authors in [12] propose a heuristic under the semi-partitioned scheduling algorithm in [11] to allocate tasks to processors while taking the data dependencies into account. Although the fixed tasks can miss their deadlines in the EDF-fm scheduling approach, a hard real-time property can be guaranteed on the input/output interfaces of the application with the external environment, using the extension of the framework in [5] proposed in [12]. In [11], the authors also propose three task-allocation heuristics under EDF-fm to allocate independent tasks to processors in which the one called fm-LUF requires the least number of processors. In a similar way, this heuristic can be used while taking data dependencies into account using the approach presented in [12]. However, in these approaches [12], [11], the deadline misses of the fixed tasks due to task migration have significant overheads on the memory requirements and the application latency. In contrast, in this paper, we provide hard real-time guarantees for all tasks in an application modeled as an SDF graph. Moreover, we use partitioned scheduling and to utilize processors efficiently, we adopt the unfolding graph transformation technique. By using our proposed approach, as shown in Section VI, processors can be more efficiently utilized while imposing considerably lower overheads on the memory requirements and the application latency compared to the approaches in [12], [11]. In addition, our proposed approach supports heterogeneous platforms while the approaches in [11], [12] can only support homogeneous platforms.

In [9], the authors propose an approach to increase the application throughput in a homogeneous platform with a fixed number of processors. This approach considers partitioned scheduling and exploits an unfolding transformation to fully utilize the platform by replicating the bottleneck tasks which are the ones with the maximum workload, i.e., highest utilization, when mapping a streaming application modeled as an SDF. However, to guarantee a throughput constraint under limited resources, the approach in [9] does not always replicate the right tasks, as shown in Section IV. Consequently, this leads to more parallelism than needed which increases the memory requirements and application latency unnecessarily. In contrast, we propose an approach that supports heterogeneous platforms. In addition, our proposed approach first detects which tasks cause the capacity fragmentation in partitioned scheduling on the processors. Note that these tasks are not the bottleneck tasks identified and used in [9]. This is because, the bottleneck tasks efficiently utilize the processors' capacity and there is no need to replicate them. Then, using the unfolding transformation technique, we replicate the detected tasks causing the capacity fragmentation to distribute their workloads among more parallel tasks and utilize the platform more efficiently with less unused capacity on the processors. As a result, shown in Section VI, our proposed approach can reduce the number of required processors to guarantee the same throughput while keeping a low memory and latency overheads under partitioned scheduling in comparison to [9].

In [17], the authors use the same approach as in [9] for energy efficiency purpose under partitioned scheduling algorithms, when there are a lot of processors available on a cluster heterogeneous MPSoC. To reduce energy consumption, they iteratively take the bottleneck tasks which are limiting the processors to work at a lower frequency and replicate them. By replicating the application tasks with heavy utilization, their utilization is distributed among more task's replicas while still providing the same application performance. Consequently, the workload distribution of these bottleneck tasks enables the processors to work at a lower frequency, thereby reducing the energy consumption. In our paper, however, we focus on and solve a totally different problem, that is, how the unfold-

ing transformation technique can be exploited to reduce the number of required processors when a partitioned scheduling algorithm is used. In our approach, we do not search for and take the bottleneck task, which is taken in [17], for replication in every iteration. In contrast, we detect which task is responsible for fragmentation of the processors' capacity when using a partitioned scheduling algorithm and try to resolve this fragmentation by replicating this task such that the number of processors is reduced. We do not replicate the bottleneck task because it can efficiently utilize the processor and it does not contribute to the fragmentation of the processors' capacity.

III. Background

Given the fact that we use the unfolding transformation in [9] to replicate the tasks in an application modeled as an SDF graph, such transformation converts the initial graph into an equivalent CSDF graph. Therefore, because the CSDF MoC is a superset of the SDF MoC, in this section, we first introduce the CSDF MoC, followed by the unfolding transformation proposed in [9]. Then, we briefly introduce the scheduling framework proposed in [5], which we use to schedule tasks in a CSDF graph. After that, we present the system model considered in this paper.

A. Cyclo-Static Data Flow (CSDF)

An application modeled as a CSDF [3] graph is a directed graph $G = (V, E)$, where V is a set of tasks and E is a set of edges. Task $\tau_i \in V$ represents computation and edges represent the transfer of data tokens between tasks. Each task $\tau_i \in V$ has an *execution sequence* $[f_i(1),\ f_i(2),\ \cdots,\ f_i(P_i)]$ of length P_i, i.e, it has P_i phases. This means that the execution of each phase ϕ of task τ_i is associated with a certain function $f_i(\phi)$. Therefore, the k-th time the task τ_i is fired, the function according to the phase $(((k-1) \bmod P_i)+1)$ is being executed, i.e., $f_i(((k-1) \bmod P_i)+1)$. Consequently, the execution time and the data production/consumption rate for each output/input edge of task τ_i are also defined for each phase. Therefore, each task $\tau_i \in V$ has the following sequences of length P_i: a sequence of the *worst-case execution time* $[C_i(1), C_i(2), \cdots, C_i(P_i)]$, a predefined *data production sequence* of $[x_i^u(1), x_i^u(2), \cdots, x_i^u(P_i)]$ on its every output channel e_u, and a predefined *data consumption sequence* of $[y_i^u(1), y_i^u(2), \cdots, y_i^u(P_i)]$ on its every input channel e_u. If every task τ_i in a CSDF graph G has a single phase, i.e., $P_i = 1$, then the graph G is an SDF [2] graph that means the SDF MoC is a subset of the CSDF MoC.

An important property of the CSDF MoC that is proven in [3], is that a valid static schedule of a CSDF graph can be generated at design-time if the graph is consistent and live. A CSDF graph is said to be consistent if a non-trivial solution exists for the repetition vector $\vec{q} = [q_1, q_2, \cdots, q_n]^T \in \mathbb{N}^n$. An entry q_i indicates the number of invocations of task τ_i in one graph iteration of the CSDF graph. If a deadlock-free schedule can be found, G is then said to be live. Throughout this paper, we consider and use consistent and live SDF and CSDF graphs.

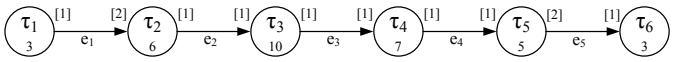
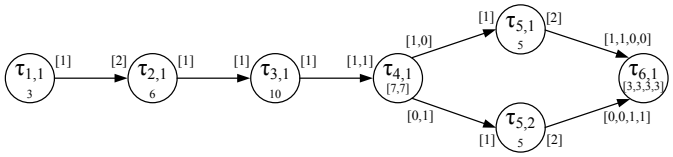
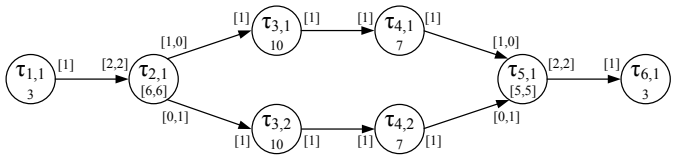


Fig. 1. An SDF graph G.



(a) A CSDF graph G'



(b) A CSDF graph G''

Fig. 2. Equivalent CSDF graphs of the SDF graph G in Figure 1 obtained by (a) replicating task τ_5 by factor 2 and (b) replicating tasks τ_3 and τ_4 by factor 2.

Fig. 1 shows an example of an SDF graph. The worst-case execution time of each task τ_i, i.e, C_i, is shown below its name. For instance, task τ_2 has worst-case execution time $C_2 = 6$ time units and its data production rate x_2^2 on channel e_2 is 1. The repetition vector of G is $\vec{q} = [2, 1, 1, 1, 1, 2]^T$. Fig. 2 shows two examples of a CSDF graph. The repetition vector of the graphs shown in Fig. 2(a) and Fig. 2(b) are $[4, 2, 2, 2, 1, 1, 4]^T$ and $[4, 2, 1, 1, 1, 1, 2, 4]^T$, respectively.

B. Unfolding Transformation of SDF Graphs

The authors in [9] have shown that an SDF graph can be transformed into an equivalent CSDF graph by using a graph unfolding transformation technique to better utilize the underlying MPSoC platform by exposing more parallelism in the SDF graph. In fact, the intuition behind the unfolding, i.e., task replication, is to evenly distribute the workload of a task in the initial SDF graph among multiple of its replicas that are running concurrently. Given a vector $\vec{f} \in \mathbb{N}^n$ of replication factors, where f_i denotes the replication factor for task τ_i, the unfolding transformation replaces task τ_i with f_i replicas of task τ_i. To ensure the functional equivalence, the production and consumption sequences on channels in the obtained CSDF graph are calculated accordingly to the production and consumption rates in the initial SDF graph. After the replication, each replica $\tau_{i,k} \in G'$, $k \in [1, f_i]$, of task $\tau_i \in G$ will have the repetition $q_{i,k}$ [9]:

$$q_{i,k} = \frac{q_i \cdot \text{lcm}(\vec{f})}{f_i}, \tag{1}$$

where $\text{lcm}(\vec{f})$ is the least common multiple of all replication factors in $\vec{f}$. For example, after the unfolding of the SDF graph in Fig. 1 with replication vector $\vec{f} = [1, 1, 1, 1, 2, 1]$, the CSDF graph shown in Fig. 2(a) is obtained which has the repetition vector $\vec{q'} = [4, 2, 2, 2, 1, 1, 4]^T$, e.g., $q_{5,1} = q_{5,2} = \frac{1 \cdot \text{lcm}(1,1,1,1,2,1)}{2} = 1$.

C. Strictly Periodic Scheduling Framework

In [5], the real-time strictly periodic scheduling (SPS) framework for acyclic (C)SDF graphs is proposed. In this framework, every task $\tau_i \in V$ in an acyclic (C)SDF graph G is converted to a real-time implicit-deadline periodic task by deriving the *minimum period* (T_i) and *earliest start time* (S_i). In this framework, the *minimum period* (T_i) of every task $\tau_i \in V$ can be computed as:

$$T_i = \frac{\text{lcm}(\vec{q})}{q_i} \cdot s, \ \forall \tau_i \in V, \quad (2) \qquad s = \left\lceil \frac{\hat{W}}{\text{lcm}(\vec{q})} \right\rceil, \quad (3)$$

where $\text{lcm}(\vec{q})$ is the *least common multiple* of all repetition entries in $\vec{q}$, $\hat{W} = max_{\tau_j \in V}\{C_j.q_j\}$ is the maximum task workload of the (C)SDF graph, and C_j is the worst-case execution time of task τ_j. In general, the derived periods of tasks satisfy the condition $q_1 T_1 = q_2 T_2 = \cdots = q_n T_n = \alpha$, where α is the *graph iteration period* representing the duration needed by the graph to complete one iteration. Note that the derived period in Equation (2) is the minimum period for a task scheduled by SPS. But, there exist other longer valid periods for a task by scaling the minimum period by taking any integer $s > \left\lceil \frac{\hat{W}}{\text{lcm}(\vec{q})} \right\rceil$. Once task periods are computed, the utilization of task τ_i, denoted as u_i, can be computed as $u_i = C_i/T_i$, where $u_i \in (0, 1]$. Moreover, the throughput of each task τ_i can be computed as $1/T_i$. The throughput $\mathcal{R}$ of graph G, defined as the number of samples the graph can produce during a given time interval when its tasks are scheduled as strictly periodic tasks, is determined by the period of the output task (T_{out}) and is given by $\mathcal{R} = 1/T_{out}$. Note that when all tasks have the minimum periods, graph G can reach the maximum throughput achievable by the SPS framework. In this paper, we take this maximum achievable throughput as the throughput constraint.

Then, the *earliest start time* (S_i) of task τ_i is calculated such that τ_i is never blocked on reading data tokens from any input FIFO channel connected to it during its periodic execution, using the following expression:

$$S_i = \begin{cases} 0 & \text{if } \text{prec}(\tau_i) = \emptyset \\ \max_{\tau_j \in \text{prec}(\tau_i)} (S_{j \to i}) & \text{if } \text{prec}(\tau_i) \neq \emptyset \end{cases} \quad (4)$$

where $\text{prec}(\tau_i)$ is the set of predecessors of τ_i, and $S_{j \to i}$ is given by

$$S_{j \to i} = \min_{t \in [0, S_j + \alpha]} \{ t : \mathsf{prd}_{[S_j, \max(S_j, t) + k)}(\tau_j) \geq \mathsf{cns}_{[t, \max(S_j, t) + k]}(\tau_i) \ \forall k \in [0, 1, \cdots, \alpha] \} \quad (5)$$

where $\mathsf{prd}_{[t_s, t_e)}(\tau_j)$ is the number of tokens produced by τ_j during the time interval $[t_s, t_e)$, $\mathsf{cns}_{[t_s, t_e]}(\tau_i)$ is the number of tokens consumed by τ_i during the time interval $[t_s, t_e]$, and S_j is the earliest start time of a predecessor task τ_j.

The authors in [5] also provide a method to calculate the minimum required buffer size for each communication channel and the latency $\mathcal{L}$ of the (C)SDF graph scheduled in a strictly periodic fashion. In this method, once the start time of tasks have been calculated, the minimum buffer size of communication channel e_u connecting tasks τ_j and τ_i, denoted with $b_u(\tau_j, \tau_i)$, is calculated using the following expression:

$$b_u(\tau_j, \tau_i) = \max_{k \in [0, 1, \cdots, \alpha]} \Big\{ \mathsf{prd}_{[S_j, \max(S_j, S_i) + k)}(\tau_j) - \mathsf{cns}_{[S_i, \max(S_j, S_i) + k)}(\tau_i) \Big\} \quad (6)$$

that is the maximum number of unconsumed data tokens in channel e_u during the execution of τ_j and τ_i in one graph iteration period. The latency is also calculated as the elapsed time between the arrival of a data sample to the application and the output of the processed sample by the application.

D. System Model

The considered MPSoC platforms in this work are heterogeneous containing two types of processors [1], i.e., performance-efficient (PE) and energy-efficient (EE) processors, with distributed memories. We use Π_{PE} and Π_{EE} to denote the sets consisting of all PE processors and all EE processors, respectively. We denote the heterogeneous MPSoCs containing all PE and EE processors by $\Pi = \{\Pi_{PE}, \Pi_{EE}\}$.

The processors execute a set $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$ of n periodic implicit-deadline tasks, i.e., each task τ_i has a relative deadline D_i equal to its period T_i. Tasks can be preempted at any time. Every periodic task $\tau_i \in \Gamma$ is represented by a tuple $\tau_i = (C_i, S_i, T_i)$, where C_i is the worst-case execution time, S_i is the start time, and T_i is the period of the task. Since tasks may run on two different types of processors (PE and EE), the worst-case execution time value C_i for each task τ_i has two values, i.e., C_i^{PE} and C_i^{EE}, when EE and PE processors run at their maximum operating frequencies supported by the hardware platform. The utilization of task τ_i on a PE processor and an EE processor, denoted as u_i^{PE} and u_i^{EE}, is defined as $u_i^{PE} = C_i^{PE}/T_i$ and $u_i^{EE} = C_i^{EE}/T_i$, respectively. The total utilizations of the tasks assigned to a PE processor j and an EE processor k can be calculated by:

$$U(\pi_j^{PE}) = \sum_{\tau_i \in \Gamma_{\pi_j}} \frac{C_i^{PE}}{T_i}, \quad U(\pi_k^{EE}) = \sum_{\tau_i \in \Gamma_{\pi_k}} \frac{C_i^{EE}}{T_i} \quad (7)$$

where Γ_{π_j} and Γ_{π_k} represent sets of tasks assigned to PE processor j and EE processor k, respectively. In this paper, we consider Partitioned Earliest Deadline First (EDF) [7] scheduling algorithm to schedule the tasks on MPSoCs. The EDF is known to be optimal scheduling algorithm for periodic tasks on uniprocessors [4].

IV. Motivational Example

In this section, we take the SDF graph shown in Fig. 1 as our motivational example to demonstrate the necessity and efficiency of our proposed approach, presented in Section V, compared to related approaches [9], [12], [11], and [13] in terms of memory requirements, application latency, and number of required processors on a homogeneous platform[2],

[1] We refer to the ARM big.LITTLE architecture [1] including Cortex A15 'big' (PE) and Cortex A7 'little' (EE).

[2] In this section, we adopt a homogeneous platform because the related approaches [9], [12], [11] can support only such platform. Later, in Section VI-B, we compare our proposed approach and the approach proposed in [13] in terms of memory requirements and application latency on different heterogeneous platforms for a set of real-life benchmarks.

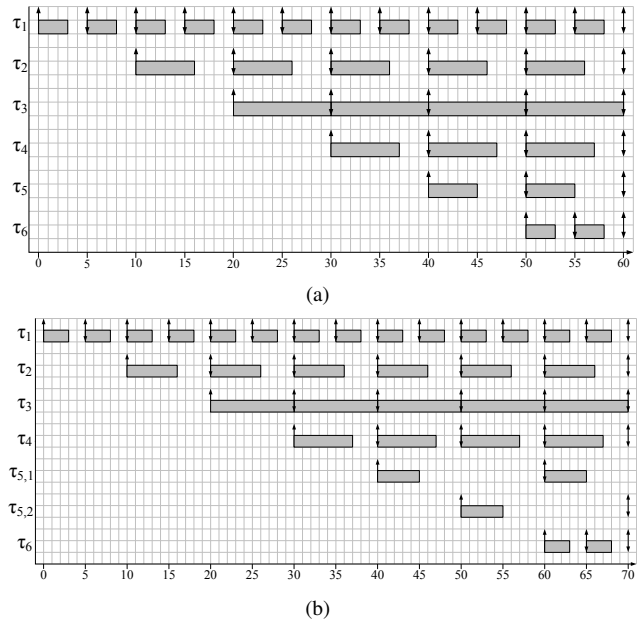


Fig. 3. A strictly periodic execution of the tasks in (a) the SDF graph G in Fig. 1 and (b) the CSDF graph G' in Fig. 2(a). The x-axis represents the time.

i.e., including only PE processors, to schedule the tasks in the SDF graph under a throughput constraint. By applying the SPS framework, briefly explained in Section III-C, for graph G shown in Fig. 1, the task set $\Gamma = \{\tau_1 = (C_1 = 3, S_1 = 0, T_1 = 5),\ \tau_2 = (6, 10, 10),\ \tau_3 = (10, 20, 10),\ \tau_4 = (7, 30, 10),\ \tau_5 = (5, 40, 10),\ \tau_6 = (3, 50, 5)\}$ of 6 strictly periodic implicit-deadline tasks can be derived. Based on these tuples, a strictly periodic schedule, as shown in Fig. 3(a), can be obtained for this graph. In this schedule, for instance, task τ_3 starts at time instant 20, executes for 10 time units, and repeats its execution every 10 time units. Since task τ_6 is the output task in this graph, the throughput of this schedule can be computed as $\mathcal{R} = \frac{1}{T_6} = \frac{1}{5}$. In this example, we consider this throughput as the throughput constraint. The application latency $\mathcal{L}$ for this schedule is 55 which is the elapsed time between the arrival of the first sample to the application, at $t = 0$, and the departure of the processed sample from task τ_6, at $t = 55$. The minimum number of processors needed for this schedule using an optimal scheduling algorithm, for instance [18], is $\mathrm{m_{OPT}} = \left\lceil \sum_{\tau_i \in \Gamma} u_i \right\rceil = \left\lceil \frac{3}{5} + \frac{6}{10} + \frac{10}{10} + \frac{7}{10} + \frac{5}{10} + \frac{3}{5} \right\rceil = 4$. However, using the partitioned EDF and the First-Fit Decreasing (FFD) [10] allocation algorithm, that is proven to be the resource efficient heuristic allocation algorithm [19], 6 processors are required for this schedule with task allocation $\Gamma_\Pi = \{\Gamma_{\pi_1} = \{\tau_3\}, \Gamma_{\pi_2} = \{\tau_4\}, \Gamma_{\pi_3} = \{\tau_1\}, \Gamma_{\pi_4} = \{\tau_2\}, \Gamma_{\pi_5} = \{\tau_6\}, \Gamma_{\pi_6} = \{\tau_5\}\}$. We refer to this scheduler as partitioned First-Fit Decreasing EDF (FFD-EDF) scheduler.

To reduce the number of required processors under the FFD-EDF scheduler while guaranteeing the throughput constraint, in this paper, we adopt the unfolding graph transformation technique presented in [9]. Let us assume that the platform has only 5 processors. Then, to schedule the application tasks on 5 processors under FFD-EDF scheduler, our proposed approach, explained in Section V, replicates task τ_5 in G by a factor of 2. Fig. 2(a) shows the CSDF graph obtained after applying the unfolding transformation on the initial graph G. By applying the SPS framework for graph G', shown in Fig. 2(a), the task set $\Gamma' = \{\tau_{1,1} = (3, 0, 5),\ \tau_{2,1} = (6, 10, 10),\ \tau_{3,1} = (10, 20, 10),\ \tau_{4,1} = (7, 30, 10),\ \tau_{5,1} = (5, 40, 20),\ \tau_{5,2} = (5, 50, 20),\ \tau_{6,1} = (3, 60, 5)\}$ of 7 strictly periodic tasks can be derived that is schedulable on 5 processors under FFD-EDF scheduler, with task allocation $\Gamma_\Pi = \{\Gamma_{\pi_1} = \{\tau_{3,1}\}, \Gamma_{\pi_2} = \{\tau_{4,1}, \tau_{5,1}\}, \Gamma_{\pi_3} = \{\tau_{1,1}, \tau_{5,2}\}, \Gamma_{\pi_4} = \{\tau_{2,1}\}, \Gamma_{\pi_5} = \{\tau_{6,1}\}\}$, while guaranteeing the throughput constraint of $\frac{1}{5}$. This is because, the workload of task τ_5 with $u_5 = \frac{5}{10}$ is now evenly distributed between two replicas $\tau_{5,1}$ and $\tau_{5,2}$ of task τ_5, i.e., $u_{5,1} = u_{5,2} = \frac{5}{20}$. Apparently, this workload distribution using the unfolding transformation can enable the FFD-EDF scheduler to more efficiently utilize the processors and schedule the tasks on fewer processors while guaranteeing the throughput constraint. The strictly periodic schedule of the task set Γ' is shown in Figure 3(b).

The approach in [9] is very close to our approach as it adopts the unfolding transformation technique to increase the throughput of an SDF graph scheduled on an MPSoC with fixed number of processors under partitioned scheduling. However, to schedule Γ on a platform with 5 processors under the throughput constraint of $\frac{1}{5}$, the approach in [9] performs differently. It first scales the period of the tasks in Γ using Equation (3) to make Γ schedulable on 5 processors under FFD-EDF scheduler. Due to scaling the periods, i.e., $s = 6 > \left\lceil \frac{10}{2} \right\rceil = 5$, however, the throughput is dropped to $\frac{1}{6}$. Then, to increase the throughput, the approach in [9] replicates the bottleneck task, i.e., the task with the heaviest workload during one graph iteration, and scales again the minimum computed periods of the tasks such that the new task set can be scheduled on 5 processors under FFD-EDF scheduler. This procedure is repeated until no throughput improvement can be gained anymore by task replication under the resource constraint. For our example in Fig. 1, the approach in [9] replicates tasks τ_3 and τ_4 by a factor of 2 that results in the throughput of $\frac{1}{3}$. Fig. 2(b) shows the CSDF graph G'' obtained after applying the unfolding transformation on graph G. Then, to schedule the tasks on 5 processors under FFD-EDF scheduler, the periods of tasks are scaled by using Equation (3), i.e., $s = 5 > \left\lceil \frac{12}{4} \right\rceil = 3$, where the throughput of $\frac{1}{5}$ finally could be achieved with the derived task set $\Gamma'' = \{\tau_{1,1} = (3, 0, 5),\ \tau_{2,1} = (6, 10, 10),\ \tau_{3,1} = (10, 20, 20),\ \tau_{3,2} = (10, 30, 20),\ \tau_{4,1} = (7, 40, 20),\ \tau_{4,2} = (7, 50, 20),\ \tau_{5,1} = (5, 60, 10),\ \tau_{6,1} = (3, 70, 5)\}$ of 8 strictly periodic tasks and the task allocation $\Gamma_\Pi = \{\Gamma_{\pi_1} = \{\tau_{4,1}, \tau_{1,1}\}, \Gamma_{\pi_2} = \{\tau_{4,2}, \tau_{2,1}\}, \Gamma_{\pi_3} = \{\tau_{6,1}\}, \Gamma_{\pi_4} = \{\tau_{3,1}, \tau_{3,2}\}, \Gamma_{\pi_5} = \{\tau_{5,1}\}\}$.

The approaches in [11], [12], adopt differently the semi-partitioned scheduling EDF-fm to allow certain tasks to migrate between processors for efficiently utilizing the remaining capacity on the processors. Under EDF-fm scheduling, the LUF heuristic in [11] allocates the tasks in Γ to 5 processors with task allocation $\Gamma_\Pi = \{\Gamma_{\pi_1} = \{\tau_3\}, \Gamma_{\pi_2} = \{\tau_4, \tau_5\}, \Gamma_{\pi_3} = \{\tau_5, \tau_1\}, \Gamma_{\pi_4} = \{\tau_6, \tau_2\}, \Gamma_{\pi_5} = \{\tau_2\}\}$, where task τ_5 is allowed to migrate between π_2 and π_3 and task τ_2 is

TABLE I
THROUGHPUT $\mathcal{R}$ (1/TIME UNITS), LATENCY $\mathcal{L}$ (TIME UNITS), MEMORY REQUIREMENTS M (BYTES), AND NUMBER OF PROCESSORS M FOR G UNDER DIFFERENT SCHEDULING/ALLOCATION APPROACHES.

Scheduling	Allocation	$\mathcal{R}$ [$\frac{1}{\text{t.u}}$]	$\mathcal{L}$ [t.u]	M [B]	m	m_{OPT}
EDF	FFD	1/5	55	155	6	4
	our	1/5	65 **(105)**	189 **(327)**	5 **(4)**	4
	FFD-EP [9]	1/5	75	228	5	4
EDF-*fm*	FFD-SP [12]	1/5	90	197	5	4
	LUF [11]	1/5	94	217	5	4
EDF-sh [13]		1/5	113 **(192)**	217 **(311)**	5 **(4)**	4

allowed to migrate between π_4 and π_5. In this task mapping, however, the fixed tasks τ_1, τ_4, and τ_6 that are allocated to the same processors as the migrating tasks τ_2 and τ_5, can miss their deadline by a bounded tardiness. To reduce the number of affected tasks by tardiness, the FFD-SP heuristic is proposed in [12] to restrict the task migrations. Under EDF-fm scheduling, this approach allocates the tasks in Γ to 5 processors with task allocation $\Gamma_\Pi = \{\Gamma_{\pi_1} = \{\tau_3\}, \Gamma_{\pi_2} = \{\tau_4, \tau_5\}, \Gamma_{\pi_3} = \{\tau_5, \tau_1\}, \Gamma_{\pi_4} = \{\tau_6\}, \Gamma_{\pi_5} = \{\tau_2\}\}$, where only task τ_5 is allowed to migrate between π_2 and π_3. Similar to the approach in [12], EDF-sh [13] allocates the tasks in Γ to 5 processors with task allocation $\Gamma_\Pi = \{\Gamma_{\pi_1} = \{\tau_3\}, \Gamma_{\pi_2} = \{\tau_4, \tau_5\}, \Gamma_{\pi_3} = \{\tau_5, \tau_1\}, \Gamma_{\pi_4} = \{\tau_6\}, \Gamma_{\pi_5} = \{\tau_2\}\}$, where only task τ_5 is allowed to migrate between π_2 and π_3.

The reduction on the number of required processors using our proposed approach and the related approaches, however, comes at the expense of more memory requirements and longer application latency either because of task replication[3], i.e., more tasks and data communication channels, or task migration, i.e., task tardiness. The throughput $\mathcal{R}$, latency $\mathcal{L}$, memory requirements M, i.e., sum of the buffer sizes of the communication channels in the graph and the code size of the tasks, and the number of required processors m for different scheduling/allocation approaches are given in Table I. Table I clearly shows that our proposed approach can reduce the number of required processors while keeping a low memory and latency increase compared to the related approaches for the same throughput constraint.

Let us now assume that the platform has only 4 processors. Then, all the related approaches, except EDF-sh, fail to guarantee the throughput constraint of $\frac{1}{5}$ under this resource constraint. However, our approach finds a vector of replication factors $\vec{f} = [1, 2, 1, 1, 5, 1]$ such that the CSDF graph obtained after applying the unfolding transformation on the initial SDF graph G, is schedulable on 4 processors under FFD-EDF scheduler using the SPS framework while guaranteeing the throughput constraint of $\frac{1}{5}$. EDF-sh can also allocate the tasks in Γ to 4 processors with task allocation $\Gamma_\Pi = \{\Gamma_{\pi_1} = \{\tau_3\}, \Gamma_{\pi_2} = \{\tau_4, \tau_2\}, \Gamma_{\pi_3} = \{\tau_2, \tau_5, \tau_1\}, \Gamma_{\pi_4} = \{\tau_5, \tau_6\}\}$, where task τ_2 is allowed to migrate between π_2 and π_3 and task τ_5 is allowed to migrate between π_3 and π_4. The memory requirement and application latency to schedule G on 4 processors using our proposed approach and EDF-sh are given in the third and seventh rows of Table I in parenthesis. As a result, our proposed approach can decrease the application latency by 45.3% while increasing the memory requirement by only 4.9% compared to EDF-sh.

From the above example, we can see the deficiencies of the related approaches because they have significant impact on the memory requirements and application latency when reducing the number of processors. Oppositely, our proposed approach that adopts the graph unfolding transformation, can reduce the number of processors while introducing lower memory and latency increase compared to the related approaches for the same throughput constraint.

[3]When replicating a task, its period is enlarged. As a consequence, the production of data tokens that are required by its data-dependent tasks to execute are postponed that results in a further offsetting of their start time, when calculating the earliest start time of tasks in SPS framework using Equation (4), hence increasing the application latency.

V. PROPOSED APPROACH

As explained and shown in Section IV, the partitioned scheduling algorithms, potentially, has the disadvantage that processors cannot be fully utilized, i.e., capacity fragmentation, because the static allocation of tasks on processors leaves an amount of unused capacity that is not sufficient to accommodate another task. Therefore, in this section, we present our novel approach that aims to exploit these unused capacity on the processors to reduce the number of processors needed to schedule the tasks in a hard real-time streaming application, modeled as an acyclic SDF graph and subjected to a throughput constraint, onto a heterogeneous MPSoC under partitioned scheduling algorithms, i.e., FFD-EDF scheduler. Our propose approach can achieve this goal by replicating tasks such that the required capacity of each resulting task replica is sufficiently small to make use of the available capacity on the processors.

The rationale behind our approach is the following: our approach first detects every task which cannot be entirely allocated to any individual under-utilized processor due to insufficient free capacity while, in total, there exists sufficient remaining capacity on under-utilized processors to schedule the tasks. Then, our approach replicates some of these tasks to distribute their workloads equally among more parallel replicas and fit them entirely on the remaining capacity of the processors without increasing the number of processors. As a result, our approach can alleviate the capacity fragmentation due to the FFD-EDF scheduler and utilize the processors more efficiently. In this section, therefore, we present a novel heuristic algorithm to derive the proper replication factor for each task in an SDF graph and the task allocation to reduce the number of required processors while guaranteeing the throughput constraint. In our approach, we use the SPS framework [5] to convert the tasks in the SDF graph to a set of periodic tasks.

The algorithm is given in Algorithm 1. It takes as input an SDF graph G, and a heterogeneous platform $\Pi = \{\Pi_{PE}, \Pi_{EE}\}$ with fixed number of PE and EE processors onto which the tasks in the graph have to be allocated. The algorithm returns as output a CSDF graph G', that is functionally equivalent to the initial SDF graph, and a task allocation set Γ_Π if a successful allocation is found. Otherwise, it returns false as output.

Algorithm 1: Proposed task allocation and finding proper replication factors for an SDF graph.

Input: An SDF graph $G = (V, E)$ and a heterogeneous MPSoC $\Pi = \{\Pi_{PE}, \Pi_{EE}\}$.
Output: *True*, an equivalent CSDF graph $G' = (V', E')$, and a task allocation set Γ_Π if a successful task allocation onto platform Π is found, *False* otherwise.

1 $\vec{f} = [1, 1, \cdots, 1]$; $G' \leftarrow G$; $\Pi' \leftarrow \Pi$;
2 Calculate period T'_i for PE type of processors for each task $\tau'_{i,k}$ in G' by using Equation (2) and Equation (3);
3 $\Gamma \leftarrow$ Sort tasks in G' in order of decreasing utilization;
4 **while** $True$ **do**
5 $\quad \Gamma_\Pi \leftarrow \{\Gamma_{\pi_1}, \Gamma_{\pi_2}, \cdots, \Gamma_{\pi_{|\Pi'|}}\}$;
6 $\quad \Gamma_1 \leftarrow \emptyset$;
7 $\quad$ **for** $\tau'_{i,k} \in \Gamma$ **do**
8 $\quad\quad$ **for** $1 \le j \le |\Pi'|$ **do**
9 $\quad\quad\quad$ **if** π_j *is an* EE *processor* **then**
10 $\quad\quad\quad\quad U_{left} = \sum_{\ell=1}^{j-1}(1 - U(\pi_\ell^{EE}))$; $u_i = u_i^{EE}$;
11 $\quad\quad\quad$ **if** π_j *is a* PE *processor* **then**
12 $\quad\quad\quad\quad U_{left} = \frac{C_i^{\mathrm{PE}}}{C_i^{\mathrm{EE}}} \sum_{\ell=1}^{|\Pi_{\mathrm{EE}}|}(1 - U(\pi_\ell^{EE})) + \sum_{\ell=|\Pi_{\mathrm{EE}}|+1}^{j-1}(1 - U(\pi_\ell^{PE}))$; $u_i = u_i^{PE}$;
13 $\quad\quad\quad$ Check EDF schedulability test on π_j;
14 $\quad\quad\quad$ **if** $\tau'_{i,k}$ *is not schedulable on* π_j **then**
15 $\quad\quad\quad\quad$ **continue**;
16 $\quad\quad\quad$ **else**
17 $\quad\quad\quad\quad$ **if** $U(\pi_j) = 0 \wedge U_{left} \ge u_i$ **then**
18 $\quad\quad\quad\quad\quad$ **if** $\tau'_{i,k}$ *is not stateful/in/out* **then**
19 $\quad\quad\quad\quad\quad\quad \Gamma_1 \leftarrow \Gamma_1 + \{\tau'_{i,k}, \pi_j\}$;
20 $\quad\quad\quad\quad \Gamma_{\pi_j} \leftarrow \tau'_{i,k}$;
21 $\quad\quad\quad\quad$ **break**;
22 $\quad\quad$ **if** $\tau'_{i,k}$ *is not allocated* **then**
23 $\quad\quad\quad$ **if** $u_i > U_{left}$ **then**
24 $\quad\quad\quad\quad$ **return** False;
25 $\quad\quad\quad \Pi' \leftarrow \Pi' + \pi^{PE}$;
26 $\quad\quad\quad$ **go to 5**
27 $\quad$ **for** $|\Pi_{EE}| < j \le |\Pi'|$ **do**
28 $\quad\quad$ **if** $\Gamma_{\pi_j} = \emptyset$ **then**
29 $\quad\quad\quad \Pi' \leftarrow \Pi' - \pi_j^{PE}$;
30 $\quad$ **if** $|\Pi'_{PE}| \le |\Pi_{PE}|$ **then**
31 $\quad\quad$ **break**;
32 $\quad$ **if** $\Gamma_1 \neq \emptyset$ **then**
33 $\quad\quad u_{left} = 0$;
34 $\quad\quad$ **for** $\{\tau'_{i,k}, \pi_j\} \in \Gamma_1$ **do**
35 $\quad\quad\quad$ **if** $1 - U(\pi_j) > u_{left}$ **then**
36 $\quad\quad\quad\quad u_{left} = 1 - U(\pi_j)$; $sel = i$;
37 $\quad$ **else**
38 $\quad\quad$ **return** False;
39 $\quad f_{sel} = f_{sel} + 1$; $f_{sel} \in \vec{f}$;
40 $\quad$ Get CSDF graph $G' = (V', E')$ by unfolding G with replication factors $\vec{f}$ using the method in Section III-B;
41 $\quad$ Calculate period T'_i for PE type of processors for each task $\tau'_{i,k}$ in G' by using Equation (2) and Equation (3);
42 $\quad \Gamma \leftarrow$ Sort tasks in G' in order of decreasing utilization;
43 **return** True, G', Γ_Π;

In Line 1, the algorithm initializes the replication factor of all tasks in graph G to 1, G' to G, and Π' to Π. In Line 2, the tasks in the graph G' are converted to periodic tasks using the SPS framework, explained in Section III-C, where the minimum period T'_i of each task in G' is calculated for PE type of processors, i.e., using C_i^{PE} for each task $\tau'_{i,k}$, by Equation (2) and Equation (3). *In this paper, we take the maximum throughput of graph G, achievable by the SPS framework with the minimum calculated periods, as the throughput constraint. Note that we can set another throughput constraint by scaling the minimum calculated periods.* Then, the algorithm builds a set of periodic tasks Γ in Line 3 and sorts the tasks in the order of decreasing utilization. Next, the algorithm enters to a **while loop**, Lines 4 to 42, where the task allocation is started on platform Π'. The body of the **while loop**, then, is repetitively executed to better utilize the processors' capacity using the graph unfolding transformation, explained in Section III-B, and allocate the tasks on platform Π'.

In Line 5, a task allocation set Γ_Π is created, to keep the tasks allocated to each processor individually. *Please note that in sets Π' and Γ_Π, the processors are ordered according to their type, where EE processors are followed by PE processors, to first utilize the energy-efficient processors*. In Line 6, an empty task set Γ_1 is defined to keep the candidate tasks for replication. In Lines 7 to 26, the algorithm allocates every task $\tau'_{i,k} \in \Gamma$ to one of the processors according to the FFD-EDF scheduler. In Lines 9 to 12, the total unused capacity U_{left} from the first processor π_1 to the current processor π_j is calculated. The current processor π_j can be either an EE processor or a PE processor. If it is an EE processor, all the previous processors are also EE processors due to the ordering of processors based on their type in platform Π'. In this case, the total unused capacity is calculated in Line 10 and stored in variable U_{left}. Otherwise, if π_j is a PE processor, the total unused capacity from π_1 to the current processor π_j, that includes all the EE processors followed by a subset of PE processors, is calculated in Line 12 and stored in variable U_{left}. Since the tasks have different utilization on the PE and EE processors, the total unused capacity on the EE processors are scaled accordingly by the proportion of the worst-case execution time of task $\tau'_{i,k}$ on the PE processor and EE processor, in Line 12.

In Line 13, the EDF schedulability test [7] is performed to check the schedulability of task $\tau'_{i,k}$ on processor π_j, i.e., $\tau'_{i,k}$ is schedulable if the total utilization of all tasks currently allocated to processor π_j (including $\tau'_{i,k}$) is not greater than the utilization bound of 1. If task $\tau'_{i,k}$ is not schedulable on processor π_j, the procedure of visiting the next processors is continued in Line 15. Otherwise, the candidate tasks for replication are identified first in Lines 17 to 19. If task $\tau'_{i,k}$ is allocated to an unused processor π_j while there is, in total, a sufficient unused capacity on the other under-utilized processors, the task is selected as a candidate to be replicated. This condition is checked in Line 17. *Note that stateful tasks, whose next execution depends on the current execution, and input and output tasks, which are connected to the external environment, are not replicated*. So, if task $\tau'_{i,k}$ satisfies the condition in Line 18, it is added in Line 19 to task set Γ_1 together with the processor π_j which it will be allocated to. Task $\tau'_{i,k}$ is actually allocated on processor π_j in Line 20 and

the procedure of vising the next processors is terminated in Line 21.

If task $\tau'_{i,k}$ is not allocated after visiting all processors in platform Π' and if the utilization of the task is larger than the total unused capacity left on the platform, then the algorithm cannot allocate the application tasks onto the given platform and returns False in Line 24. Otherwise, a PE processor is added to platform Π' in Line 25. This is because to reasonably find all candidate tasks for replication, the algorithm first checks how the processors are finally utilized by continuing the task mapping through adding an extra processor and finding a valid tasks' allocation using the FFD-EDF scheduler. For instance, the capacity of a processor that is fragmented by a big task can be efficiently exploited later by smaller tasks. Therefore there is no need to replicate such a big task. Later, by iteratively replicating the selected tasks, the algorithm gradually exploits the processors' capacity more efficiently and removes the extra added PE processors to finally find a valid tasks' allocation on the given platform Π. Next, the procedure is moved to Line 5 to find new tasks' allocation on the new platform Π'.

In Lines 27 to 29, the reduction of the number of required processors is performed by removing PE processors. If a PE processor with no allocated tasks is found, it means the task set Γ requires one PE processor fewer to be scheduled under FFD-EDF scheduler. Therefore, the PE processor with no allocated tasks is removed from platform Π' in Line 29. Then, Line 30 checks whether the number of PE processors in platform Π' is fewer than or equal to the number of PE processors in the given platform Π (*Note that both platforms* Π' *and* Π *have an equal number of EE processors as the algorithm only adds/removes PE processor to/from platform* Π'). If yes, then the CSDF graph G' and the task allocation set Γ_Π are returned in Line 43 and the algorithm terminates successfully.

If not, to better utilize the processors, a task is selected among the candidate tasks in Γ_1 for replication, in Lines 32 to 36. If task set Γ_1 is empty then no task could be selected for replication, therefore the algorithm cannot allocate the application tasks onto platform Π and returns False as output in Line 38. Among all the candidates in task set Γ_1, the task allocated to a processor with the largest amount of unused capacity is identified as a fragmentation-responsible task, in Lines 35 and 36. Then, the replication factor of this task is increased by one in Line 39 and the initial SDF graph is transformed into an equivalent CSDF graph using the unfolding transformation technique with unfolding vector $\vec{f}$, in Line 40. The periods of the tasks in the obtained CSDF graph are calculated again for PE type of processors using Equation (2) and Equation (3) in Line 41 and the new periodic tasks are sorted in Γ in the order of decreasing utilization, in Line 42. The body of the **while loop**, then, is repeated to either find successfully a task allocation of the transformed graph onto platform Π or fail due to lack of candidate tasks for replication, i.e., empty task set Γ_1.

VI. Evaluation

In this section, we present the experiments to evaluate our proposed approach in Section V. The experiments have been performed on a set of seven real-life streaming applications (benchmarks) modeled as acyclic SDF graphs taken from [12]. All SDF graphs are consistent and live. These benchmarks, from different application domains, are listed in Table II. In this table, $|V|$ denotes the number of tasks in a benchmark and $|E|$ denotes the number of communication channels among tasks in the corresponding SDF graph of the benchmark.

TABLE II
Benchmarks used for evaluation taken from [12].

Domain	Application	$\|V\|$	$\|E\|$
Signal Processing	Fast Fourier transform (FFT) kernel	32	32
	Multi-channel beamformer	57	70
	Time delay equalization (TDE)	35	35
Cryptography	Data Encryption Standard (DES)	55	64
	Serpent	120	128
Video processing	MPEG2 video	23	26
Sorting	Bitonic Parallel Sorting	41	48

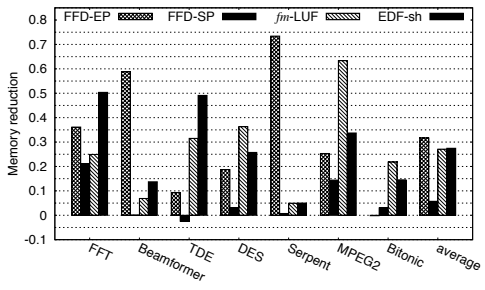
To demonstrate the effectiveness and efficiency of our proposed approach, we perform two experiments. In the first experiment, Section VI-A, we consider a homogeneous platform as considered in the related works [11], [12], [9]. In this experiment, we compare the application latency, the memory requirements, and the minimum number of processors needed to schedule the tasks of each benchmark under a given throughput constraint for a homogeneous platform, i.e, platform with only PE processors, obtained with six different scheduling/allocation approaches: (i) Partitioned EDF with FFD heuristic; (ii) Partitioned EDF with our proposed heuristic; (iii) Partitioned EDF with the heuristic proposed in [9]; (iv) Semi-partitioned EDF-*fm*, with the FFD-SP heuristic proposed in [12]; (v) Semi-partitioned EDF-*fm*, with the LUF heuristic proposed in [11]; (vi) Semi-partitioned EDF-sh [13]. These approaches are denoted in Table III with *FFD*, *our*, *FFD-EP*, *FFD-SP*, *fm-LUF*, and *EDF-sh*, respectively. In the second experiment, Section VI-B, we consider heterogeneous platforms, including PE and EE processors, as considered in the related work [13]. In this experiment, we compare the application latency and the memory requirements needed to schedule the tasks of each benchmark under a given throughput constraint obtained with Partitioned EDF with our proposed heuristic and Semi-partitioned EDF-sh [13] for different heterogeneous platforms. *Please note that we use the approach presented in [12] to handle data dependencies when using the scheduling/allocation approaches in [11], [13] for comparison with our approach.* The throughput constraint $\mathcal{R}$ of each benchmark, that is the maximum achievable throughput under the SPS framework, is given in the second column in Table III.

A. Homogeneous platform

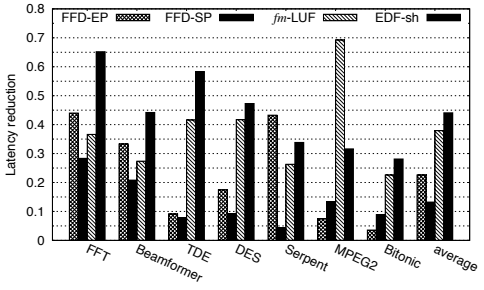
Let us first compare our approach with the related approaches in terms of the number of required processors. The number of required processors to guarantee the throughput constraint for each benchmark using an optimal scheduler, for instance [18], denoted as m_{OPT}, is given in the third column in Table III. To find the number of required processors using our proposed approach and the related approaches proposed in [11], [12], [9], [13], we set the number of PE processors on

TABLE III
COMPARISON OF DIFFERENT SCHEDULING/ALLOCATION APPROACHES.

Benchmark	$\mathcal{R}$ [$\frac{1}{\text{t.u.}}$]	OPT	Partitioned																	
			FFD			our			FFD-EP			FFD-SP			*fm*-LUF			EDF-sh		
		m_{OPT}	m_{FFD}	M_{FFD}[B]	$\mathcal{L}_{FFD}$[t.u.]	m_{our}	$\frac{M_{our}}{M_{FFD}}$	$\frac{\mathcal{L}_{our}}{\mathcal{L}_{FFD}}$	m_{EP}	$\frac{M_{EP}}{M_{FFD}}$	$\frac{\mathcal{L}_{EP}}{\mathcal{L}_{FFD}}$	m_{SP}	$\frac{M_{SP}}{M_{FFD}}$	$\frac{\mathcal{L}_{SP}}{\mathcal{L}_{FFD}}$	m_{LUF}	$\frac{M_{LUF}}{M_{FFD}}$	$\frac{\mathcal{L}_{LUF}}{\mathcal{L}_{FFD}}$	m_{sh}	$\frac{M_{sh}}{M_{FFD}}$	$\frac{\mathcal{L}_{sh}}{\mathcal{L}_{FFD}}$
FFT	1/6016	24	30	144680	192512	24 **(26)**	1.545 **(1.115)**	1.313 **(1.063)**	24	2.420	2.344	26	1.413	1.483	26	1.485	1.676	24	3.114	3.772
Beamformer	1/5076	26	28	14492	60912	26	1.144	1.166	26	2.781	1.750	26	1.145	1.474	26	1.229	1.606	26	1.326	2.091
TDE	1/32205	20	25	516282	1127175	20 **(21)**	1.597 **(1.180)**	1.286 **(1.086)**	21	1.301	1.195	20	1.560	1.396	21	1.722	1.860	20	3.139	3.086
DES	1/704	26	33	3381	33088	26 **(27)** **(28)**	1.182 **(1.103)** **(1.073)**	1.213 **(1.106)** **(1.085)**	27	1.357	1.340	27	1.138	1.218	28	1.684	1.862	26	1.592	2.301
Serpent	1/3336	39	42	59815	370296	39 **(40)**	1.016 **(1.005)**	1.090 **(1.027)**	40	3.78	1.81	40	1.012	1.074	39	1.068	1.479	39	1.069	1.648
MPEG2	1/7680	8	9	61909	138240	8	1.104	1.055	8	1.478	1.141	8	1.290	1.217	9	3.014	3.432	8	1.665	1.544
Bitonic	1/91	11	13	2374	2275	11	1.104	1.080	11	1.102	1.120	11	1.139	1.185	11	1.413	1.395	11	1.291	1.502



(a) Memory reduction



(b) Latency reduction

Fig. 4. Memory and latency reduction of our approach compared to the related approach with the same number of processors.

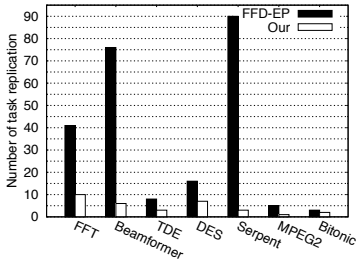


Fig. 5. The total number of task replications needed by FFD-EP and our proposed approach.

the homogeneous platform initially to m_{OPT}. Then, if the task set cannot be scheduled on the platform, we add one more PE processor and repeat the task allocation procedure again until a successful task allocation is found.

As can be seen in Table III, the FFD approach requires considerably more processors, on average 17.6% more, than the number of required processors by an optimal scheduler, see column m_{FFD}. In contrast, our approach and EDF-sh require the same number of processors as the optimal scheduler while maintaining the same throughput for this set of benchmarks, see columns m_{our} and m_{sh}, respectively. For the other approaches, although they require fewer processors than FFD, they still require more processors than our approach for some benchmarks. For instance, the approach FFD-EP requires one more processor for TDE, DES, and Serpent, see column m_{EP}; The approach FFD-SP requires two more processors for FFT and one more processor for DES and Serpent, see column m_{SP}; Finally the approach *fm*-LUF requires two more processors for FFT and DES and one more processor for TDE and MPEG2, see column m_{LUF}. Although this difference in terms of number of required processors is not too large, it clearly reveals that our approach is more capable of scheduling the benchmarks with fewer processors compared to the FFD-EP, FFD-SP, and *fm*-LUF approaches while meeting the same throughput constraint.

However, this reduction on the number of required processors comes at the expense of increased memory requirements and application latency. For each benchmark, columns M_{FFD} and $\mathcal{L}_{FFD}$ report the memory requirements, expressed in bytes, and the application latency, expressed in time units, under FFD, respectively. The memory requirements is computed as the sum of the buffer sizes of the communication channels in the (C)SDF graph and the code size of the tasks. For each benchmark, the increase on memory requirements and application latency by our approach over FFD are given in columns $\frac{M_{our}}{M_{FFD}}$ and $\frac{\mathcal{L}_{our}}{\mathcal{L}_{FFD}}$, respectively, that are on average 24.2% and 17.2%, respectively. Similarly, the increases on memory requirements and application latency are on average respectively 100% and 52.85% for FFD-EP, 24.3% and 29.2% for FFD-SP, 65.9% and 90.2% for *fm*-LUF, and finally 88.5% and 127.8% for EDF-sh compared to FFD. From these numbers, we can conclude that not only our approach requires fewer processors compared to the related approaches, but also it imposes, on average, lower memory and latency overheads.

To further compare our approach with the related approaches, we compute the memory requirements and application latency of our approach when equal number of processors as the related approaches are used, see the bolded numbers in parenthesis in columns m_{our}, $\frac{M_{our}}{M_{FFD}}$, and $\frac{\mathcal{L}_{our}}{\mathcal{L}_{FFD}}$. To ease the interpretation of Table III for this comparison, Fig. 4(a) and Fig. 4(b) illustrate the memory and latency reductions obtained by our approach compared to the related approaches, respectively. For instance, the reduction on memory requirements is computed using the following equation:

$$r = \frac{M_{rel} - M_{our}}{M_{rel}} \tag{8}$$

where M_{rel} is the memory requirements of scheduling an application using a related approach and M_{our} denotes the memory requirements achieved by our approach for the same number of processors. In Fig. 4(a), we can see that our approach can reduce the memory requirements by an average of 31.43%,

TABLE IV
RUNTIME (IN SECONDS) COMPARISON OF DIFFERENT SCHEDULING/ALLOCATION APPROACHES.

Benchmark	t_{FFD}	t_{our}	$t_{FFD\text{-}EP}$	$t_{FFD\text{-}SP}$	t_{fm-LUF}	$t_{EDF\text{-}sh}$
FFT	0.001	5.95	451.48	0.22	0.17	0.024
Beamformer	0.011	5.16	126.30	0.100	0.037	0.022
TDE	0.005	3.96	138.32	0.011	0.013	0.011
DES	0.002	9.41	14.20	0.28	1.013	0.021
Serpent	0.025	56.43	960.30	1.44	0.45	0.09
MPEG2	0.001	0.015	3.25	0.002	0.002	0.004
Bitonic	0.001	0.127	0.093	0.003	0.011	0.034

5.72%, 27.11%, and 27.46% compared to FFD-EP, FFD-SP, *fm*-LUF, and EDF-sh, respectively. In Fig. 4(a), however, there are two exceptions where our approach requires 2.43% and 0.19% more memory for TDE and Bitonic compared to FFD-SP and FFD-EP, respectively. In Fig. 4(b), we can also see that our approach can reduce the application latency considerably for all benchmarks by an average of 22.60%, 13.24%, 37.92%, and 44.09% compared to FFD-EP, FFD-SP, *fm*-LUF, and EDF-sh, respectively. This comparison clearly demonstrates that for most of the benchmarks our approach is more efficient than the related approaches in exploiting the available resources. Compared to FFD-EP, that is the closest approach to our approach as both approaches adopt the graph unfolding transformation, our efficiency comes from significantly reducing the number of required task replications due to our novel Algorithm 1, as shown in Fig. 5. This figure clearly shows that, by replicating the right tasks, our proposed approach can reduce the total number of task replications significantly, up to 30 times, compared to FFD-EP. From Figure 4, it can be also observed that our proposed approach works better for some applications than for others compared to the related approaches. Given the (C)SDF graph of each application has different properties, e.g, the number of tasks, the tasks' workload, the graph's topology, repetition vector, etc., the applications are represented with a different set of periodic tasks in terms of the number of tasks and the utilization of tasks. Therefore, this variation on the number of tasks and the utilization of tasks in the set of periodic tasks according to each application can have different impact on the performance of different scheduling/allocation approaches.

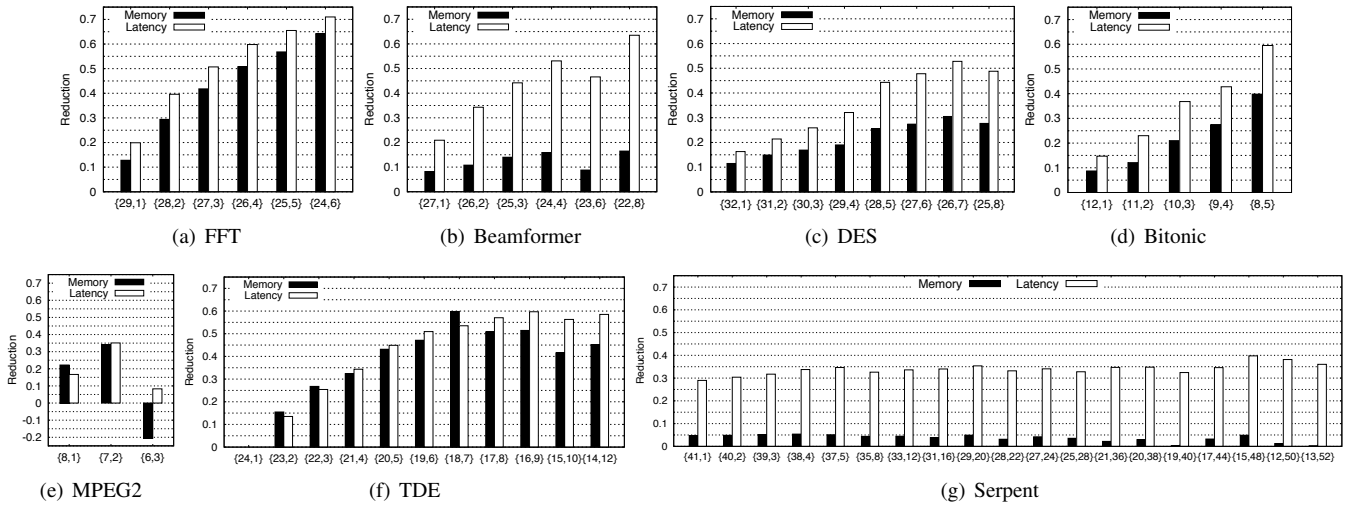
Finally, we evaluate the efficiency of our algorithm in terms of the execution time. We compare the execution time of our algorithm with the corresponding execution times of FFD, FFD-EP, FFD-SP, *fm*-LUF, and EDF-sh. The comparison is given in Table IV. As can be seen from Table IV, the execution time of FFD and EDF-sh are always within less than 34 millisecond, while the execution times of FFD-SP and *fm*-LUF are within less than 1.5 seconds. However, the execution time of our algorithm is longer than FFD, FFD-SP, *fm*-LUF, and EDF-sh due to its iterative execution nature, but it is within less than 10 seconds for most of the cases and within less than 1 minute for one case which is reasonable given that our proposed approach is a design-time approach and that it achieves better resource utilization. Among all the approaches, FFD-EP has the highest execution time, which is within less than 17 minutes, due to excessive number of algorithm iterations. This excessive number of iterations is due to the excessive number of required task replications in FFD-EP as shown in Fig. 5.

B. Heterogeneous platform

To compare our proposed approach and EDF-sh [13] on heterogeneous platforms, in this section, we conduct experiments on a set of heterogeneous platforms including different number of PE and EE processors. To do so, we initially generate a heterogeneous platform having $m_{FFD}-1$ PE processors (see Table III for m_{FFD}) and 1 EE processor for each benchmark and iteratively replace one PE processor with one EE processor (or more EE processors if the task set is not schedulable on the platform). However, due to the restrictive allocation rules in EDF-sh to ensure bounded tardiness for deadline misses, EDF-sh cannot find a task allocation for some heterogeneous platforms that have fewer than a certain number of PE processors. Therefore, we only compare our approach with EDF-sh on the heterogeneous platforms for which EDF-sh can successfully allocate the tasks for each benchmark. Fig. 6 shows the memory and latency reductions obtained by our approach compared to EDF-sh for each benchmark individually. The reductions are computed using Equation (8). In Fig. 6, the x-axis shows different heterogeneous platforms, comprised of different number of PE and EE processors denoted by {number of PEs, number of EEs}. The y-axis shows the reduction on the memory requirements and application latency. Note that for the Serpent benchmark, a subset of heterogeneous platforms is shown in Fig. 6(g) due to the space limitation, while for the other benchmarks all successful heterogeneous platforms are shown.

From Fig. 6, it can be observed that our proposed approach outperforms EDF-sh in terms of memory requirements and application latency for most of the cases. Compared to EDF-sh, our approach can reduce the memory requirements and application latency by an average of 42.6% and 51.1%, 12.4% and 43.8%, 21.7% and 36.2%, 21.8% and 35.4%, 11.9 % and 20.1%, 37.6 % and 42.2%, and 3.6 % and 33.8% for the FFT, Beamformer, DES, Bitonic, MPEG, TDE, and Serpent benchmarks, respectively. For the MPEG benchmark, however, our proposed approach increases the memory requirements compared to EDF-sh by 20.6% on a platform including 6 PE and 3 EE processors. This is because our approach excessively replicates a task to utilize the unused capacity left on the under-utilized processors. Therefore, the memory requirements increase significantly due to the code and data memory overheads. However, since the replicated task has low impact on the application latency, our approach can still reduce the application latency by 8.3% compared to EDF-sh. For the TDE benchmark, both approaches find a task allocation without requiring either task replication (our) or task migration (EDF-sh) on a platform including 24 PE and 1 EE processors, therefore no reduction is achieved for both memory requirements and latency in this case.

In addition, it can be observed in Fig. 6 that for most of the cases by replacing more PE processors with EE processors on the platform, our approach can further reduce the memory requirements and application latency compared to EDF-sh.



(a) FFT (b) Beamformer (c) DES (d) Bitonic

(e) MPEG2 (f) TDE (g) Serpent

Fig. 6. Memory and latency reduction of our approach compared to EDF-sh [13] for real-life benchmarks on different heterogeneous platforms.

This is mainly because, by replacing more number of PE processors with EE processors on the platform, the number of migrating tasks under EDF-sh scheduler is considerably increased while the number of task replications is only gently increased in our approach. As a result, more fixed tasks are affected by migrating tasks and can miss their deadlines, by a bounded tardiness, under EDF-sh scheduler that comes at the expense of more memory requirements and longer application latency. According to the approach presented in [12], the memory requirements increases due to both the size of buffers, that have to be enlarged to handle task tardiness, and the code size overhead of task replicas, which are necessary in case of migrating tasks. In addition, the application latency increases due to the postponement of task start times needed to handle task tardiness.

VII. Conclusion

In this paper, we have presented a novel heuristic algorithm that determines a replication factor for each task in an acyclic SDF graph, which is subject to a throughput constraint, such that the number of required processors to schedule the tasks in the obtained transformed graph is reduced under partitioned scheduling algorithms. By performing tasks replication, the tasks' workload is distributed among more parallel tasks' replicas with larger period and lower utilization in the obtained transformed graph. Therefore, the required capacity of the tasks which are replicated, is split up in multiple smaller chunks that can more likely fit into the left capacity on the processors and alleviate the capacity fragmentation due to partitioned scheduling algorithms, hence reducing the number of required processors. The experiments on a set of real-life streaming applications show that our proposed approach can reduce the number of required processors by up to 7 processors with increasing the memory requirements and application latency by 24.2% and 17.2% on average compared to FFD while meeting the same throughput constraint. We also show that our approach can still reduce the number of required processors by up to 2 processors and considerably improve the memory requirements and application latency by up to 31.43% and 44.09% on average compared to the other related approaches while meeting the same throughput constraint.

References

[1] P. Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, 17, 2011.
[2] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9), 1987.
[3] G. Bilsen et al. Cycle-static dataflow. *IEEE Transactions on signal processing*, 44(2), 1996.
[4] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM computing surveys (CSUR)*, 43(4), 2011.
[5] M. Bamakhrama and T. Stefanov. On the hard-real-time scheduling of embedded streaming applications. *Design Automation for Embedded Systems*, 17(2), 2013.
[6] E. Cannella et al. Adaptivity support for MPSoCs based on process migration in polyhedral process networks. *VLSI Design*, 2012, 2012.
[7] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1), 1973.
[8] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. In *RTSS*, 2010.
[9] J. Spasic, D. Liu, and T. Stefanov. Exploiting resource-constrained parallelism in hard real-time streaming applications. In *DATE*, 2016.
[10] E. G. Coffman Jr, M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: A survey. In *Approximation algorithms for NP-hard problems*. PWS Publishing Co., 1996.
[11] J. H. Anderson, V. Bud, and U. C. Devi. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *ECRTS*, 2005.
[12] E. Cannella, M. Bamakhrama, and T. Stefanov. System-level scheduling of real-time streaming applications using a semi-partitioned approach. In *DATE*, 2014.
[13] K. Yang and J. H Anderson. Soft real-time semi-partitioned scheduling with restricted migrations on uniform heterogeneous multiprocessors. In *RTNS*, 2014.
[14] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *PACT*, 2010.
[15] A. Burns et al. Partitioned EDF scheduling for multiprocessors using a C= D task splitting scheme. *Real-Time Systems*, 48(1), 2012.
[16] A. K. Singh et al. Mapping on multi/many-core systems: survey of current and emerging trends. In *DAC*, 2013.
[17] J. Spasic, D. Liu, and T. Stefanov. Energy-efficient mapping of real-time applications on heterogeneous MPSoCs using task replication. In *CODES+ISSS*, 2016.
[18] S. K. Baruah et al. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6), 1996.
[19] H. Aydin and Q. Yang. Energy-aware partitioning for multiprocessor real-time systems. In *IPDPS*, 2003.