Generalized strictly periodic scheduling analysis, resource optimization, and implementation of adaptive streaming applications

Niknam, S.

Chapter 4

Exploiting Parallelism in Streaming Applications to Efficiently Utilize Processors

Sobhan Niknam, Peng Wang, Todor Stefanov. "Resource Optimization for Real-Time Streaming Applications using Task Replication". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 37, No. 11, pp. 2755-2767, Nov 2018.

IN this chapter, we present our novel algorithm to derive an alternative application specification for efficient utilization of processors, which corresponds to the second research contribution, briefly introduced in Section 1.5.2, to address research question **RQ2(A)**, described in Section 1.4.2. The remainder of this chapter is organized as follows. Section 4.1 introduces, in more details, the problem statement and the addressed research question. It is followed by Section 4.2, which gives a summary of the contributions presented in this chapter. Section 4.3 gives an overview of the related work. Section 4.4 introduces the extra background material needed for understanding the contributions of this chapter. Section 4.5 gives a motivational example. Section 4.6 presents our proposed algorithm. Section 4.7 presents the experimental evaluation of our proposed algorithm. Finally, Section 4.8 ends the chapter with conclusions.

4.1 Problem Statement

Recall, from Section 2.2, that in real-time systems, tasks can be scheduled on multiprocessor systems using three main classes of algorithms, i.e., global, partitioned, and hybrid scheduling algorithms, based on whether a task can migrate between processors [29]. Under global scheduling algorithms, all the tasks can migrate between all 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 [24]. 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) [54]. 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 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, Bastoni et al. [14] concluded that the partitioned algorithm outperforms the other algorithms when hard real-time systems are considered.

Although partitioned scheduling algorithms do not impose any migration and memory overheads, they are known to be non-optimal for scheduling real-time periodic tasks [29]. 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 periodic 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 satisfy a performance requirement, i.e., throughput, in an application should be minimized. This can be difficult because often the given initial application specification, i.e., the initial 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 specification 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 of the application, 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, in this chapter, we investigate the possibility to determine the right amount of parallelism in a streaming application, modeled as an acyclic SDF graph, to minimize the number of required processors under partitioned scheduling algorithms while satisfying a given performance requirement.

4.2 Contributions

In order to address the problem described in Section 4.1, in this chapter, we propose a novel algorithm to find a proper replication factor for each task in an initial application specification, such that the obtained alternative one requires fewer processors under partitioned scheduling algorithms and a given throughput requirement is satisfied. More specifically, the main novel contributions of this chapter 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 technique in [81], we propose an approach to determine a replication factor for each task of the application 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 issue introduced by partitioned scheduling algorithms, hence reducing the number of required processors.
- We show, on a set of real-life streaming applications, that our algorithm

significantly reduces the number of required processors compared to the 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 algorithm can still reduce the number of required processors compared to the related approaches in [4, 23, 81, 92] with significantly improving the memory requirements and application latency while maintaining the same application throughput.

Scope of work. In this chapter, we consider that streaming applications are modeled as acyclic SDF graphs. This restriction comes from the related approaches that are adopted for comparison with our proposed algorithm. These approaches can only be applied on sets of implicit deadline periodic tasks which can be derived from acyclic SDF graphs using the SPS framework, described in Section 2.3.

4.3 Related Work

In order to overcome the scheduling problems in global and partitioned scheduling algorithms, briefly explained in Section 4.1, a restricted-migration semi-partitioned scheduling algorithm, called EDF-fm, in the class of hybrid scheduling algorithms, is proposed in [4] 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 [92], 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, [20] proposes the C=D approach to split real-time periodic tasks on homogeneous multiprocessor systems while on each processor a normal EDF scheduler is used. In the C=D approach, a task which cannot be entirely allocated on any processor is split up in two parts that can be entirely allocated on different processors. 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 [4, 20, 92] only consider

sets of independent tasks. In contrast, 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 runtime overhead is imposed to the system by our algorithm in comparison to [20] and no task is subjected to a deadline miss in comparison to [4, 92]. Compared to the approaches in [4, 20] that only support homogeneous platforms, our proposed algorithm also supports heterogeneous platforms.

To allocate data-dependent application tasks to a multiprocessor platform, many techniques have already been devised [75]. Existing approaches which are close to our work are [8, 23, 81]. The authors in [8] propose the SPS framework, briefly described in Section 2.3, to only convert each actor 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 [8], however, no optimization technique for different system design metrics, such as, throughput, latency, memory, number of processors, etc., is proposed. In contrast, in this chapter, we propose a heuristic algorithm on top of the SPS framework to optimize the number of required processors when scheduling a hard real-time streaming application with a given throughput requirement onto a heterogeneous MPSoC under partitioned scheduling algorithms.

Using the SPS framework, the authors in [23] propose a heuristic under the semi-partitioned scheduling algorithm in [4] 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 proposed extension of the SPS framework in [23]. In [4], 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 [23]. However, in these approaches [4, 23], the deadline misses of the fixed tasks due to task migration have significant overheads on the memory requirements and the application latency. In contrast, 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 algorithm, as shown in Section 4.7, processors can be more efficiently utilized while imposing considerably lower overheads on the memory

requirements and the application latency compared to the approaches in [4,23]. In addition, our proposed algorithm supports heterogeneous platforms while the approaches in [4,23] can only support homogeneous platforms.

In [81], 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 technique 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 satisfy a given throughput requirement under limited resources, the approach in [81] does not always replicate the right tasks, as shown in Section 4.5. Consequently, this leads to more parallelism than needed which increases the memory requirements and application latency unnecessarily. In contrast, we propose an algorithm that supports heterogeneous platforms. In addition, our proposed algorithm 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 [81]. 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 4.7, our proposed algorithm 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 [81].

In [80], the authors use the same approach as in [81] 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 this chapter, however, we focus on and solve a totally different problem, that is, how the unfolding transformation technique can be exploited to reduce the number of required processors when a partitioned scheduling algorithm is used. In our algorithm, we do not search for and take the bottleneck task, which is taken in [80], 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.

4.4 Background

In this section, we first introduce the unfolding transformation technique, presented in [81], that we use to replicate the tasks in an application initially modeled as an SDF graph. Then, we present the system model considered in this chapter.

4.4.1 Unfolding Transformation of SDF Graphs

The authors in [81] 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., replication, of an actor in the initial SDF graph is to evenly distribute the workload of the actor among multiple of its replicas that are running concurrently. Given a vector $\vec{f} \in \mathbb{N}^{|\mathcal{A}|}$ of replication factors, where f_i denotes the replication factor for actor $A_i \in \mathcal{A}$, the unfolding transformation replaces actor A_i with f_i replicas of actor A_i, denoted by $A_{i,k}, k \in [1, f_i]$. To ensure the functional equivalence, the production and consumption sequences on FIFO 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 $A_{i,k}$ of actor A_i will have the repetition

$$q_{i,k} = \frac{q_i \cdot \text{lcm}(\vec{f})}{f_i}, \tag{4.1}$$

where $\text{lcm}(\vec{f})$ is the least common multiple of all replication factors in $\vec{f}$. For example, consider the SDF graph G shown in Figure 4.1 with the repetition vector $\vec{q} = [2, 1, 1, 1, 1, 2]^T$, derived using Theorem 2.1.1. After unfolding of G with replication vector $\vec{f} = [1, 1, 1, 1, 2, 1]$, the CSDF graph G' shown in

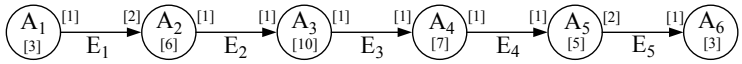
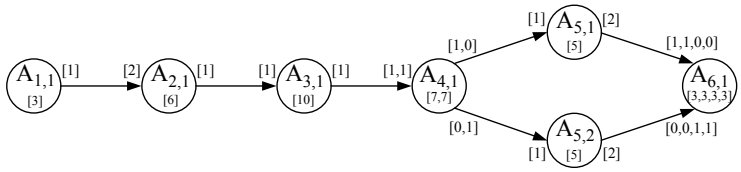
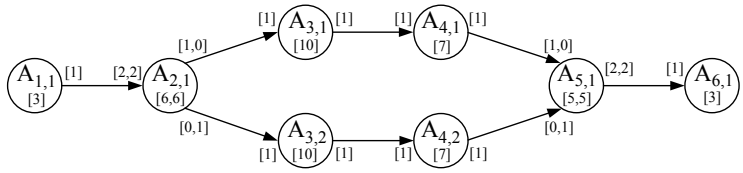


Figure 4.1: *An SDF graph G.*



(a) A CSDF graph G'



(b) A CSDF graph G''

Figure 4.2: *Equivalent CSDF graphs of the SDF graph G in Figure 4.1 obtained by (a) replicating actor A_5 by factor 2 and (b) replicating actors A_3 and A_4 by factor 2.*

Figure 4.2(a) is obtained which has the repetition vector $\vec{q}' = [4,2,2,2,1,1,4]^T$, e.g.,

$$q_{5,1} = q_{5,2} = \frac{1 \cdot \text{lcm}(1,1,1,1,2,1)}{2} = 1.$$

4.4.2 System Model

The considered MPSoC platforms in this chapter 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 containing the PE processors and the EE processors, respectively. We denote the heterogeneous MPSoCs containing all PE and EE processors by $\Pi = \{\Pi_{PE}, \Pi_{EE}\}$. Since application tasks may run on two different types of processors (PE and EE), the worst-case execution time value C_i for each periodic task $\tau_i \in \Gamma$ has two values, i.e., C_i^{PE} and C_i^{EE}, when EE and PE processors run at their maximum operating clock frequencies supported by

[1] We refer to the ARM big.LITTLE architecture [40] including Cortex A15 'big' (PE) and Cortex A7 'LITTLE' (EE) that is shown in Figure 1.1.

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. Now, let us consider an x-partition ${}^x\Gamma$ of task set Γ. Then, the total utilizations of the tasks allocated on a PE processor j and an EE processor k can be calculated by:

$$u_{\pi_j^{PE}} = \sum_{\tau_i \in {}^x\Gamma_j} \frac{C_i^{PE}}{T_i}, \quad u_{\pi_k^{EE}} = \sum_{\tau_i \in {}^x\Gamma_k} \frac{C_i^{EE}}{T_i} \tag{4.2}$$

where ${}^x\Gamma_j$ and ${}^x\Gamma_k \in {}^x\Gamma$ represent sets of tasks allocated on PE processor j and EE processor k, respectively.

4.5 Motivational Example

In this section, we take the SDF graph G shown in Figure 4.1 as our motivational example to demonstrate the necessity and efficiency of our proposed algorithm, presented in Section 4.6, compared to the related approaches [81], [23], [4], and [92] in terms of memory requirements, application latency, and number of required processors on a homogeneous platform[2], i.e., including only PE processors, to schedule the actors in the SDF graph under a given throughput requirement. By applying the SPS framework [8], briefly described in Section 2.3, for graph G, the task set $\Gamma = \{\tau_1 = (C_1 = 3, T_1 = 5, S_1 = 0, D_1 = T_1 = 5), \tau_2 = (6, 10, 10, 10), \tau_3 = (10, 10, 20, 10), \tau_4 = (7, 10, 30, 10), \tau_5 = (5, 10, 40, 10), \tau_6 = (3, 5, 50, 5)\}$ of six IDP tasks can be derived. Based on these tuples, a strictly periodic schedule, as shown in Figure 4.3(a), can be obtained for this graph. Using Equation (2.15), 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 given throughput requirement. Moreover, using Equation (2.19), 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, according to Equation (2.8), is $\check{m}_{\text{OPT}} = \left\lceil \sum_{\tau_i \in \Gamma} u_i \right\rceil = \left\lceil \frac{3}{5} + \frac{6}{10} + \frac{10}{10} + \frac{7}{10} + \frac{5}{10} + \frac{3}{5} \right\rceil = 4$. However, using the partitioned EDF and the First-Fit Decreasing (Utilization) [28] allocation algorithm, that is proven to be the resource efficient heuristic allocation algorithm [5], 6 processors are required for this schedule with task

[2] In this section, we adopt a homogeneous platform because the related approaches [4, 23, 81] can support only such platform. Later, in Section 4.7.2, we compare our proposed approach and the approach proposed in [92] in terms of memory requirements and application latency on different heterogeneous platforms for a set of real-life applications.

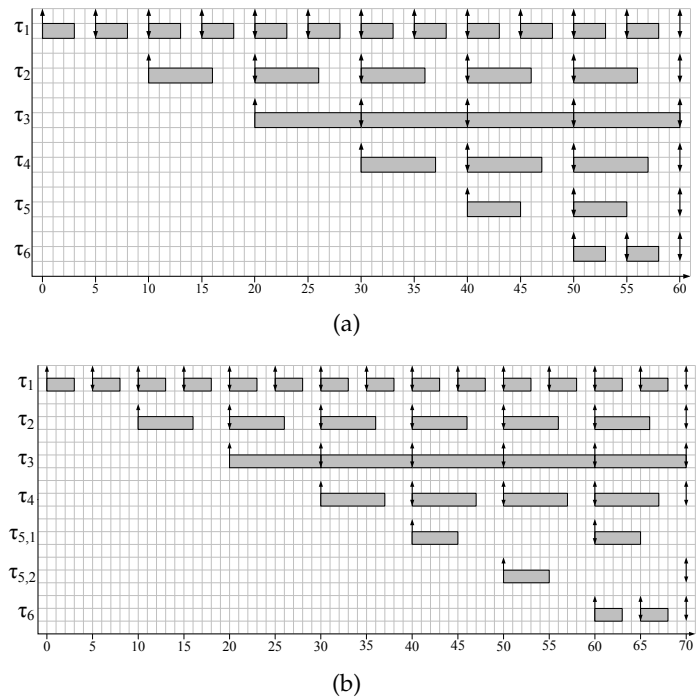


Figure 4.3: *A strictly periodic execution of tasks corresponding to the actors in: (a) the SDF graph G in Figure 4.1 and (b) the CSDF graph G′ in Figure 4.2(a). The x-axis represents the time.*

allocation ${}^6\Gamma = \{{}^6\Gamma_1 = \{\tau_3\}, {}^6\Gamma_2 = \{\tau_4\}, {}^6\Gamma_3 = \{\tau_1\}, {}^6\Gamma_4 = \{\tau_2\}, {}^6\Gamma_5 = \{\tau_6\}, {}^6\Gamma_6 = \{\tau_5\}\}$. 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 satisfying the given throughput requirement $\mathcal{R} = \frac{1}{5}$, we adopt the unfolding graph transformation technique in [81], briefly explained in Section 4.4.1. Let us assume that the platform has only 5 processors. Then, to schedule the application on 5 processors under FFD-EDF scheduler, our proposed algorithm, explained in Section 4.6, replicates actor A_5 in graph G by a factor of 2. Figure 4.2(a) shows the CSDF graph G' obtained after applying the unfolding transformation on the initial graph G shown in Figure 4.1. By applying the SPS framework for graph G', the task set $\Gamma' = \{\tau_{1,1} = (3, 5, 0, 5), \tau_{2,1} = (6, 10, 10, 10), \tau_{3,1} = (10, 10, 20, 10), \tau_{4,1} = (7, 10, 30, 10), \tau_{5,1} = (5, 20, 40, 20), \tau_{5,2} = (5, 20, 50, 20), \tau_{6,1} = (3, 5, 60, 5)\}$ of seven IDP tasks can be derived which is schedulable on 5 processors under FFD-EDF scheduler, with task

allocation ${}^5\Gamma' = \{{}^5\Gamma'_1 = \{\tau_{3,1}\}, {}^5\Gamma'_2 = \{\tau_{4,1}, \tau_{5,1}\}, {}^5\Gamma'_3 = \{\tau_{1,1}, \tau_{5,2}\}, {}^5\Gamma'_4 = \{\tau_{2,1}\}, {}^5\Gamma'_5 = \{\tau_{6,1}\}\}$, while satisfying the given throughput requirement of $\frac{1}{5}$. This is because, the workload of task τ_5, corresponding to actor A_5 of graph G, with $u_5 = \frac{5}{10}$ is now evenly distributed between two tasks $\tau_{5,1}$ and $\tau_{5,2}$, corresponding to replicas $A_{5,1}$ and $A_{5,2}$ of actor A_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 satisfying the throughput requirement. The strictly periodic schedule of the task set Γ' is shown in Figure 4.3(b).

The approach in [81] 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 requirement of $\frac{1}{5}$, the approach in [81] performs differently. It first scales the period of the tasks in Γ using Equation (2.13) to make Γ schedulable on 5 processors under FFD-EDF scheduler. Due to scaling the periods, i.e., $s = 6 > \lceil \frac{10}{2} \rceil = 5$, however, the throughput is dropped to $\frac{1}{6}$. Then, to increase the throughput, the approach in [81] replicates the actor corresponding to the bottleneck task, i.e., the actor 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 Figure 4.1, the approach in [81] replicates actors A_3 and A_4 corresponding to tasks τ_3 and τ_4 by a factor of 2 that results in the throughput of $\frac{1}{3}$. Figure 4.2(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 (2.13), i.e., $s = 5 > \lceil \frac{12}{4} \rceil = 3$, where the throughput of $\frac{1}{5}$ finally could be achieved with the derived task set $\Gamma'' = \{\tau_{1,1} = (3, 5, 0, 5), \tau_{2,1} = (6, 10, 10, 10), \tau_{3,1} = (10, 20, 20, 20), \tau_{3,2} = (10, 20, 30, 20), \tau_{4,1} = (7, 20, 40, 20), \tau_{4,2} = (7, 20, 50, 20), \tau_{5,1} = (5, 10, 60, 10), \tau_{6,1} = (3, 5, 70, 5)\}$ of eight IDP tasks and the task allocation ${}^5\Gamma'' = \{{}^5\Gamma''_1 = \{\tau_{4,1}, \tau_{1,1}\}, {}^5\Gamma''_2 = \{\tau_{4,2}, \tau_{2,1}\}, {}^5\Gamma''_3 = \{\tau_{6,1}\}, {}^5\Gamma''_4 = \{\tau_{3,1}, \tau_{3,2}\}, {}^5\Gamma''_5 = \{\tau_{5,1}\}\}$.

The approaches in [4,23], 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 [4] allocates the tasks in Γ to 5 processors with task

allocation ${}^5\Gamma_= \{{}^5\Gamma_1 = \{\tau_3\}, {}^5\Gamma_2 = \{\tau_4, \tau_5\}, {}^5\Gamma_3 = \{\tau_5, \tau_1\}, {}^5\Gamma_4 = \{\tau_6, \tau_2\}, {}^5\Gamma_5 = \{\tau_2\}\}$, where task τ_5 is allowed to migrate between π_2 and π_3 and task τ_2 is allowed to migrate between π_4 and π_5. In this task allocation, 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 [23] to restrict the task migrations. Under EDF-fm scheduling, this approach allocates the tasks in Γ to 5 processors with task allocation ${}^5\Gamma = \{{}^5\Gamma_1 = \{\tau_3\}, {}^5\Gamma_2 = \{\tau_4, \tau_5\}, {}^5\Gamma_3 = \{\tau_5, \tau_1\}, {}^5\Gamma_4 = \{\tau_6\}, {}^5\Gamma_5 = \{\tau_2\}\}$, where only task τ_5 is allowed to migrate between π_2 and π_3. Similar to the approach in [23], EDF-sh [92] allocates the tasks in Γ to 5 processors with task allocation ${}^5\Gamma = \{{}^5\Gamma_1 = \{\tau_3\}, {}^5\Gamma_2 = \{\tau_4, \tau_5\}, {}^5\Gamma_3 = \{\tau_5, \tau_1\}, {}^5\Gamma_4 = \{\tau_6\}, {}^5\Gamma_5 = \{\tau_2\}\}$, 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 algorithm 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 $\mathcal{M}$, i.e., the 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 4.1. Table 4.1 clearly shows that our proposed algorithm can reduce the number of required processors while keeping a low memory and latency increase compared to the related approaches for the same throughput requirement.

Let us now assume that the platform has only 4 processors. Then, all the related approaches, except EDF-sh, fail to satisfy the throughput requirement 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 satisfying the throughput requirement of $\frac{1}{5}$. EDF-sh can also allocate the tasks in Γ to 4 processors with task allocation ${}^4\Gamma = \{{}^4\Gamma_1 = \{\tau_3\}, {}^4\Gamma_2 = \{\tau_4, \tau_2\}, {}^4\Gamma_3 = \{\tau_2, \tau_5, \tau_1\}, {}^4\Gamma_4 = \{\tau_5, \tau_6\}\}$, 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

[3]When replicating an actor, the period of the task corresponding to the actor is enlarged. As a consequence, the production of data tokens that are required by its data-dependent tasks to execute are postponed which results in a further offsetting of their start time, when calculating the earliest start time of tasks in the SPS framework using Equation (2.16), hence increasing the application latency.

Table 4.1: *Throughput $\mathcal{R}$ (1/time units), latency $\mathcal{L}$ (time units), memory requirements $\mathcal{M}$ (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]	$\mathcal{M}$ [B]	$\check{\text{m}}$	$\check{\text{m}}_{\text{OPT}}$
EDF	FFD	1/5	55	155	6	4
	our	1/5	65 **(105)**	189 **(327)**	5 **(4)**	4
	FFD-EP [81]	1/5	75	228	5	4
EDF-*fm*	FFD-SP [23]	1/5	90	197	5	4
	LUF [4]	1/5	94	217	5	4
EDF-sh [92]		1/5	113 **(192)**	217 **(311)**	5 **(4)**	4

using our proposed algorithm and EDF-sh are given in the third and seventh rows of Table 4.1 in parenthesis. As a result, our proposed algorithm 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 algorithm which 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 requirement.

4.6 Proposed Algorithm

As explained and shown in Section 4.5, the partitioned scheduling algorithms, potentially, have 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 which is not sufficient to accommodate another task. Therefore, in this section, we present our novel algorithm 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, e.g., FFD-EDF scheduler. Our propose algorithm 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 algorithm is the following: our algorithm 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 algorithm 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 algorithm 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 actor in an SDF graph and the task allocation to reduce the number of required processors while satisfying a given throughput requirement.

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 actors 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 ${}^{x}\Gamma$ if a successful allocation, i.e., $x \leq |\Pi|$, is found. Otherwise, it returns false as output.

In Line 1, the algorithm initializes the replication factor of all actors in graph G to 1, G' to G, and Π' to Π. In Line 2, the actors in the graph G' are converted to periodic tasks using the SPS framework, explained in Section 2.3, where the minimum period T'_i of each task $\tau'_{i,k}$ corresponding to actor $A_{i,k}$ in G' is calculated for PE type of processors, i.e., using C_i^{PE}, by Equation (2.12) and Equation (2.13). *In this chapter, we take the maximum throughput of graph G, achievable by the SPS framework with the minimum calculated periods, as the throughput requirement. Note that we can set another throughput requirement 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 37, 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 4.4.1, and allocate the tasks on platform Π'.

In Line 5, a task allocation set ${}^{|\Pi'|}\Gamma$ is created, to keep the tasks allocated to each processor individually. *Please note that in sets Π' and ${}^{|\Pi'|}\Gamma$, 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 5, an empty task set Γ_1 is also defined to keep the candidate tasks for replication. In Lines 6 to 23, the algorithm allocates every task $\tau'_{i,k} \in \Gamma$ to one of the processors according

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

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

$\vec{f} = [1, 1, \cdots, 1]$; $G' \leftarrow G$; $\Pi' \leftarrow \Pi$;
Calculate period T'_i for PE type of processors for each task $\tau'_{i,k}$ corresponding to actor $A_{i,k}$ in G' by using Equation (2.12) and Equation (2.13);
$\Gamma \leftarrow$ Sort tasks corresponding to actors in G' in order of decreasing utilization;
while *True* **do**
  ${}^{|\Pi'|}\Gamma \leftarrow \{{}^{|\Pi'|}\Gamma_1, {}^{|\Pi'|}\Gamma_2, \cdots, {}^{|\Pi'|}\Gamma_{|\Pi'|}\}$; $\Gamma_1 \leftarrow \emptyset$;
  for $\tau'_{i,k} \in \Gamma$ **do**
    for $1 \leq j \leq |\Pi'|$ **do**
      if π_j *is an* EE *processor* **then**
        $u_{left} = \sum_{\ell=1}^{j-1} (1 - u_{\pi_\ell^{EE}})$; $u_i = u_i^{EE}$;
      if π_j *is a* PE *processor* **then**
        $u_{left} = \frac{C_i^{\text{PE}}}{C_i^{\text{EE}}} \sum_{\ell=1}^{|\Pi_{\text{EE}}|} (1 - u_{\pi_\ell^{EE}}) + \sum_{\ell=|\Pi_{\text{EE}}|+1}^{j-1} (1 - u_{\pi_\ell^{PE}})$; $u_i = u_i^{PE}$;
      Check EDF schedulability test on π_j;
      if *task* $\tau'_{i,k}$ *is not schedulable on* π_j **then continue**;
      else
        if $u_{\pi_j} = 0 \wedge u_{left} \geq u_i$ **then**
          if *actor* $A_{i,k}$ *corresponding to task* $\tau'_{i,k}$ *is not stateful/in/out* **then**
            $\Gamma_1 \leftarrow \Gamma_1 + \{\tau'_{i,k}, \pi_j\}$;
        ${}^{|\Pi'|}\Gamma_j \leftarrow \tau'_{i,k}$;
        break;
    if *task* $\tau'_{i,k}$ *is not allocated* **then**
      if $u_i > u_{left}$ **then return** False;
      $\Pi' \leftarrow \Pi' + \pi^{PE}$;
      go to 5
  for $|\Pi_{EE}| < j \leq |\Pi'|$ **do**
    if ${}^{|\Pi'|}\Gamma_j = \emptyset$ **then**
      $\Pi' \leftarrow \Pi' - \pi_j^{PE}$;
  if $|\Pi'_{PE}| \leq |\Pi_{PE}|$ **then break**;
  if $\Gamma_1 \neq \emptyset$ **then**
    $u_{left} = 0$;
    for $\{\tau'_{i,k}, \pi_j\} \in \Gamma_1$ **do**
      if $1 - u_{\pi_j} > u_{left}$ **then**
        $u_{left} = 1 - u_{\pi_j}$; $sel = i$;
  else return False;
  $f_{sel} = f_{sel} + 1$; $f_{sel} \in \vec{f}$;
  Get CSDF graph $G' = (\mathcal{A}', \mathcal{E}')$ by unfolding G with replication factors $\vec{f}$ using the method in Section 4.4.1;
  Calculate period T'_i for PE type of processors for each task $\tau'_{i,k}$ corresponding to actor $A_{i,k}$ in G' by using Equation (2.12) and Equation (2.13);
  $\Gamma \leftarrow$ Sort tasks corresponding to actors in G' in order of decreasing utilization;
return True, G', ${}^{|\Pi'|}\Gamma$;

to the FFD-EDF scheduler. In Lines 8 to 11, 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 9 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 11 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 11.

In Line 12, the EDF schedulability test [54] 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 13. Otherwise, the candidate tasks for replication are identified first in Lines 15 to 17. 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 15. *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 16, it is added in Line 17 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 18 and the procedure of vising the next processors is terminated in Line 19.

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 21. Otherwise, a PE processor is added to platform Π' in Line 22. 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 24 to 26, 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 26. Then, Line 27 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 38 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 28 to 32. 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 33. 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 31 and 32. Then, the replication factor of the actor corresponding to this task in the initial SDF graph is increased by one in Line 34 and the initial SDF graph is transformed into an equivalent CSDF graph using the unfolding transformation technique with unfolding vector $\vec{f}$, in Line 35. The periods of the tasks corresponding to actors in the obtained CSDF graph are calculated again for PE type of processors using Equation (2.12) and Equation (2.13) in Line 36 and the new periodic tasks are sorted in Γ in the order of decreasing utilization, in Line 37. 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.

4.7 Experimental Evaluation

In this section, we present the experiments to evaluate our proposed algorithm in Section 4.6. The experiments have been performed on a set of seven real-life streaming applications modeled as acyclic SDF graphs taken from [23]. These applications, from different application domains, are listed in Table 4.2. In this

Table 4.2: *Benchmarks used for evaluation taken from [23].*

Domain	Application	$\|\mathcal{A}\|$	$\|\mathcal{E}\|$
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

table, $|\mathcal{A}|$ and $|\mathcal{E}|$ denote the number of actors and FIFO communication channels in the corresponding SDF graph of an application.

To demonstrate the effectiveness and efficiency of our proposed algorithm, we perform two experiments. In the first experiment, in Section 4.7.1, we consider a homogeneous platform as considered in the related works [4,23,81]. In this experiment, we compare the application latency, the memory requirements, and the minimum number of processors needed to schedule the tasks of each application under a given throughput requirement 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 algorithm; (iii) partitioned EDF with the heuristic proposed in [81]; (iv) semi-partitioned EDF-*fm*, with the FFD-SP heuristic proposed in [23]; (v) semi-partitioned EDF-*fm*, with the LUF heuristic proposed in [4]; (vi) semi-partitioned EDF-sh [92]. These approaches are denoted in Table 4.3 with FFD, our, FFD-EP, FFD-SP, *fm*-LUF, and EDF-*sh*, respectively. In the second experiment, in Section 4.7.2, we consider heterogeneous platforms, including PE and EE processors, as considered in the related work [92]. In this experiment, we compare the application latency and the memory requirements needed to schedule the tasks of each application under a given throughput requirement obtained with partitioned EDF with our proposed heuristic algorithm and semi-partitioned EDF-sh [92] for different heterogeneous platforms. *Please note that we use the approach presented in [23] to handle data dependencies when using the scheduling/allocation approaches in [4,92] for comparison with our algorithm.* The throughput requirement $\mathcal{R}$ for each application, that is, the maximum achievable throughput under the SPS framework, is given in the second column in Table 4.3.

Table 4.3: *Comparison of different scheduling/allocation approaches.*

Benchmark	$\mathcal{R}\ [\frac{1}{\text{t.u.}}]$	OPT	Partitioned										Semi-partitioned								
			FFD			our			FFD-EP				FFD-SP			*fm*-LUF			EDF-sh		
		$\check{m}_{\text{OPT}}$	$\check{m}_{\text{FFD}}$	$\mathcal{M}_{\text{FFD}}$[B]	$\mathcal{L}_{\text{FFD}}$[t.u.]	$\check{m}_{\text{our}}$	$\frac{\mathcal{M}_{\text{our}}}{\mathcal{M}_{\text{FFD}}}$	$\frac{\mathcal{L}_{\text{our}}}{\mathcal{L}_{\text{FFD}}}$	$\check{m}_{\text{EP}}$	$\frac{\mathcal{M}_{\text{EP}}}{\mathcal{M}_{\text{FFD}}}$	$\frac{\mathcal{L}_{\text{EP}}}{\mathcal{L}_{\text{FFD}}}$	$\check{m}_{\text{SP}}$	$\frac{\mathcal{M}_{\text{SP}}}{\mathcal{M}_{\text{FFD}}}$	$\frac{\mathcal{L}_{\text{SP}}}{\mathcal{L}_{\text{FFD}}}$	$\check{m}_{\text{LUF}}$	$\frac{\mathcal{M}_{\text{LUF}}}{\mathcal{M}_{\text{FFD}}}$	$\frac{\mathcal{L}_{\text{LUF}}}{\mathcal{L}_{\text{FFD}}}$	$\check{m}_{\text{sh}}$	$\frac{\mathcal{M}_{\text{sh}}}{\mathcal{M}_{\text{FFD}}}$	$\frac{\mathcal{L}_{\text{sh}}}{\mathcal{L}_{\text{FFD}}}$
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

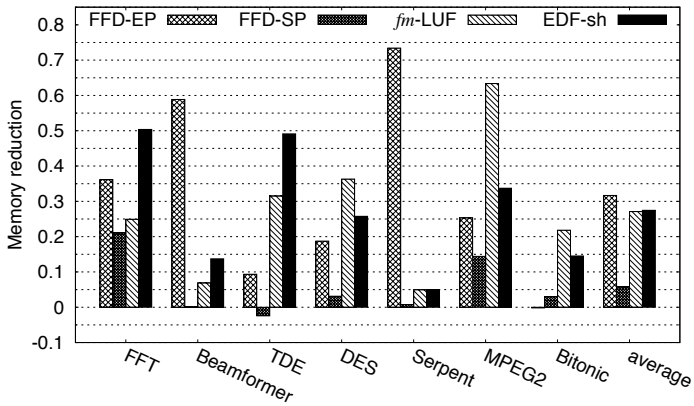
4.7.1 Homogeneous platform

Let us first compare our algorithm with the related approaches in terms of the number of required processors. The minimum number of required processors to satisfy the throughput requirement for each application using an optimal scheduler, denoted as $\check{m}_{OPT}$ and calculated using Equation (2.8), is given in the third column in Table 4.3. To find the minimum number of required processors using our proposed algorithm and the related approaches proposed in [4, 23, 81, 92], we set the number of PE processors on the homogeneous platform initially to $\check{m}_{OPT}$. 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.

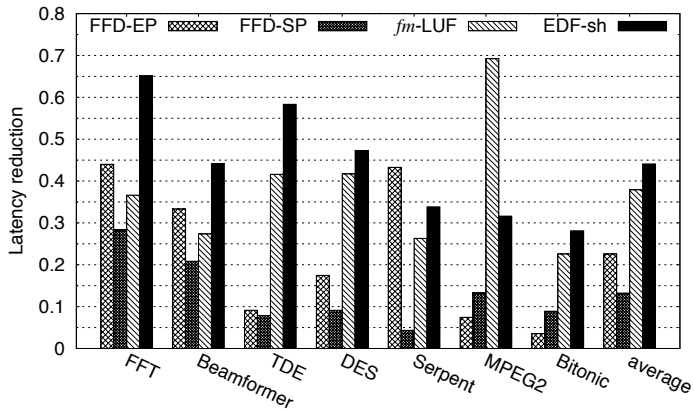
As can be seen in Table 4.3, the FFD approach requires considerably more processors, on average 17.6% more, than the number of required processors by an optimal scheduler, see column $\check{m}_{FFD}$. In contrast, our algorithm and EDF-sh require the same number of processors as the optimal scheduler while maintaining the same throughput for this set of applications, see columns $\check{m}_{our}$ and $\check{m}_{sh}$, respectively. For the other approaches, although they require fewer processors than FFD, they still require more processors than our algorithm for some applications. For instance, the approach FFD-EP requires one more processor for TDE, DES, and Serpent, see column $\check{m}_{EP}$; The approach FFD-SP requires two more processors for FFT and one more processor for DES and Serpent, see column $\check{m}_{SP}$; Finally the approach *fm*-LUF requires two more processors for FFT and DES and one more processor for TDE and MPEG2, see column $\check{m}_{LUF}$. Although this difference in terms of number of required processors is not too large, it clearly reveals that our algorithm is more capable of scheduling the applications with fewer processors compared to the FFD-EP, FFD-SP, and *fm*-LUF approaches while satisfying the same throughput requirement.

However, this reduction on the number of required processors comes at the expense of increased memory requirements and application latency. For each application, columns $\mathcal{M}_{FFD}$ 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 FIFO communication channels in the (C)SDF graph and the code size of the tasks. For each application, the increase on memory requirements and application latency by our algorithm over FFD are given in columns $\frac{\mathcal{M}_{our}}{\mathcal{M}_{FFD}}$ 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 algorithm achieves fewer processors compared to the related approaches, but also it imposes, on average, lower memory and latency overheads.



(a) Memory reduction



(b) Latency reduction

Figure 4.4: *Memory and latency reduction of our algorithm compared to the related approach with the same number of processors.*

To further compare our algorithm with the related approaches, we compute the memory requirements and application latency of our algorithm when equal number of processors as the related approaches are used, see the bolded numbers in parenthesis in columns $\check{\mathrm{m}}_{\mathrm{our}}$, $\frac{\mathcal{M}_{\mathrm{our}}}{\mathcal{M}_{\mathrm{FFD}}}$, and $\frac{\mathcal{L}_{\mathrm{our}}}{\mathcal{L}_{\mathrm{FFD}}}$. To ease the interpretation of Table 4.3 for this comparison, Figure 4.4(a) and Figure 4.4(b) illustrate the memory and latency reductions obtained by our algorithm compared to

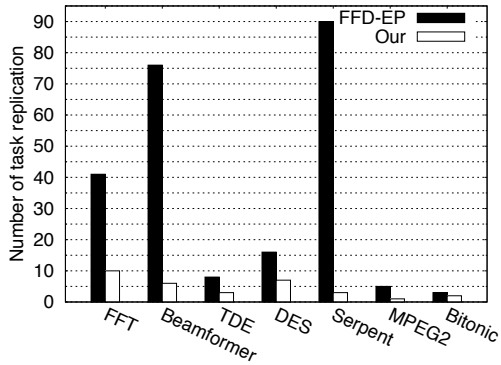


Figure 4.5: *Total number of task replications needed by FFD-EP and our proposed algorithm.*

the related approaches, respectively. For instance, the reduction on memory requirements is computed using the following equation:

$$r = \frac{\mathcal{M}_{\text{rel}} - \mathcal{M}_{\text{our}}}{\mathcal{M}_{\text{rel}}} \tag{4.3}$$

where $\mathcal{M}_{\text{rel}}$ is the memory requirements of scheduling an application using a related approach and $\mathcal{M}_{\text{our}}$ denotes the memory requirements achieved by our algorithm for the same number of processors. In Figure 4.4(a), we can see that our algorithm can reduce the memory requirements by an average of 31.43%, 5.72%, 27.11%, and 27.46% compared to FFD-EP, FFD-SP, *fm*-LUF, and EDF-sh, respectively. In Figure 4.4(a), however, there are two exceptions where our algorithm achieves 2.43% and 0.19% more memory for TDE and Bitonic compared to FFD-SP and FFD-EP, respectively. In Figure 4.4(b), we can also see that our algorithm can reduce the application latency considerably for all applications 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 applications our algorithm is more efficient than the related approaches in exploiting the available resources. Compared to FFD-EP, that is the closest approach to our algorithm as both 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 Figure 4.5. This figure clearly shows that, by replicating the right tasks, our proposed algorithm can reduce the total number of task replications significantly, by up to 30 times, compared to FFD-EP. From Figure 4.4, it can be also observed that our proposed algorithm 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 actors, the actors'

Table 4.4: *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

workload, the graph's topology, repetition vector, etc., the applications are represented with a different set of periodic tasks by using the SPS framework 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 4.4. As can be seen from Table 4.4, 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 algorithm is used at design-time 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 Figure 4.5.

4.7.2 Heterogeneous platform

To compare our proposed algorithm and EDF-sh [92] 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 $\breve{m}_{FFD}-1$ PE processors (see Table 4.3 for $\breve{m}_{FFD}$) and 1 EE processor for each application and iteratively replace one PE processor with one EE processor (or more EE processors

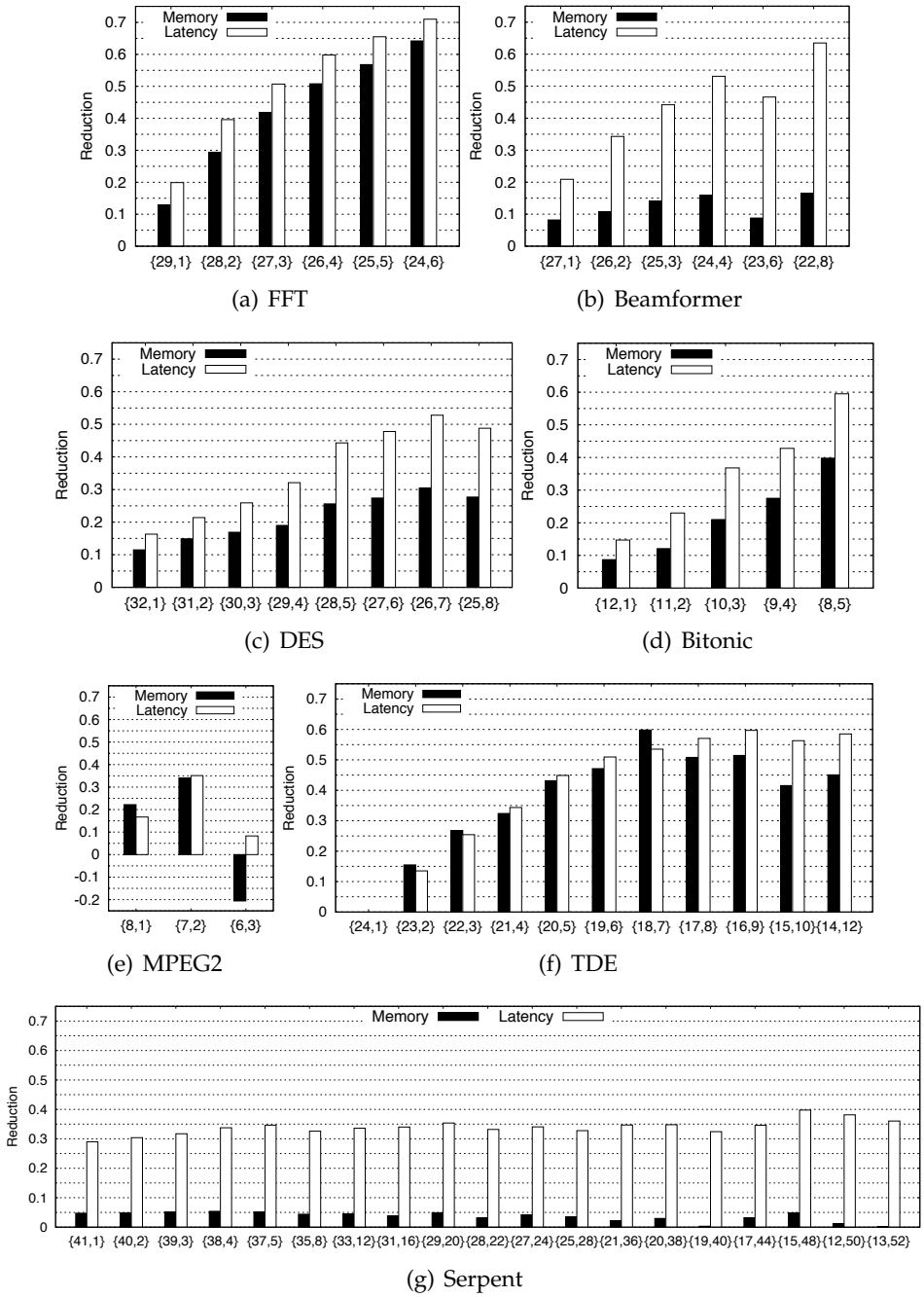


Figure 4.6: *Memory and latency reduction of our algorithm compared to EDF-sh [92] for real-life applications on different heterogeneous platforms.*

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 algorithm with EDF-sh on the heterogeneous platforms for which EDF-sh can successfully allocate the tasks for each application. Figure 4.6 shows the memory and latency reductions obtained by our algorithm compared to EDF-sh for each application individually. The reductions are computed using Equation (4.3). In Figure 4.6, 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.

From Figure 4.6, it can be observed that our proposed algorithm outperforms EDF-sh in terms of memory requirements and application latency for most of the cases. Compared to EDF-sh, our algorithm 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 applications, respectively. For the MPEG application, however, our proposed algorithm 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 algorithm 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 algorithm can still reduce the application latency by 8.3% compared to EDF-sh. For the TDE application, 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 Figure 4.6 that for most of the cases by replacing more PE processors with EE processors on the platform, our algorithm can further reduce the memory requirements and application latency compared to EDF-sh. 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 by our algorithm. 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 [23], the memory requirements increase 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.

4.8 Conclusions

In this chapter, we have presented a novel heuristic algorithm which determines a replication factor for each actor in an acyclic SDF graph, with a given throughput requirement, such that the number of processors needed to schedule the periodic tasks corresponding to actors 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 needed processors. The experiments on a set of real-life streaming applications show that our proposed algorithm can reduce the number of needed 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 satisfying the same throughput requirement. We also show that our algorithm can still reduce the number of needed 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 satisfying the same throughput requirement.