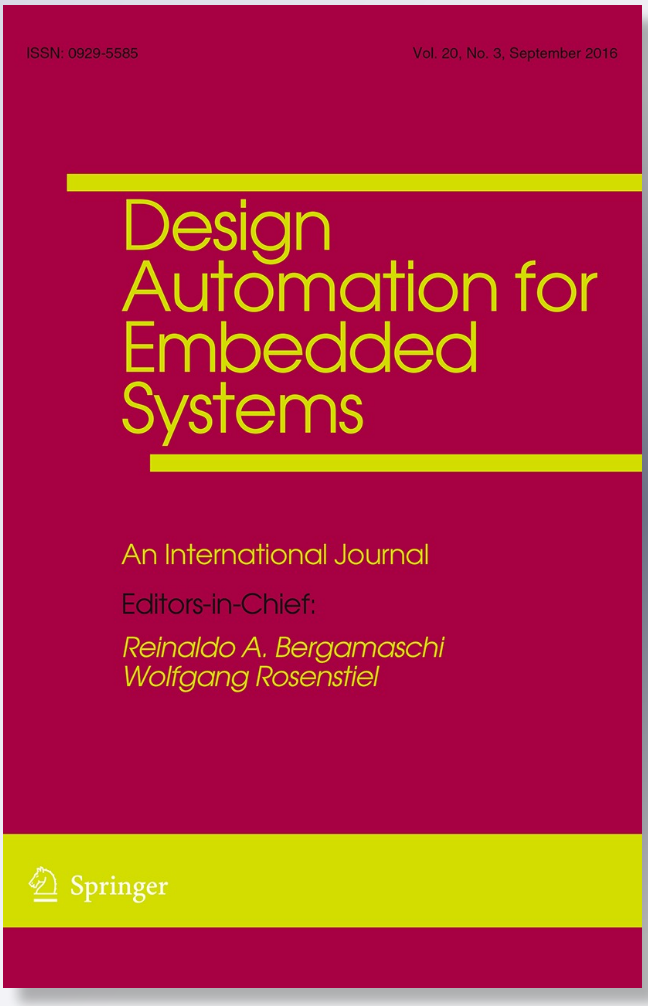
Energy efficient semi-partitioned scheduling for embedded multiprocessor streaming systems

Emanuele Cannella & Todor P. Stefanov

Design Automation for Embedded Systems
An International Journal

ISSN 0929-5585
Volume 20
Number 3

Des Autom Embed Syst (2016)
20:239-266
DOI 10.1007/s10617-016-9176-2



ISSN: 0929-5585 Vol. 20, No. 3, September 2016

Design Automation for Embedded Systems

An International Journal

Editors-in-Chief:

Reinaldo A. Bergamaschi
Wolfgang Rosenstiel

Springer

Springer

Des Autom Embed Syst (2016) 20:239–266
DOI 10.1007/s10617-016-9176-2

Energy efficient semi-partitioned scheduling for embedded multiprocessor streaming systems

Emanuele Cannella[1] · **Todor P. Stefanov**[1]

Received: 29 June 2015 / Accepted: 31 May 2016 / Published online: 16 June 2016

Abstract In this paper, we study the problem of energy minimization when mapping streaming applications with throughput constraints to homogeneous multiprocessor systems in which voltage and frequency scaling is supported with a discrete set of operating voltage/frequency modes. We propose a soft real-time semi-partitioned scheduling algorithm which allows an even distribution of the utilization of tasks among the available processors. In turn, this enables processors to run at a lower frequency, which yields to lower energy consumption. We show on a set of real-life applications that our semi-partitioned scheduling approach achieves significant energy savings compared to a purely partitioned scheduling approach and an existing semi-partitioned one, EDF-os, on average by 36 % (and up to 64 %) when using the lowest frequency which guarantees schedulability and is supported by the system. By using a periodic frequency switching scheme that preserves schedulability, instead of this lowest supported fixed frequency, we obtain an additional energy saving up to 18 %. Although the throughput of applications is unchanged by the proposed semi-partitioned approach, the mentioned energy savings come at the cost of increased memory requirements and latency of applications.

Keywords Energy efficient multiprocessor scheduling · Energy-efficient design · Real-time multiprocessor scheduling · Model-based design · Embedded streaming systems

1 Introduction

Modern Multiprocessor Systems-on-Chip (MPSoCs) offer ample amount of parallelism. In recent years, we have witnessed the transition from single core to multi-core and finally to

✉ Emanuele Cannella
emanuele.cannella@gmail.com

Todor P. Stefanov
t.p.stefanov@liacs.leidenuniv.nl

[1] Leiden Institute of Advanced Computer Science, Leiden University, Leiden, The Netherlands

many-core MPSoCs (e.g., [13]). Exploiting the available parallelism in these modern MPSoCs to guarantee system performance is a challenging task because it requires the designer to expose the parallelism available in the application and decide how to allocate and schedule the tasks of the application on the available processors. Another challenging task is to achieve energy efficiency of these MPSoCs. Energy efficiency is a desirable feature of a system, for several reasons. For instance, in battery-powered devices, energy efficiency can guarantee longer battery life. In general, energy-efficient design decreases heat dissipation and, in turn, improves system reliability.

To address the challenge of exploiting the available parallelism and guaranteeing system performance, Models-of-Computation (MoCs) such as Synchronous Dataflow (SDF) [18] are commonly used as a parallel application specification. Then, to derive a valid assignment and scheduling of the tasks of applications, recent works (e.g., [4]) have proposed techniques that derive periodic real-time task sets from the initial application specification. This derivation allows the designer to employ scheduling algorithms from real-time theory [9] to guarantee timing constraints and temporal isolation among different tasks and different applications, using fast schedulability tests. In contrast with [4], existing techniques that exploit the analysis of self-timed scheduling of SDF graphs to guarantee throughput constraints (e.g., [22]) necessitate a complex design space exploration (DSE) to determine the minimum number of processors needed to schedule the applications, and the mapping of tasks to processors. Based on the analysis of [4], a promising semi-partitioned approach [7] has been proposed recently to schedule streaming applications. In semi-partitioned scheduling most tasks are assigned statically to processors, while others (usually a few) can migrate. In [7], these migrations are allowed at job boundaries only, to reduce overheads. Semi-partitioned approaches which satisfy this property are said to have *restricted migrations*.

To address the energy efficiency challenge, mentioned above, many techniques to reduce energy consumption have been proposed in the past decade. These techniques exploit voltage/frequency scaling (VFS) of processors and have been applied to both streaming applications and periodic independent real-time tasks sets. VFS techniques can be either *offline* or *online*. Offline VFS uses parameters such as the worst-case execution time (WCET) and period of tasks to determine, at design-time, appropriate voltage/frequency modes for processors and how to switch among them, if necessary. Online VFS exploits the fact that at run-time some tasks can finish earlier than their WCET and determines, at run-time, the voltage/frequency modes to obtain further energy savings.

Problem statement To the best of our knowledge, the potential of semi-partitioned scheduling with restricted migrations together with VFS techniques to achieve lower energy consumption has not been completely explored. Therefore, in this paper, we study the problem of energy minimization when mapping streaming applications with throughput constraints using such semi-partitioned approach for homogeneous multiprocessor systems in which voltage and frequency scaling is supported with a discrete set of operating voltage/frequency modes.

1.1 Contributions

We propose a semi-partitioned scheduling algorithm, called Earliest Deadline First based semi-partitioned stateless (EDF-ssl), which is targeted at streaming applications where some of the tasks may be stateless. We derive conditions that ensure valid scheduling of the tasks of applications, under EDF-ssl, in two cases. First, when using the lowest frequency which guarantees schedulability and is supported by the system. Second, when using a periodic frequency switching scheme that preserves schedulability, which can achieve higher energy savings. In general, our EDF-ssl allows an even distribution of the utilization of tasks among the available processors. In turn, this enables processors to run at a lower frequency, which yields to lower power consumption. Moreover, compared to a purely partitioned scheduling approach, our experimental results show that our technique achieves the same application throughput with significant energy savings (up to 64 %) when applied to real-life streaming applications. These energy savings, however, come at the cost of higher memory requirements and latency of applications.

1.2 Scope of work

Assumptions In our work we make some assumptions that we describe and motivate below:

(1) We consider systems with distributed program and data memory to ensure predictability of the execution at run-time and scalability.

(2) We consider semi-partitioned scheduling, which is a hybrid between two extremes, *partitioned* and *global* scheduling. In partitioned scheduling, tasks are statically assigned to processors. Such scheduling algorithms allow no task migration, thus have low run-time overheads, but cannot efficiently utilize the available processors due to bin-packing issues [16]. In global scheduling (e.g., [5]), tasks are allowed to migrate among processors, which guarantees optimal utilization of the available processors but at the cost of higher run-time overheads and excessive memory overhead on distributed memory systems. This memory overhead is introduced because in distributed memory systems the code of all tasks should be replicated on all the available cores. As shown in [7], semi-partitioning can ameliorate the bin-packing issues of partitioned scheduling without incurring the excessive overheads of global scheduling.

(3) We assume that the system's communication infrastructure is predictable, i.e., it provides guaranteed communication latency. We include the worst-case communication latency when computing the WCET of a task. The WCET in our approach includes the worst-case time needed for the task's computation, the worst-case time needed to perform inter-task data communication on the considered platform and the worst-case overhead of the underlying scheduler as explained in Sect. 2.3.

Limitations The problem addressed in this paper, described earlier in the problem statement, is extremely complex. In order to make it more tractable, our approach considers certain limitations. However, we argue that even under these limitations many hardware platforms and applications can be handled by our proposed technique. In what follows, we list the limitations considered in our proposed approach.

(1) We assume that applications are modeled as acyclic SDF graphs. Although this assumption limits the scope of our work, our analysis is still applicable to the majority of streaming applications. In fact, a recent work [23] has shown that around 90 % of streaming applications can be modeled as acyclic SDF graphs.

(2) We use a VFS technique in which the voltage/frequency mode is changed globally over the considered set of processors. Our technique, therefore, finds applicability in two kinds of hardware platforms:

   - Hardware platforms that apply a single voltage/frequency mode to all the processors of the system (e.g., the OMAP 4460, as in [28]).
   - Hardware platforms that allow VFS management at the granularity of voltage islands (clusters), which may contain several cores (as in [13]). In these systems, our technique can be applied by considering each cluster as a separate sub-system with global VFS.

In the second kind of platforms listed above, our VFS technique can be used to complement existing approaches that derive an efficient partitioning of tasks to clusters, such as [8,17]. Note that our proposed technique does not consider per-core VFS, therefore it may be less beneficial for systems which support this kind of VFS granularity. However, per-core VFS is deemed unlikely to be implemented in next generation of many-core systems, due to excessive hardware overhead [10].

(3) Our technique uses *offline* VFS because we do not exploit the dynamic slack created at run-time by the earlier completion of some tasks. This choice is motivated by the following two reasons. (i) Online VFS may require VFS transitions for each execution of a task. Given that in our approach tasks execute periodically, with very short periods, online VFS would incur significant transitions overhead. For instance, the period of tasks in the applications that we consider can be as low as $100\,\mu$s. Since the VFS transition delay overhead of modern embedded systems is in the range of tens of μs [20], the overhead of online VFS would be substantial with such short task periods. (ii) Moreover, the existence of a global frequency for the whole voltage island renders *online* VFS less applicable. This is because online VFS would only be effective if *all* cores in the voltage island have dynamic slack at the same time.

1.3 Related work

Several techniques addressing energy minimization for streaming applications have already been presented. Among these, the closest to our work are [15,22,24]. Wang et al. [24] considers applications modeled as Directed Acyclic Graphs, applies certain transformation on the initial graph and then generates task schedules using a genetic algorithm, assuming *per-core* VFS. [22] assumes that applications are modeled as SDF graphs, and is composed of an offline and online VFS phases, to achieve energy optimization. As shown in Sect. 3, our approach exploits results from real-time theory that allow, in the presence of stateless tasks, to set the global system frequency to the lowest value which guarantees schedulability and is supported by the system. Both [24] and [22] cannot in general make the system execute at the lowest frequency that supports schedulability because they use pure partitioned assignment of tasks to processors and non-preemptive scheduling. Finally, [15] considers both per-core and global VFS but assumes applications modeled as Homogeneous SDF graphs, and that task mapping and the static execution order of tasks is given. By contrast, our approach handles a more expressive MoC and does not assume that the initial task mapping is given.

In addition, several techniques to achieve energy efficiency for systems executing periodic independent real-time tasks have been proposed. Among these techniques, the ones presented in [10] and [21] are closely related to our approach because they consider global VFS. The authors in [10] study the problem of energy minimization when executing a periodic workload on homogeneous multiprocessor systems. Their approach, however, considers pure partitioned scheduling. As we show in this paper, pure partitioned scheduling can not achieve the highest possible energy efficiency. In our approach, instead, we consider semi-partitioned scheduling and we show that this approach yields significant energy savings compared to a pure partitioned one. The authors in [21] also address the problem of energy minimization under a periodic workload with real-time constraints. However, their approach allows migration of tasks at any time and to any processor. Therefore, their approach considers global scheduling of tasks. As explained earlier, in distributed memory systems global task scheduling entails high overheads, in terms of required memory and number of required preemptions and migrations of tasks. Our approach considers semi-partitioned scheduling in order to reduce such overheads, while obtaining higher energy efficiency than pure partitioned approaches.

Similar to our work, other related approaches exploit task migration to achieve energy efficiency, such as [14] and [27]. In [14], the authors re-allocate tasks at run-time to reduce the fragmentation of idle times on processors. This in turn allows the system to exploit the longer idle times by switching the corresponding processors off. As explained earlier, in our approach we do not exploit run-time processor transitions to the off state because such transitions incur high overheads, especially when running dataflow tasks which have short periods.

The approach presented in [27] is closely related to ours because it leverages a semi-partitioned approach, where tasks migrate with a predictable pattern, to achieve energy efficiency. The author in [27] presents a heuristic to assign tasks to processors in order to obtain an improved load balancing. When tasks cannot entirely fit on one processor, they are split in two shares which are assigned to two different processors. Our work differs from [27] in two main aspects. First, we allow tasks with heavy utilization to be divided in more than two shares. This can yield to much higher energy savings compared to the technique proposed in [27]. Second, we allow job parallelism, i.e., we allow the concurrent execution on different processors of jobs of the same task. This, in turn, contributes to an improved balancing of the load among processors, which allows us to apply voltage and frequency scaling more effectively, as will be shown in Sect. 3. Moreover, the applicability of the analysis proposed in [27] to task sets with data dependencies, as in our case, is questionable. In fact, the semi-partitioned scheduling algorithm underlying [27] is identical to the one proposed by Anderson et al. in [1]. As the latter paper shows, under this semi-partitioned scheduling algorithm tasks can miss deadlines by a value called tardiness, even when VFS is not considered. Since in our case tasks communicate data, to guarantee that data dependencies among tasks are respected this tardiness must be analyzed. However, an analysis of task tardiness is not given by [27].

As mentioned earlier, the approach we propose in our paper exploits the concurrent execution on different processors of jobs of the same task. In a similar fashion, related work that exploit parallel execution of a task on different processors to achieve energy efficiency are [25] and [19]. In [25] the authors exploit the *data* parallelism available in the input application. That is, jobs of an application are divided in sub-jobs which process independent subsets of the input data. These sub-jobs can therefore be executed independently and concurrently on different processors, obtaining a more balanced load on processors, which in turn allows a more effective scaling of voltage and frequency of processors. The approach presented in [25], however, incurs a drawback in the case of distributed memory architectures. In fact, the mentioned sub-jobs of the application can be seen as separate instances of the input application, which execute independent chunks of input data. This means that, in distributed memory architectures, the code of the whole application has to be replicated on all the processors which execute these sub-jobs. By contrast, in our approach only certain tasks of the input application have to be replicated (only migrating tasks), which reduces significantly the memory overhead of our approach compared to the one in [25]. An approach similar to [25] has been proposed by the authors in [19]. The technique presented in [19] also divides computation-intensive tasks to sub-tasks which can be concurrently executed on multiple cores. As in [25], this yields to a more balanced load on processors, and in turn allows the system to run at a lower frequency. Moreover, the authors in [19] consider systems with discrete set of operating frequencies. Similar to our technique, when the lowest frequency which guarantees schedulability is not supported by the system, the analysis in [19] employs a processor frequency switching scheme to obtain this lowest frequency and still meet all deadlines. However, our analysis is different from [19] in several aspects. First, when assigning sub-tasks load to the available processors, [19] considers only *symmetric* distribution of the load of a task to different processors. In contrast, in our paper, as shown in Example 4 in Sect. 3, in order to obtain optimal energy savings we allow an asymmetric distribution of the load of certain tasks to the available processors. Second, two major differences concern the derivation of the periodic VFS switching scheme that guarantees schedulability. The first difference is that the analysis in [19] does not account for the overheads incurred when performing VFS transitions. By contrast, our analysis take this realistic overhead into account. The second difference is that in [19] such periodic VFS switching scheme is derived in order to meet *all* the deadlines of tasks. This requires the system to perform very frequent VFS transitions, especially when tasks have short periods as in our case. Conversely, in our approach we allow some task deadlines to be missed, by a bounded amount. This allows our approach to perform much fewer VFS transitions. As VFS transitions incur time and energy overhead in realistic systems, our approach guarantees higher effectiveness compared to [19].

The semi-partitioned scheduling that we propose, EDF-ssl, allows only restricted migrations. Notable examples of existing semi-partitioned scheduling algorithms with restricted migrations are EDF-fm [1] and EDF-os [2], from which our EDF-ssl inherits some properties, as explained in Sect. 2.4. The closest to our EDF-ssl is EDF-os because it allows migrating tasks to run on two or more processors, not strictly on two as in EDF-fm. The fundamental difference between EDF-os and our proposed EDF-ssl lays in the kind of applications that are considered by these two scheduling algorithms. In EDF-ssl we consider applications in which some of the tasks may be stateless and therefore can execute different jobs of the same task in parallel, if released on different processors. By contrast, EDF-os considers applications modeled as sets of tasks where all tasks are *stateful*. This means that different jobs of the same task cannot be executed concurrently. As explained in detail in Sect. 3, this fact prevents EDF-os from achieving energy-optimal results when streaming applications have stateless tasks with high utilization. This phenomenon is also described in the experimental results section (Sect. 5.3). Similar to our work, analyses of scheduling algorithms that allow jobs within a single task to run concurrently are presented in [12,26]. However, both these works consider global scheduling algorithms which, as mentioned earlier, entail high overheads especially in distributed memory architectures. In addition, in both [12] and [26] the potential of exploiting job parallelism to achieve higher energy efficiency is not explored.

2 Background

In Sects. 2.1 and 2.2 we introduce the system model and task set model assumed in our work, respectively. Then, we summarize techniques instrumental to our approach: soft real-time scheduling of acyclic SDF graphs (Sect. 2.3) and some properties of the EDF-os semi-partitioned approach which are leveraged in our work (Sect. 2.4).

2.1 System model

We consider a system composed of a set $\Pi = \{\pi_1, \pi_2, \ldots, \pi_M\}$ of M homogeneous processors. As explained in Sect. 1, we consider the problem of mapping applications to systems (or clusters) in which all the cores belong to the same voltage/frequency island. This means that any processor in the system either runs at the same "global" frequency and voltage level, or is idle. Each idle processor has no tasks assigned to it and consumes negligible power. We assume that the system supports only a discrete set $\Phi = \{F_1, F_2, \ldots, F_N\}$ of N operating frequencies, where the maximum frequency is $F_N = F_{\max}$. To ease the explanation of our analysis, based on this maximum frequency $F_{\max}$ we define the normalized system speed as follows.

Definition 1 *(Normalized speed)* Given a frequency F at which the system runs, this system is said to run at a *normalized system speed* $\alpha = F/F_{\max}$.

This definition creates a one-to-one correspondence between any frequency at which the considered system runs and its normalized speed. We will exploit this correspondence throughout this paper. Given the set of supported frequencies Φ, by applying Definition 1 we obtain a set of supported normalized system speeds $A = \{\alpha_1, \alpha_2, \ldots, \alpha_N\}$, where $\alpha_N = \alpha_{\max} = 1$.

2.2 Task set model

As shown in Sect. 2.3 below, the input application, modeled as an acyclic SDF graph with n actors, can be converted to a set $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n real-time periodic tasks. We assume that tasks can be preempted at any time. A periodic task $\tau_i \in \Gamma$ is defined by a 4-tuple $\tau_i = (C_i, T_i, S_i, \Delta_i)$, where C_i is the WCET of the task, T_i is the task period, S_i is the start time of the task, and Δ_i represents the task tardiness bound, as defined in Definition 2 below. Note that C_i is obtained at the maximum available processor frequency, $F_{\max}$. Therefore, we can derive the worst-case task execution requirement in clock cycles as $CC_i = C_i \cdot F_{\max}$. In this paper, we consider only *implicit-deadline* tasks, which have relative deadline D_i equal to their period T_i. The utilization of a task τ_i is given by $u(\tau_i) = C_i/T_i$ (also denoted by u_i). The cumulative utilization of the task set is denoted with $U_\Gamma = \sum_{\tau_i \in \Gamma} u(\tau_i)$.

The kth job of task τ_i is denoted by $\tau_{i,k}$. Job $\tau_{i,k}$ of τ_i, for all $k \in \mathbb{N}_0$, is released in the system at the time instant $r_{i,k} = S_i + kT_i$. The absolute deadline of job $\tau_{i,k}$ is $d_{i,k} = S_i + (k+1)T_i$, which is coincident with the arrival of job $\tau_{i,k+1}$. We denote the actual completion time of $\tau_{i,k}$ as $z_{i,k}$. Note that the conversion of the input application to a corresponding periodic task set, as described in Sect. 2.3, creates a one-to-one correspondence between actor v_i of the application and task $\tau_i \in \Gamma$. Similarly, there is a one-to-one correspondence between the kth invocation $v_{i,k}$ of v_i and job $\tau_{i,k}$ of τ_i. These correspondences will be exploited throughout this paper.

In this paper, we consider as soft real-time (SRT) those systems in which tasks are allowed to miss their deadline by a certain bounded value, called tardiness. The bound on tardiness is defined as follows.

Definition 2 *(Tardiness bound)* A task τ_i is said to have a *tardiness bound* Δ_i if $z_{i,k} \leq (d_{i,k} + \Delta_i), \forall k \in \mathbb{N}_0$.

Note that even if job $\tau_{i,k}$ has tardiness greater than zero, in our approach the release time of the next job $\tau_{i,k+1}$ is not affected. That is, job $\tau_{i,k+1}$ will be available to be scheduled although the previous job $\tau_{i,k}$ of task τ_i has not yet finished its execution. However, in such a case, if job $\tau_{i,k}$ and job $\tau_{i,k+1}$ are released *on the same processor* and the local scheduler is EDF (as we assume in this paper) job $\tau_{i,k+1}$ will always have to wait until the completion of job $\tau_{i,k}$ because this job has higher priority. This is because the deadline of job $\tau_{i,k}$ is by definition earlier than that of job $\tau_{i,k+1}$.

2.3 Soft real-time scheduling of SDF graphs

In this paper we consider applications modeled as acyclic SDF graphs [18]. An SDF graph G is composed of a set of actors V and a set of edges E, through which actors communicate.

We define as *input actor* of G an actor that receives the input stream of the application, and as *output actor* of G an actor that produces the output stream of the application. The authors in [4] show that the actors in any acyclic SDF graph can be scheduled as a set of real-time periodic tasks Γ, as defined in Sect. 2.2. Their analysis begins with the computation of the WCET C_i of an SDF actor v_i. The value of C_i is computed such that both the worst-case communication and computation of v_i is included:

$$C_i = C^R \cdot \sum_{e_u \in \mathrm{inp}(v_i)} y_i^u + C_i^C + C^W \cdot \sum_{e_r \in \mathrm{out}(v_i)} x_i^r \tag{1}$$

In Eq. (1), C^R/C^W represents the (platform-dependent) worst case time needed to read/write a single token from/to an input/output channel; y_i^u/x_i^r is the number of tokens read/written by actor v_i from/to edge e_u/e_r; $\mathrm{inp}(v_i)/\mathrm{out}(v_i)$ is the set of input/output edges of v_i; and C_i^C is the worst-case computation time of actor v_i. Note that C_i^C includes also the worst-case overhead incurred by the underlying scheduler (e.g., EDF), following the analysis of [11].

2.3.1 Derivation of minimum periods of tasks

Based on the WCETs of each actor computed by Eq. (1) and on the properties of the graph, the authors in [4] derive the minimum period T_i of each task τ_i (which corresponds to SDF actor $v_i \in V$), using Lemma 2 in [4]. These derived task periods ensure that each actor v_i executes q_i times in every *iteration period* H:

$$q_1 T_1 = q_2 T_2 = \cdots = q_n T_n = H \tag{2}$$

where q_i, derived from the properties of the SDF graph, is the number of repetitions of v_i per graph iteration [18] and n is the number of actors in the graph.

The technique presented in [4] has been recently extended by [7], which considers that the derived periodic task set is scheduled by an SRT scheduler with bounded task tardiness (see Definition 2). This extension is summarized in the following subsection.

2.3.2 Earliest start times and buffer size calculation

As long as tardiness is bounded by a value Δ_i for each task τ_i, earliest start times of each task τ_i can be derived (Lemma 1 in [7]). The earliest start times of task τ_i corresponds to the parameter S_i defined in Sect. 2.2. Similarly, minimum buffer sizes can be calculated (Lemma 2 in [7]). Earliest start times and minimum buffer sizes are derived such that, in the resulting schedule, every actor can be released strictly periodically, without incurring any buffer underflow or overflow. Note that the technique in [7] allows tasks to miss their deadlines by a bounded value, because it uses an SRT scheduling algorithm. However, in the analysis, the worst-case tardiness that may affect each task is considered to guarantee that data-dependencies among tasks are respected. Furthermore, [7] shows that applications achieve the same throughput under SRT and HRT schedulers. In addition, even under a SRT scheduling algorithm, [7] guarantees hard real-time behavior at the interfaces between the system and the environment, provided that the buffers which implement these interfaces are appropriately sized to compensate for the tardiness of input and output actors.

The complete formulas to derive earliest start times of tasks and minimum buffer sizes are not reported and explained here due to space limitations but can be found in [7]. However, the intuition behind these formulas is the following. Let us consider the data-dependent actors v_s (source) and v_d (destination) shown in Fig. 1. They exchange data tokens over edge e_u.

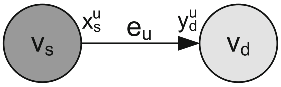


Fig. 1 SDF actors v_s and v_d with dependency over e_u

Every invocation of v_s produces x_s^u tokens to e_u, and every invocation of v_d consumes y_d^u tokens from e_u. To derive the earliest start time of the destination actor v_d in presence of task tardiness, Lemma 1 in [7] considers the worst case scheduling of the source and destination actors. This worst case scheduling, when deriving start times, occurs when the source actor v_s completes its jobs *as late as possible* (ALAP) and the destination actor v_d is released as soon as possible, with no tardiness. ALAP completion schedule in case of tardiness is defined below.

Definition 3 *(ALAP completion schedule in case of tardiness)* The ALAP completion schedule considers that all invocations $v_{i,j}$ (jobs $\tau_{i,j}$) of an actor v_i (task τ_i) incur the maximum tardiness Δ_i, therefore complete at $z_{i,k} = d_{i,k} + \Delta_i$.

The ALAP completion schedule of actor v_s can be represented by a fictitious actor $\tilde{v_s}$, which has the same period as v_s, no tardiness, and start time $\tilde{S_s} = S_s + \Delta_s$. At run-time, any invocation of v_s, even if delayed by the maximum allowed tardiness Δ_s, will never complete later than the corresponding invocation of $\tilde{v_s}$. Then, the earliest start time of the destination actor v_d can be calculated considering $\tilde{v_s}$ as source actor.

Once the start times of source and destination actors have been calculated, Lemma 2 in [7] allows the derivation of the required size of the buffer that implements the communication over e_u. Lemma 2 in [7] considers the worst case scheduling for buffer size derivation. This occurs when the source actor executes as soon as possible, with no tardiness, and the destination actor completes its jobs as late as possible. Similarly to the analysis used to derive start times, the ALAP schedule of destination actor v_d can be represented by a fictitious actor $\tilde{v_d}$ which has the same period as v_d, no tardiness, and start time $\tilde{S_d} = S_d + \Delta_d$. Then, the buffer size can be derived considering v_s as source actor and $\tilde{v_d}$ as destination actor.

Example 1 Consider the SDF graph shown in Fig. 2a, which has three actors (v_1, v_2, v_3) with WCET indicated between parentheses ($C_1=2$, $C_2=3$, $C_3=2$) and production/consumption rates indicated above the corresponding edges. Using Lemma 2 in [4], we derive the following minimum periods: $T_1=T_3=6$ and $T_2=3$, as shown in Fig. 2b. Then, suppose that the underlying SRT scheduling algorithm guarantees tardiness bounds $\Delta_1=1$, $\Delta_2=2$ (as indicated in Fig. 2a and shown in Fig. 2b), whereas $\Delta_3=0$. By Lemma 1 in [7], using these tardiness bounds, we derive the earliest start times S_i shown in Fig. 2b. For instance, note that $S_2=7$ ensures that any invocation of v_2 will always have enough data to read as soon as it is released. This holds even when all the invocations of v_1 incur the largest tardiness Δ_1, i.e., they execute according to the ALAP completion schedule.

Note that, similar to our approach, the work in [7] also uses semi-partitioned scheduling, in which some tasks are assigned to a single core (*fixed tasks*) whereas some others may migrate between cores (*migrating tasks*). Only stateless tasks are allowed to migrate. The definition of stateless task is given below.

Definition 4 *(Stateless task)* A task τ_i is said to be stateless when it does not keep an internal state between two successive jobs.

The authors in [7] allow only stateless tasks to migrate because the overhead incurred in moving a (potentially large) internal state can be high, in distributed memory systems.

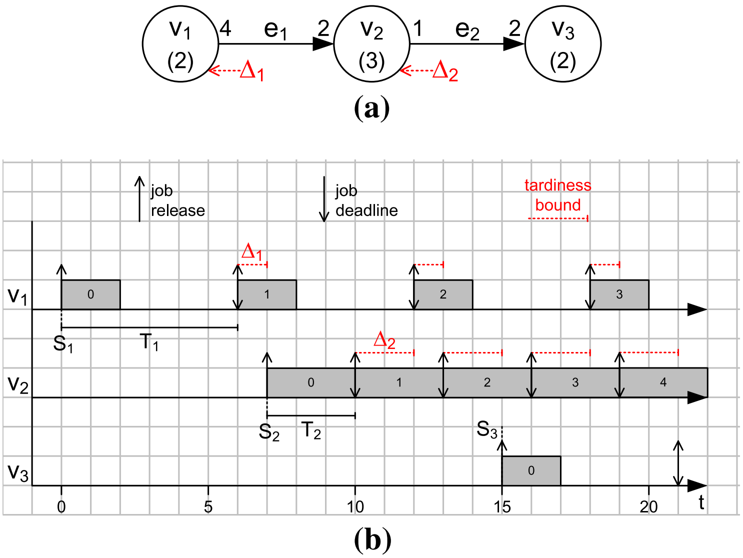


Fig. 2 Example of the approach described in [7]. **a** Simple example of an SDF graph. **b** Derived periodic task set and minimum start times. *Up arrows* represent job releases, down arrows represent task *deadlines*. Only the first period of v_3 is shown due to space constraints

2.4 EDF-os semi-partitioned algorithm

Our scheduling algorithm, presented in Sect. 3, inherits some definitions and properties from EDF-os [2]. Far from being a complete description of the EDF-os algorithm, the rest of this subsection presents this "common ground" between our approach and EDF-os.

EDF-os is aimed at soft real-time systems, in which tasks may miss deadlines, but only up to a bounded value. Under EDF-os, processors are assumed to run at the highest available frequency, which means that any processor π_k can handle a total cumulative utilization up to 1. Tasks can be either *fixed* or *migrating*. Migrating tasks are allowed to migrate between any number of processors, with the restriction that migration can only happen at job boundaries. Each task τ_i is assigned a (potentially zero) *share* of the available utilization of a processor, as defined below.

Definition 5 *(Task share)* A task τ_i is said to have a share $s_{i,k}$ on π_k when a part $s_{i,k}$ of its utilization u_i is assigned to π_k.

In turn, the *task fraction* of task τ_i on processor π_k is defined as follows.

Definition 6 *(Task fraction)* Given $s_{i,k}$, π_k executes a fraction $f_{i,k} = \frac{s_{i,k}}{u_i}$ of τ_i's total execution requirement.

If task τ_i is migrating, it has non-zero shares on several processors. If τ_i is fixed, it has non-zero shares on a single processor. The share assignment in EDF-os ensures that the cumulative sum of the shares of a task, over all the processors, equals the task utilization $u_i = \sum_{k=1}^{M} s_{i,k}$, with M being the total number of processors in the system.

The total share allocation on processor π_k is denoted by $\sigma_k \triangleq \sum_{\tau_i \in \Gamma} s_{i,k}$. In order to avoid overloading processor π_k in the long run, σ_k must always be lower than the total processor utilization:

$$\sigma_k \leq 1.0 \tag{3}$$

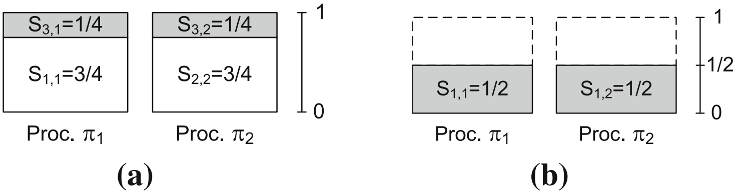


Fig. 3 Share assignments considered in Example 2 and Example 3. Migrating tasks are indicated in *gray*. **a** Share assignment considered in Example 2. **b** Share assignment considered in Example 3

In addition, to avoid overloading a processor in the long run, EDF-os enforces that, in the long run, the fraction of workload executed on π_k is equal to the task fraction $f_{i,k}$ given by Definition 6. This long-run workload distribution according to task fractions is obtained by leveraging results from Pfair scheduling [5]. In particular, out of the *first* ν consecutive jobs released by τ_i, EDF-os ensures that the number of jobs released on processor π_k is between $\lfloor f_{i,k} \cdot \nu \rfloor$ and $\lceil f_{i,k} \cdot \nu \rceil$ (Property 1 in [2]). Note that for $\nu \to \infty$ the fraction of jobs released on π_k tends to $f_{i,k}$, as expected. In turn, out of *any* c consecutive jobs of a migrating task τ_i, the number of jobs released on π_k (indicated as $c_{i,k}$) is bounded by the following expression:

$$c_{i,k} \leq f_{i,k} \cdot c + 2 \tag{4}$$

The above expression is given by Property 6 in [2]. For a more detailed explanation of assignment rules for jobs of migrating tasks, the reader is referred to [1] and [2].

Example 2 Given the task set $\{\tau_1 = (C_1 = 3, T_1 = 4), \tau_2 = (3, 4), \tau_3 = (1, 2)\}$, the EDF-os algorithm derives the task assignment shown in Fig. 3a. The utilization of task τ_3 in Fig. 3a is split in two shares, $s_{3,1} = 1/4$ on π_1 and $s_{3,2} = 1/4$ on π_2. Therefore, in the long run half of the jobs of τ_3 will be released on π_1 and the other half will be released on π_2.

3 Proposed semi-partitioned algorithm: EDF-ssl

In this section we describe our proposed semi-partitioned scheduler, called EDF-ssl. In EDF-ssl, only stateless tasks (recall Definition 4) are allowed to be migrating. We enforce this condition because migrating the internal state of a stateful task can be prohibitive in a distributed memory system. Note that under EDF-ssl task migrations can only happen at job boundaries. Once a job is released on a certain processor, it cannot migrate to another one. Moreover, EDF-ssl exploits the fact that migrating tasks are stateless by allowing successive jobs to execute in parallel on different processors.

With our EDF-ssl we want to show that, in the presence of stateless tasks, semi-partitioned scheduling can be used to improve energy efficiency, while achieving the same application throughput compared to purely partitioned scheduling. To achieve better energy efficiency it may be beneficial to run processors at voltage/frequency levels lower than the maximum. The following example shows that under certain conditions the classical partitioned VFS techniques (e.g., [3]) are not effective. Moreover, existing semi-partitioned approaches do not exploit the presence of some stateless tasks in the considered applications and therefore cannot be applied to achieve energy efficiency, if these stateless tasks have high utilization.

Example 3 Consider a single stateless task $\tau_1 = (C_1 = 3, T_1 = 3)$. The task utilization is $u_1 = 1$. In this case, existing partitioned VFS techniques can not be effective, because

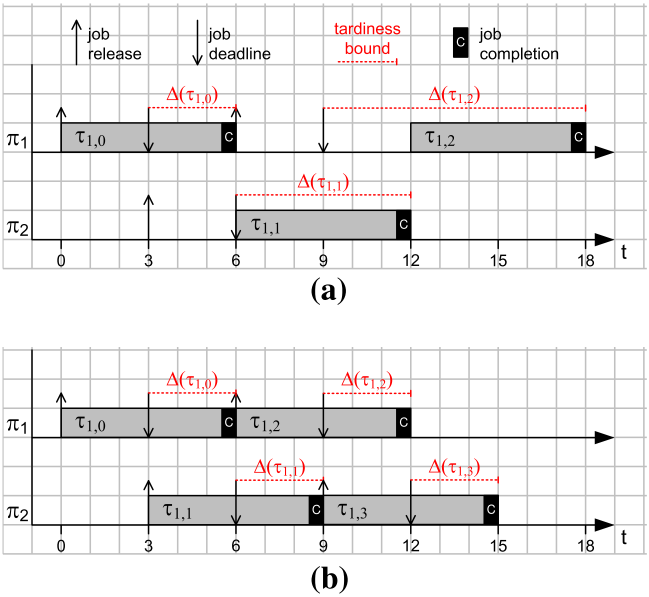


Fig. 4 Job executions of τ_1, as defined in Example 3, according to the share assignment of Fig. 3b. *Up arrows* indicate job releases, *down arrows* indicate job deadlines. *Black rectangles* indicate job completion. **a** Job executions according to EDF-os rules. **b** Job executions according to EDF-ssl rules

τ_1 can only be assigned to one processor and this processor must run at its highest voltage/frequency level, because $u_1 = 1$. Moreover, even existing semi-partitioned approaches cannot distribute the utilization of τ_1 over more than one processor, as shown in the following. Assume that to improve energy efficiency the utilization of τ_1 has to be split over two cores, π_1 and π_2, running at half of the maximum frequency, i.e., at normalized processors speed $\alpha = 1/2$. Note that under these conditions Eq. (3) has to be changed accordingly. We enforce therefore $\sigma_1 \leq \alpha$ and $\sigma_2 \leq \alpha$. The resulting assignment of shares of τ_1 is shown in Fig. 3b.

In this scenario, the problem of EDF-os is that it does not consider job parallelism. This means that job $\tau_{i,k+1}$ of a migrating task τ_i has to wait for the completion of the previous job $\tau_{i,k}$. For instance, in Fig. 4a, job $\tau_{1,0}$ is released on π_1 at time 0. Since $\alpha = 1/2$, $\tau_{1,0}$ finishes at time 6. Therefore job $\tau_{1,1}$, although released at time 3 on π_2, has to wait until time 6 to start executing. As shown in Fig. 4a, although jobs of τ_1 are assigned alternatively to π_1 and π_2, the tardiness Δ incurred by successive jobs of τ_1 increases unboundedly. Our EDF-ssl avoids this linkage between processors by allowing jobs released by a migrating task to execute in parallel, exploiting the fact that migrating tasks are assumed to be stateless. As depicted in Fig. 4b, this leads to bounded tardiness for all jobs of τ_1.

Under our EDF-ssl, necessary (but not sufficient) conditions to guarantee schedulability are the following. First, the total utilization of the task set Γ cannot be higher than the total available utilization on processors: $U_\Gamma \leq \alpha \cdot M$, where M is the number of available processors in the system and assuming that they all run at the same normalized speed $\alpha \leq 1$. Second, α must be greater than the utilization of any stateful task in Γ: $\alpha \geq u_{\text{s,max}}$, where $u_{\text{s,max}}$ is the utilization of the heaviest stateful task in Γ. This is because stateful tasks are fixed, and any processor to which the utilization $u_{\text{s,max}} > \alpha$ is assigned will be overloaded.

We merge the above two conditions in the following expression, which provides necessary higher and lower bounds for α:

$$\max\{U_\Gamma/M,\ u_{\mathrm{s,max}}\} \leq \alpha \leq 1 \tag{5}$$

We now proceed with a detailed description of our EDF-ssl. As in all semi-partitioned approaches (e.g., [1,2]), EDF-ssl is composed of two phases, an assignment phase and an execution phase, which are described in Sects. 3.1 and 3.2, respectively. Tardiness bounds guaranteed under EDF-ssl are derived in Sect. 3.3, for the case of processors running at a fixed normalized speed α. Finally, Sect. 3.4 presents a processor speed switching technique, called "Pulse Width Modulation (PWM) scheme", that provides a certain normalized speed in the long run. Tardiness bounds are derived also for the latter scenario.

3.1 Assignment phase

The assignment phase of EDF-ssl is given in Algorithm 1. It consists mainly of 3 steps, which we explain below. Note that under EDF-ssl processors can run at a normalized speed α lower than 1. Therefore, to avoid overloading processors in the long run, we modify condition (3) as follows:

$$\sigma_k \leq \alpha, \ \forall \pi_k \in \Pi \tag{6}$$

which means that the total share assignment on any processor π_k cannot exceed its normalized speed. Moreover, note that executing Algorithm 1 makes only sense if condition (5) is satisfied.

First step (lines 1–5) In this step, the algorithm finds the set of stateful tasks Γ_s within the original task set Γ. Then, it uses the First-Fit Decreasing Heuristic (FFD) [16] to allocate these stateful tasks as *fixed* tasks over the available processors. This means that if $\tau_i \in \Gamma_s$ is assigned to processor π_k, its share on π_k should be equal to the whole task utilization: $s_{i,k} = u_i$ and $s_{i,l} = 0, \forall l \neq k$.

Second step (lines 6–10) This step tries to assign all the remaining (stateless) tasks as *fixed* tasks over the remaining available processor utilization, using FFD. The tasks which can not be assigned as fixed are added to a set of tasks Γ_{na}, which are assigned in the next step.

Third step (lines 11–17) The final step assigns all the remaining tasks, which could not be allocated as fixed tasks. Considering the processor list in reversed order $\{\pi_M, \pi_{M-1}, \ldots \pi_1\}$, task $\tau_i \in \Gamma_{\mathrm{na}}$ is allocated a share on successive processors, considering the remaining utilization on each processor, in a sequential order. (The remaining utilization on processor π_k is given by $(\alpha - \sigma_k)$). The assignment of task τ_i finishes when the sum of its shares over the processors equals the task utilization u_i. The third step considers the processor list in reversed order as a way to minimize the number of processors, which already have fixed tasks, that are utilized to assign migrating shares. This can lead to a lower number of tasks with tardiness.

Example 4 Consider the SDF graph example in Fig. 2a. In Example 1, we derived the corresponding task set $\Gamma = \{\tau_1 = (2, 6), \tau_2 = (3, 3), \tau_3 = (2, 6)\}$. The total utilization of the task set is $U_\Gamma = 1/3 + 1 + 1/3 = 5/3$. Assume that we want to execute this task set on $M = 3$ processors. By condition (5), $\alpha \geq U_\Gamma/M = 5/9$, therefore the lowest α which could provide schedulability is $\alpha_{\mathrm{opt}} = 5/9$. Running the system at this lowest speed α_{opt} minimizes the energy consumption. Now, if the system supports the speed α_{opt}, we can simply set the system speed to that value. In this case, we can derive tardiness bounds using the result in Sect. 3.3, which considers fixed processors speed.

However, suppose that the considered system supports a set of normalized speeds $A = \{0.25, 0.5, 0.75, 1\}$. Note that $\alpha_{\mathrm{opt}} \notin A$. In this case, we have two choices. *Choice 1)* We set

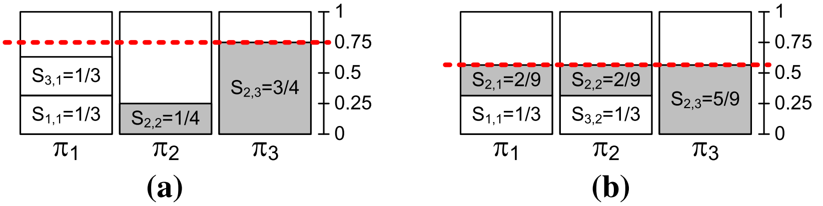


Fig. 5 Share assignments considered in Example 4. Values of α are shown in *red*. **a** $\alpha = 0.75$. **b** $\alpha = \alpha_{\text{opt}} = 5/9$. (Color figure online)

Algorithm 1: Share assignment heuristic.

```
Input: A set of $M$ processors $\Pi = \{\pi_1, \pi_2, \ldots, \pi_M\}$, their normalized speed $\alpha$, a set of $n$ periodic
       tasks $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$.
Result: An $M$-partition describing the share assignment onto $M$ processors if $\Gamma$ is schedulable, False
        otherwise.
Find $\Gamma_s = \{\tau : \tau \in \Gamma \wedge \tau$ is stateful$\}$;
for $\tau_i \in (\Gamma_s$, sorted by decreasing utilization$)$ do
    Try to assign $s_{i,k} = u_i$ of task $\tau_i$ on a single $\pi_k$ using FF;
    if FF fails for all $\pi_k \in \Pi$ then
        return False;

$\Gamma_{\text{na}} = \emptyset$ (the set of unassigned tasks, initially empty)
for $\tau_i \in (\Gamma - \Gamma_s$, sorted by decreasing utilization$)$ do
    Try to assign $s_{i,k} = u_i$ of task $\tau_i$ on $\pi_k$ using FF;
    if FF fails for all $\pi_k \in \Pi$ then
        $\Gamma_{\text{na}} = \Gamma_{\text{na}} \cup \tau_i$;

k = M (start share assignment from processor $\pi_M$ to $\pi_1$);
for $\tau_i \in \Gamma_{na}$ do
    $u_{\text{remaining}} = u_i$;
    while $u_{remaining} > 0$ do
        $s_{i,k} = \min(u_{\text{remaining}}, (\alpha - \sigma_k))$;
        $\sigma_k, u_{\text{remaining}} = (\sigma_k + s_{i,k}), (u_{\text{remaining}} - s_{i,k})$;
        if $\sigma_k = \alpha$ then
            k = k-1
```

the system speed to the lowest $\alpha \in A$ such that $\alpha > \alpha_{\text{opt}}$, condition which could provide schedulability: $\alpha = 0.75$. We can then refer again to Sect. 3.3 to derive tardiness bounds in this scenario. Fig. 5a shows the share assignment of tasks in Γ, when $\alpha = 0.75$ and assuming that input and output actors (τ_1, τ_3) are stateful. *Choice 2)* We use the periodic speed switching technique described in Sect. 3.4 to get the normalized speed α_{opt} in the long run, and we derive the corresponding tardiness bounds. Fig. 5b shows the assignment obtained when $\alpha = \alpha_{\text{opt}} = 5/9$.

3.2 Execution phase

At run-time, EDF-ssl follows the simple rules defined below.

Job releasing rules Jobs of a fixed task τ_f are released periodically, every T_f, on a single processor. Jobs of a migrating task τ_m are distributed over all the processors on which τ_m has non-zero shares. Our EDF-ssl inherits from EDF-os the job releasing techniques for migrating tasks (see Sect. 2.4). In particular, Eq. (4), which provides an upper bound of the number of jobs released on a processor as a function of the migrating task share, is still valid. This result will be instrumental to the derivation of tardiness bounds under our EDF-ssl.

Job prioritization rules As mentioned before, jobs of fixed and migrating tasks released on a certain processor are scheduled using a local EDF scheduler. As shown in Example 3, under our EDF-ssl when a task migrates from a processor to another one, the job released on the latter processor does not wait until the completion of the job released on the former processor. This is in contrast with what happens under EDF-os. Moreover, contrary to our EDF-ssl, under EDF-os certain tasks are statically prioritized over others.

3.3 Tardiness bounds under fixed processor speed

Given the rules and properties of our EDF-ssl, described in Sects. 3.1 and 3.2, we now derive its tardiness bounds, which are provided by Theorem 1 below. Note that due to the way task shares are assigned in the third step of the assignment phase, each processor runs at most two migrating tasks.

Theorem 1 *Consider a processor π_k running at a fixed normalized speed α. Assume two migrating tasks, τ_i and τ_j, are assigned to π_k. Then, jobs of fixed and migrating tasks released on π_k may incur a tardiness of at most*

$$\Delta^{\pi_k} = \frac{2\,(C_i + C_j)}{\alpha} \tag{7}$$

where C_i and C_j are the worst-case execution time of τ_i and τ_j, respectively, and α follows Definition 1.

Proof We prove Theorem 1 by contradiction. We focus on a certain job $\tau_{q,l}$, belonging to either a fixed or a migrating task, assigned to π_k . Let assume that this job incurs a tardiness which exceeds Δ^{π_k}. We define the following time instants to assist the analysis: $\mathbf{t_d}$ is the absolute deadline of job $\tau_{q,l}$; $\mathbf{t_c} = t_d + \Delta^{\pi_k}$; and $\mathbf{t_0}$ is the latest instant before t_c such that no migrating or fixed job released before t_0 with deadline at most t_d is pending at t_0. By definition of t_0, just before t_0 π_k is either idle or executing a job with deadline later than t_d. Moreover, t_0 cannot be later than $r_{q,l}$, the release time of job $\tau_{q,l}$. Note that since we assume that job $\tau_{q,l}$ incurs a tardiness exceeding Δ^{π_k}, it follows that $\tau_{q,l}$ does not finish at or before t_c.

We denote as γ the total set of tasks, fixed and migrating, assigned to π_k. We first determine the demand placed on π_k by γ in the time interval $[t_0, t_c)$. By the definitions of t_0, t_d, and t_c, any job of any task that places a demand in $[t_0, t_c)$ on π_k is released at or after t_0 and has a deadline at or before t_d. Therefore, the demand of any task τ_i in $[t_0, t_c)$ is given by the number of jobs released in this interval multiplied by the job execution time.

The number of jobs released on π_k in $[t_0, t_c)$, by a *fixed* task τ_f, is at most $c = \lfloor \frac{t_d - t_0}{T_f} \rfloor$ because fixed tasks release all of their jobs on π_k. By contrast, a *migrating* task τ_m releases $c = \lfloor \frac{t_d - t_0}{T_m} \rfloor$ jobs, but only part of them are assigned to π_k. An upper bound of the amount of jobs assigned to π_k, out of every c consecutive jobs, is given by Eq. (4).

We can now compute the total demand from tasks assigned to π_k. We denote as γ_f and γ_m the fixed and migrating sets of tasks mapped on π_k, respectively. Note that $\gamma_m = \{\tau_i, \tau_j\}$.

Given the total number of released jobs c, from Eq. (4) the demand *dmd* from migrating tasks in $[t_0, t_c)$ is upper bounded by:

$$\begin{aligned}\mathrm{dmd}(\gamma_m, t_0, t_c) &\le \left(f_{i,k}\left\lfloor \frac{t_d - t_0}{T_i}\right\rfloor + 2\right) C_i + \left(f_{j,k}\left\lfloor \frac{t_d - t_0}{T_j}\right\rfloor + 2\right) C_j \\ &\le (t_d - t_0)\left(f_{i,k}\frac{C_i}{T_i} + f_{j,k}\frac{C_j}{T_j}\right) + 2(C_i + C_j)\end{aligned}$$

Given the definition of $f_{i,k}$ in Definition (6), we obtain:

$$\mathrm{dmd}(\gamma_m, t_0, t_c) \le (t_d - t_0)(s_{i,k} + s_{j,k}) + 2(C_i + C_j) \tag{8}$$

At the same time, the demand from fixed tasks in $[t_0, t_c)$ is upper bounded by:

$$\mathrm{dmd}(\gamma_f, t_0, t_c) \le \sum_{\tau_f \in \gamma_f} \left\lfloor \frac{t_d - t_0}{T_f} \right\rfloor C_f \le (t_d - t_0) \sum_{\tau_f \in \gamma_f} \frac{C_f}{T_f}$$

From condition (6), we obtain:

$$\mathrm{dmd}(\gamma_f, t_0, t_c) \le (t_d - t_0)(\alpha - s_{i,k} - s_{j,k}) \tag{9}$$

Combining Eq. (8) and (9), we derive an upper bound for the total demand of fixed and migrating tasks in $[t_0, t_c)$:

$$\mathrm{dmd}(\gamma_f \cup \gamma_m, t_0, t_c) \le \alpha(t_d - t_0) + 2(C_i + C_j) \tag{10}$$

To ease our analysis, we now express the total demand from tasks in clock cycles. Recall that any requirement in processor time can be converted to clock cycles. For instance, for any task τ_a, its worst-case clock cycles requirement is $CC_a = C_a \cdot F_{\max}$ (see Sect. 2.1). Then, from Eq. (10) we get:

$$\mathrm{dmd\_cc}(\gamma_f \cup \gamma_m, t_0, t_c) \le F_{\max}\left(\alpha(t_d - t_0) + 2(C_i + C_j)\right) \tag{11}$$

Now, from our initial assumption that the tardiness of job $\tau_{q,l}$ exceeds Δ^{π_k}, it follows that the amount of clock cycles provided by the processor in the interval $[t_0, t_c)$ is less than the total demand from tasks *dmd_cc* in the same time interval. In the considered interval, the total demand from tasks is upper bounded by Eq. (11), whereas the amount of clock cycles provided by processor π_k is $F_{\max}\alpha(t_c - t_0)$, because π_k runs at frequency $F_{\max}\alpha$. Therefore, we have:

$$F_{\max}\alpha(t_c - t_0) < F_{\max}\left(\alpha(t_d - t_0) + 2(C_i + C_j)\right) \tag{12}$$

Dividing both sides by $F_{\max}\alpha$:

$$t_c < t_d + 2(C_i + C_j)/\alpha \;\Rightarrow\; t_c < t_d + \Delta^{\pi_k} \tag{13}$$

Expression (13) contradicts the earlier definition of $t_c = t_d + \Delta^{\pi_k}$, therefore Theorem 1 holds. □

Note that the tardiness bound given by Eq. (7) differs from the tardiness bounds of EDF-os given by Eqs. (3) and (10) in [2]. This is caused by the differences in the *execution* phase between the two scheduling algorithm described in Sect. 3.2.

3.4 Tardiness bounds under PWM scheme

The optimal normalized speed α_{opt} which can minimize energy consumption while guaranteeing schedulability, is derived from the lower bound in expression (5). This α_{opt}, however, often may not be supported by the system. Example 4 shows such a case. Recall that by

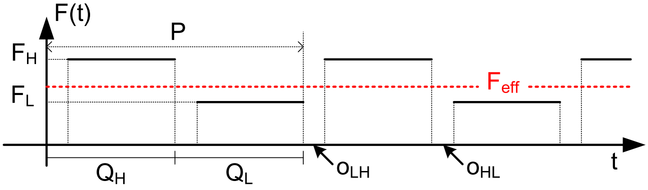


Fig. 6 PWM scheme execution

Definition 1, α_{opt} corresponds to the optimal frequency F_{opt} that can guarantee schedulability. Although running *constantly* at this optimal frequency F_{opt} may not be supported by the system, it is possible to achieve this optimal frequency value *in the long run*, exploiting a "Pulse Width Modulation" (PWM) scheme, where the system switches periodically between two supported frequencies, F_L and F_H, with $F_L < F_{\text{opt}} < F_H$. In particular, we consider the PWM technique presented in [6], which we summarize in the following subsection.

3.4.1 PWM scheme

The PWM scheme presented in [6] is aimed at uniprocessor systems with HRT constraints. The execution of the scheme at run-time is sketched in Fig. 6. The PWM scheme switches periodically between a lower frequency F_L and a higher frequency F_H. The period of the PWM scheme is denoted by P.

The duration of the interval of the low-frequency (high-frequency) mode is Q_L (Q_H). Note that $Q_L + Q_H = P$. Moreover, [6] defines $\lambda_L = \frac{Q_L}{P}$ and $\lambda_H = \frac{Q_H}{P}$, the fraction of time spent running at low and high modes, respectively.

As shown in Fig. 6, the scheme considers time overheads due to frequency switching. These overheads are denoted by o_{LH} for transitions between lower to higher frequencies, and by o_{HL} for the opposite transitions. In addition, [6] denotes the amount of clock cycles lost during frequency transitions as $\Delta_{\text{LH}} = F_L o_{\text{HL}} + F_H o_{\text{LH}}$.

Under the above definitions, the effective frequency obtained by running the processor at F_L for Q_L time and F_H for Q_H time is given by expression (8) in [6]:

$$F_{\text{eff}} = \lambda_L F_L + \lambda_H F_H - \Delta_{\text{LH}}/P \tag{14}$$

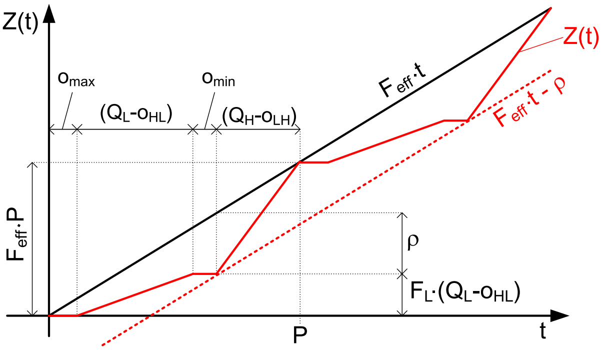
To ensure HRT execution on the system, in their analysis the authors leverage the processor supply function $Z(t)$, defined as the *minimum number of cycles that the processor can provide in every interval of length t*. From the parameters of the PWM scheme, $Z(t)$ is depicted with a solid red line in Fig. 7, with $o_{\max} = \max\{o_{\text{LH}}, o_{\text{HL}}\}$ and $o_{\min} = \min\{o_{\text{LH}}, o_{\text{HL}}\}$. Function $Z(t)$ is zero in $[0, o_{\max}]$; grows linearly with slope F_L in $[o_{\max}, o_{\max} + Q_L - o_{\text{HL}}]$; stays constant in $[o_{\max} + Q_L - o_{\text{HL}}, o_{\max} + Q_L - o_{\text{HL}} + o_{\min}]$; finally, grows with slope F_H until the end of the period P. Note that $Z(t)$ is periodic with period P.

3.4.2 Tardiness bounds derivation

In our approach, we leverage the processor supply function $Z(t)$ to derive tardiness bounds for any task running on a processor under our EDF-ssl scheduling algorithm. These tardiness bounds are given by the following theorem.

Theorem 2 *Consider a processor π_k, on which the PWM scheme described in Sect. 3.4.1 is applied to obtain an effective frequency F_{eff}. Assume that two migrating tasks, τ_i (with*

Fig. 7 Supply function $Z(t)$



WCET C_i) and τ_j (with WCET C_j), are assigned to π_k. Then, jobs of fixed and migrating tasks released on π_k may incur a tardiness of at most

$$\Delta_{PWM}^{\pi_k} = \frac{2(C_i + C_j)}{\alpha_{eff}} + \frac{\rho}{F_{eff}} \tag{15}$$

with $\rho = (F_{eff} - F_L)Q_L + F_L o_{HL} + F_{eff} o_{LH}$ and α_{eff} is derived using Definition 1 from F_{eff}.

Proof To prove Theorem 2, we first derive a lower bound for $Z(t)$. We define the following parameter:

$$\rho = max_{t \in \mathbb{R}^+}\{F_{\text{eff}} \cdot t - Z(t)\}$$

which represents the maximum difference between the “optimal” number of cycles, provided in $[0, t]$ by a processor running at F_{eff}, and $Z(t)$. From Fig. 7 we get:

$$\rho = (F_{\text{eff}} - F_L)Q_L + F_L o_{\text{HL}} + F_{\text{eff}} o_{\text{LH}} \tag{16}$$

from which we can express a lower bound for $Z(t)$ as:

$$\check{Z}(t) = F_{\text{eff}} \cdot t - \rho \tag{17}$$

$\check{Z}(t)$ is depicted in Fig. 7 with a dashed red line. We can then express $Z(t)$ as $Z(t) = \check{Z}(t) + e(t)$, with $e(t) \geq 0, \forall t \geq 0$.

Now, we follow the proof of Theorem 1. This time, the instant t_c is defined as $\mathbf{t_c} = t_d + \Delta_{\text{PWM}}^{\pi_k}$, and we assume that a certain job $\tau_{q,l}$ does not complete by time t_c. The definitions of $\mathbf{t_0}$ and $\mathbf{t_d}$ are unchanged. The demand from fixed and migrating tasks, expressed in clock cycles, is still bounded by (11). However, we have to change the left-hand side of the inequality in (12) with $Z(t_c - t_0)$, obtaining:

$$F_{\text{eff}} \cdot (t_c - t_0) - \rho + e(t_c - t_0) < F_{\max}\left(\alpha_{\text{eff}}(t_d - t_0) + 2(C_i + C_j)\right) \tag{18}$$

Since $F_{\text{eff}} = F_{\max}\alpha_{\text{eff}}$, dividing both sides by $F_{\max}\alpha_{\text{eff}}$ we get:

$$(t_c - t_0) - \frac{\rho - e(t_c - t_0)}{F_{\text{eff}}} < (t_d - t_0) + \frac{2(C_i + C_j)}{\alpha_{\text{eff}}}$$

therefore:

$$(t_c - t_d) < \frac{2(C_i + C_j)}{\alpha_{\text{eff}}} + \frac{\rho - e(t_c - t_0)}{F_{\text{eff}}} = \Delta_{\text{PWM}}^{\pi_k} - \frac{e(t_c - t_0)}{F_{\text{eff}}} \tag{19}$$

Even with $e(t_c - t_0) = 0$, which represents the worst case, expression (19) contradicts the definition of t_c, therefore Theorem 2 holds. □

Note that by Eq. (15) it follows that tardiness can be experienced even on processors with no migrating tasks, given the fact that the term ρ depends only on the parameters of the PWM scheme.

4 Start times and buffer sizes under EDF-ssl

In Sect. 2.3.2 we summarize the analysis that guarantees correctness of start times and buffer sizes under the SRT scheduling algorithm used in [7], namely EDF-fm. In order to maintain this analysis valid for our proposed EDF-ssl, we must take into account the differences between our EDF-ssl and EDF-fm.

Let us consider again the data-dependent actors v_s (source) and v_d (destination) shown in Fig. 1. We recall that, as described in Sect. 2.3, in our analysis v_s and v_d are converted into two periodic tasks τ_s and τ_d. Assume, for instance, that the system runs at a certain constant normalized speed α, and both τ_s and τ_d are assigned to the processors as migrating tasks, with the share assignment shown in Fig. 8a. Shares $s_{s,1}$ and $s_{s,2}$ of τ_s are assigned to π_1 and π_2, whereas shares $s_{d,2}$ and $s_{d,3}$ of τ_d are assigned to π_2 and π_3. In Fig. 8a, the dashed areas in each processor represent processor utilization assigned to tasks other than τ_s and τ_d. These other tasks are assumed to be of fixed type (i.e., not migrating). Since π_1 and π_3 run only one migrating tasks, by Eq. (7) we derive the following tardiness bounds: $\Delta^{\pi_1} = 2C_s/\alpha$, $\Delta^{\pi_2} = 2(C_s + C_d)/\alpha$, $\Delta^{\pi_3} = 2C_d/\alpha$, where C_s and C_d are the WCETs of τ_s and τ_d, respectively. It follows that under our EDF-ssl jobs of the same migrating task have different tardiness bounds, depending on which processor the jobs are released. For instance, jobs of τ_s will incur a tardiness of at most Δ^{π_2} when released on π_2, and Δ^{π_1} when released on π_1, with $\Delta^{\pi_2} > \Delta^{\pi_1}$. By contrast, under EDF-fm used in [7], jobs of a migrating task experience no tardiness at all, because tardiness can only be experienced by fixed tasks. In addition, under our EDF-ssl jobs of the same migrating task can execute in parallel. This cannot happen under EDF-fm.

In the remainder of this section we define a way to guarantee a correct schedule of τ_s and τ_d, with no buffer underflow or overflow, under our EDF-ssl. As shown in Fig. 8b, we assume that processors running communicating tasks have access to a shared memory where data communication buffers are allocated. Note that our approach allows data and instruction memory of all processors to be completely distributed, therefore contention can only occur when accessing the shared communication memory. In Fig. 8b, buffer b_u of size B implements the communication over edge e_u of Fig. 1. Our analysis to guarantee a correct scheduling of τ_s and τ_d comprises two parts. First, we guarantee valid start times of τ_s and τ_d and buffer size B by adapting the analysis in [7] to our EDF-ssl. Second, we define a

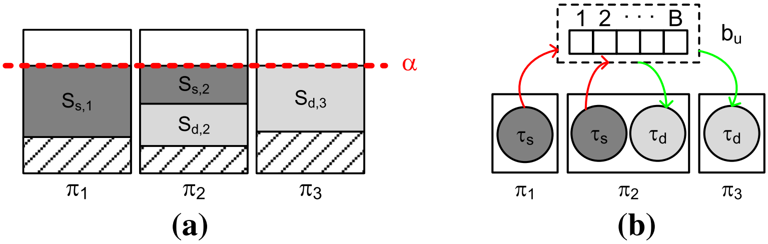


Fig. 8 Analysis of communication between data-dependent actors when both source and destination actors are implemented as migrating tasks. **a** Considered share assignment of τ_s and τ_d. **b** Scheme of the communication between τ_s and τ_d over b_u

pattern that τ_s and τ_d use when reading/writing from/to b_u to ensure functional correctness. These two parts are described below.

Part 1—Valid start times and buffer sizes As mentioned earlier, under our EDF-ssl jobs of the same migrating task can have different tardiness bounds, if released on different processors. According to Definition 2, the tardiness bound Δ_i of a certain task τ_i must be valid for all its jobs. Therefore, we set the value of Δ_i to the maximum tardiness bound among the processors which are assigned (non-zero) shares of τ_i, as follows:

$$\Delta_i = \begin{cases} \max_{k \mid s_{i,k}>0}\{\Delta^{\pi_k}\} & \text{under fixed processor speed} \\ \max_{k \mid s_{i,k}>0}\{\Delta^{\pi_k}_{PWM}\} & \text{under PWM scheme} \end{cases} \tag{20}$$

where Δ^{π_k} and $\Delta^{\pi_k}_{PWM}$ are the tardiness bounds calculated for processor π_k under fixed processor speed and under the PWM scheme described in Sect. 3.4.1, respectively. For each processor π_k, Δ^{π_k} and $\Delta^{\pi_k}_{PWM}$ are obtained using Eq. (7) and Eq. (15), respectively. Finally, in Eq. (20) $s_{i,k}$ represents the share of τ_i on π_k.

By using the tardiness bound Δ_i expressed by Eq. (20), we can represent the ALAP completion schedule (see Definition 3 in Sect. 2.3.2) of actor v_i (corresponding to task τ_i) as a fictitious actor $\tilde{v_i}$, which has the same period as v_i, no tardiness, and start time $\tilde{S_i} = S_i + \Delta_i$. From Eq. (20) it follows that at run time any invocation $v_{i,j}$ of actor v_i will never be completed later than the corresponding invocation $v_{\tilde{i},j}$ of actor $\tilde{v_i}$, regardless of which processor is executing that invocation. Therefore, the analysis for start times and buffer sizes in the presence of tardiness described in Sect. 2.3.2 can be applied considering the tardiness bounds given by Eq. (20) and it is correct for our EDF-ssl.

Part 2—Reading/writing pattern to/from b_u Let us focus on the source actor v_s in Fig. 1, and let assume the share assignment shown in Fig. 8a. Under our EDF-ssl, jobs of τ_s, which correspond to invocations of v_s, may execute in parallel if released onto different processors. Moreover, as mentioned earlier, jobs of τ_s may experience different tardiness, depending on which processor the job is released. It follows that jobs of τ_s may write out-of-order to buffer b_u in Fig. 8b. This is because job $\tau_{s,k+a}$, for some $a > 0$, may finish before job $\tau_{s,k}$ if they are released on different processors. Similarly, jobs of the destination task τ_d may read from b_u out-of-order.

In this scenario, it is clear that b_u is not a First-in First-out (FIFO) buffer. Thus, every job of τ_s/τ_d (invocation of v_s/v_d) must know *where* it has to write/read to/from b_u. **Part 1** of our analysis (described above) ensures that B, the size of b_u, is large enough to guarantee that tokens produced by τ_s will never overwrite locations which contain tokens still not consumed by τ_d. Then, given x^u_s, the amount of tokens produced on e_u by every job of τ_s, we enforce that job j of τ_s (with $j \in \mathbb{N}_0$) writes tokens to b_u in the order indicated by Algorithm 2. In fact, Algorithm 2 defines a writing pattern that follows the one which would be obtained if the jobs of τ_s wrote in-order to a FIFO buffer of size B implemented as a circular buffer. Lines 5-7 in Algorithm 2 handle the case in which the x^u_s tokens are "wrapped" in the buffer. Note that by replacing in Algorithm 2 x^u_s with y^u_d and write operations with read operations we obtain the reading pattern corresponding to job j of destination task τ_d.

As an example of the reading and writing pattern enforced by Algorithm 2, consider the case in which the production rate of source task τ_s and the consumption rate of destination task τ_d over edge e_u are the same, i.e., $x^u_s = y^u_d$. In this case, job j of the destination task τ_d must read tokens produced by job j of the source task τ_s. This is ensured if both τ_s and τ_d comply to the reading/writing pattern defined by Algorithm 2, regardless of the *order* in which (i) consecutive jobs of τ_s write to buffer b_u and (ii) consecutive jobs of τ_d read from buffer b_u. Note that Algorithm 2 captures also the case in which $x^u_s \neq y^u_d$.

Algorithm 2: Write pattern of job j of source task τ_s.

```
Input: Number of produced tokens $x_s^u$, job index $j$, buffer size $B$.
bgn = $[(x_s^u \cdot j) \mod B] + 1$;
end = $(x_s^u \cdot (j+1)) \mod B$;
if $bgn < end$ then
    write $x_s^u$ tokens from $b_u$[bgn] to $b_u$[end]
else
    write $(bgn\text{-}B\text{+}1)$ tokens from $b_u$[bgn] to $b_u$[B];
    write remaining tokens from $b_u$[1] to $b_u$[end];
```

5 Evaluation

In this section we evaluate the effectiveness of our semi-partitioned approach in terms of energy savings. We compare our results with the heuristic-based partitioned approach which guarantees the most balanced distribution of utilization of tasks among the available processors, and therefore the least energy consumption, as shown in [3]. The authors in [3] also show that the most balanced distributions are derived when worst fit decreasing (WFD) heuristic is used to determine the assignment of tasks to processors. Each processor then schedules the tasks assigned to it using a local EDF scheduler. In the rest of this section, we will refer to this partitioned approach with the acronym PAR. Note that under PAR all tasks meet their deadlines. By contrast, our proposed semi-partitioned approach will be denoted in the rest of this section with SP when fixed processor speed is used, and with PWM when the periodic speed switching scheme is adopted. Note that although under our approach tasks may experience tardiness, this has no effect on the guaranteed throughput, which remains constant among all the considered approaches (PAR, SP, PWM). However, task tardiness has an impact on buffer sizes and start times of tasks (and, in turn, on the latency of applications), as described in Sect. 4. Note that although the PAR approach provides HRT guarantees to all tasks in the system, whereas both SP and PWM only provide SRT guarantees, our comparison remains fair. This is because:

- As shown in [7] and mentioned in Sect. 2.3.2, also SP and PWM can guarantee HRT behavior at the input/output interfaces with the environment, although some of the tasks of the application may experience tardiness.
- Both SP and PWM, adopting the technique of [7] described in Sect. 2.3.2, guarantee the same throughput as PAR.

These two conditions are sufficient for the kind of applications that we consider, in which throughput constraints are more relevant than application latency and memory overheads. Note also that in our scheduling framework, since actors are released strictly periodically, the application latency is the elapsed time between the start of the first firing of the input actor and the worst-case completion of the first firing of the output actor.

5.1 Power model

As mentioned in Sect. 2.1, we consider homogeneous multiprocessor systems, in which any core can be either idle or running at a global (normalized) speed α. We assume that the system supports a discrete set of operating voltage/frequency modes. In our experiments we refer to the operating modes of a modern System-on-Chip, the OMAP 4460, as in [28]. This SoC comprises two ARM Cortex-A9 cores that can operate at $\Phi_{\text{A9}} = \{0.350, 0.700, 0.920, 1.200\}$

GHz, at a supply voltage of {0.83, 1.01, 1.11, 1.27} V, respectively. From Φ_{A9} we can derive the set of supported normalized speed:

$$A_{\mathrm{A9}} = \{0.292, 0.583, 0.767, 1.0\}$$

We use the power model of a similar dual Cortex-A9 core system, considered in [20], which we normalize to a single core:

$$p_{\mathrm{cpu}} = p_{\mathrm{dyn}} + p_{\mathrm{sta}} = (0.223 V_{\mathrm{cpu}}^2 F_{\mathrm{cpu}}) + (K_1 V_{\mathrm{cpu}} + K_2) \tag{21}$$

where $K_1 = 0.08965$, $K_2 = 0.07635$, V_{cpu} represents the voltage supplied to the CPU in Volts, and F_{cpu} represent the CPU frequency in GHz. The model comprises dynamic power p_{dyn} and static power p_{sta}, and the value of p_{dyn} assumes that the core is fully utilized. Note that expression (21) assumes that the processor runs at one of the supported normalized speeds $\alpha_i \in A_{\mathrm{A9}}$. From this α_i, we can derive the processor frequency $F_{\mathrm{cpu}} = F_i$ by Definition 1. Similarly, to a normalized speed α_i corresponds an unique voltage level V_{cpu}. Therefore, the power consumption p_{dyn} and p_{sta} depend uniquely on α_i. We make this relation explicit by using the notation $p_{\mathrm{dyn}}(\alpha_i)$, $p_{\mathrm{sta}}(\alpha_i)$, and $p_{\mathrm{cpu}}(\alpha_i)$.

5.2 Energy per iteration period

Based on the power model expressed by Eq. (21), we now proceed by deriving the energy consumption under PAR, SP, and PWM. In particular, we derive the energy consumed by the system during one *iteration period* (H) of the graph (recall Eq. (2)). Note that the iteration period of the graph is *the same* and *constant* among PAR, SP, and PWM, because the periods of all tasks do not change depending on the considered scheduling approach. Note also that, *regardless of the application latency*, every task τ_i executes q_i times during one iteration period H (recall, again, Eq. (2)). We assume that α is sufficient to guarantee schedulability, therefore $\alpha \geq \sigma_k$, for any active processor π_k. In the following, we denote the number of active cores with M_{ON}.

Static energy of (PAR, SP) Both these approaches run at a fixed speed α_i, and the static energy consumed in one iteration period H is given by:

$$E_{\mathrm{sta}}^{H,\mathrm{FIX}} = H \cdot p_{\mathrm{sta}}^{\mathrm{FIX}} = H \cdot M_{\mathrm{ON}} \cdot p_{\mathrm{sta}}(\alpha_i) \tag{22}$$

Dynamic energy of (PAR, SP) We derive the dynamic energy consumption in one iteration period H. During one iteration period, each task τ_j executes $q_j = H/T_j$ times. Each worst-case execution takes C_j/α_i time, at dynamic power $p_{\mathrm{dyn}}(\alpha_i)$. Therefore, the dynamic energy consumed by task τ_j during one iteration period H is:

$$E_{\mathrm{dyn}}^{H,\mathrm{FIX}}(\tau_j) = q_j \frac{C_j}{\alpha_i} p_{\mathrm{dyn}}(\alpha_i) \tag{23}$$

From Eq. (23) we derive the dynamic energy consumed in one iteration period H by the whole task set as follows:

$$E_{\mathrm{dyn}}^{H,\mathrm{FIX}} = \sum_{\tau_j \in \Gamma} q_j \frac{C_j}{\alpha_i} p_{\mathrm{dyn}}(\alpha_i) = \frac{p_{\mathrm{dyn}}(\alpha_i)}{\alpha_i} \sum_{\tau_j \in \Gamma} q_j C_j \tag{24}$$

Total energy of (PAR, SP) From Eqs. (22) and (24) we derive the total energy consumed during one iteration period H under (PAR, SP) by:

$$E_{\mathrm{tot}}^{H,\mathrm{FIX}} = H \cdot M_{\mathrm{ON}} \cdot p_{\mathrm{sta}}(\alpha_i) + \frac{p_{\mathrm{dyn}}(\alpha_i)}{\alpha_i} \sum_{\tau_j \in \Gamma} q_j C_j \tag{25}$$

Total energy of PWM Under PWM, the system switches periodically between normalized speeds α_L and α_H to guarantee a certain α_{eff} in the long run. Therefore, we cannot use Eq. (25) to model the energy consumption per iteration period under the PWM scheme, because that expression is only valid when the system runs constantly at one of the supported normalized speeds α_i. For the sake of clarity, we will denote $p_{\text{cpu}}(\alpha_L)$ and $p_{\text{cpu}}(\alpha_H)$, obtained from Eq. (21), with p_L and p_H, respectively. In this scenario, the total power of a single core of the system is provided by expression (9) in [6], reported below.

$$p_{\text{cpu}}^{\text{PWM}} = \lambda_L p_L + \lambda_H p_H + E_{\text{SW}}/P \tag{26}$$

where $E_{\text{SW}} = e_{\text{LH}} - p_H o_{\text{LH}} + e_{\text{HL}} - p_L o_{\text{HL}}$, which represents the energy wasted during two speed transitions. The terms λ_L, λ_H, o_{LH}, o_{HL}, P, are parameters of the PWM scheme defined in Sect. 3.4.1, whereas e_{LH} and e_{HL} represent the *energy* overhead incurred in the speed transition from α_L to α_H and vice versa. We assume that $e_{\text{LH}} = e_{\text{HL}} = 1\,\mu J$ and $o_{\text{LH}} = o_{\text{HL}} = 10\,\mu s$. These values are compatible with the findings in [20]. Now, given the number of active cores M_{ON}, we can express the total energy per iteration period H under PWM as:

$$E_{\text{tot}}^{H,\text{PWM}} = H \cdot M_{\text{ON}} \cdot p_{\text{cpu}}^{\text{PWM}} \tag{27}$$

Note that Eq. (27) depends on Eq. (26), which in turn depends on the parameters of the PWM scheme. In particular, we have to find an appropriate value for the PWM scheme period P. Since we assume that speed changes can only happen at the granularity of the operating system tick (which has period T_{OS}), we enforce P to be a multiple of T_{OS}. From Eq. (14), we derive the shortest P, multiple of T_{OS}, that makes the overhead-induced clock cycles loss less than $\epsilon = 0.01$ times the desired F_{opt}. Thus, $P \geq \Delta_{\text{LH}}/(\epsilon F_{\text{opt}})$. Given P, we find the shortest Q_H, multiple of T_{OS}, that guarantees an effective frequency F_{eff} greater than or equal to F_{opt} (from Eq. (14); note that $Q_L = P - Q_H$). At this point, all the parameters of the PWM scheme are known and the total energy consumption per iteration period can be derived using Eq. (27).

5.3 Experimental results

We evaluate the considered approaches (PAR, SP, PWM) on a set of real-life applications modeled as SDF graphs, which are listed in Table 1. For each application, the table reports:

- a short description of its functionality (column *Description*);
- the number of SDF actors (column *No. of actors*);
- the maximum WCET among actors (column *Max WCET*), expressed in seconds [s].

Table 1 Characteristics of the considered applications

Application	Description	No. of actors	Max WCET [s]
DCT	Discrete cosine transform	8	7.94×10^{-5}
JP2	JPEG2000 image encoder	6	2.88×10^{-3}
MJPEG	Motion JPEG video encoder	5	1.03×10^{-4}
MPEG2	MPEG2 video decoder	23	7.68×10^{-5}
TDE	Time-delay equalization	29	1.23×10^{-4}

In Table 2 we show the results obtained using the considered approaches (PAR, SP, PWM) on the set of applications listed in Table 1. The name of each application is shown in column *App* of Table 2. Moreover, for each application, column U_Γ reports the cumulative utilization of the corresponding task set. In addition, column R shows the throughput obtained for each application. For a certain application, given x^{out} which is the number of tokens produced by its output actor at every invocation, the application throughput can be computed as $R = x^{\text{out}}/T_{\text{out}}$ where T_{out} is the period of the output actor. Therefore, the application throughput R is given in tokens per second [tkns/s]. Note that the throughput R and the total utilization U_Γ of each application remain *constant* among all considered allocation/scheduling approaches (PAR, SP, PWM). The reason is that the WCET of each task τ_i, derived using Eq. (1), does not depend on the actual assignment of tasks to processors, because it considers the worst-case communication time among all possible assignments of tasks.

Each row in Table 2 corresponds to results obtained considering a system composed of $\hat{M}$ available cores, with $\hat{M} \in \{4, 8, 12\}$. Note that column $\hat{M}$ shows only meaningful values, those which satisfy $\hat{M} \geq \lceil U_\Gamma \rceil$. For each of the considered approaches (PAR, SP, PWM), and for each value of $\hat{M}$, we consider each possible number of active processors M_{ON} in the range $[\lceil U_\Gamma \rceil, \hat{M}]$ and look for the lowest energy consumption, thereby exploring the design space exhaustively. For every value of M_{ON} in that range, we follow a different procedure depending on the considered approach.

In PAR, we simply assign the utilization of tasks to the M_{ON} active cores using the WFD heuristic. Then, if WFD is successful, we derive the lowest $\alpha_i \in A_{\text{A9}}$ which guarantees schedulability ($\min_{\alpha_i \in A_{\text{A9}}}\{\alpha_i \geq \sigma_k, \forall \, \text{active} \, \pi_k\}$). Knowing M_{ON} and α_i, we determine the total energy per iteration period E^H_{PAR} by Eq. (25).

In SP, we find the necessary minimum speed $\alpha_{\text{opt}} = U_\Gamma / M_{\text{ON}}$. We round this speed value to the closest greater or equal value in A_{A9}, which we denote with α_i. We run Algorithm 1 with this speed value α_i and $M = M_{\text{ON}}$. If Algorithm 1 is successful, we determine the total energy per iteration period E^H_{SP} by Eq. (25) with the considered α_i and M_{ON}.

In PWM, we calculate $\alpha_{\text{opt}} = U_\Gamma / M_{\text{ON}}$ and we run Algorithm 1 with speed value α_{opt} and $M = M_{\text{ON}}$. If Algorithm 1 is successful, we use $\alpha_H = \min_{\alpha_i \in A_{\text{A9}}}\{\alpha_i \geq \alpha_{\text{opt}}\}$ and $\alpha_L = \max_{\alpha_i \in A_{\text{A9}}}\{\alpha_i \leq \alpha_{\text{opt}}\}$ and derive the total energy per iteration period E^H_{PWM} by Eq. (27).

For each valid task share assignment, we derive earliest start times of actors and buffer size requirements by using the formulas mentioned in Sect. 2.3.2 with, for each task τ_l, either of the following tardiness bound values: $\Delta_l = 0$ in PAR; Δ_l obtained by Eq. (20) in SP and PWM.

At the end of the design space exploration, for PAR and SP, we report in Table 2 the values of $M_{\text{ON}} \in [\lceil U_\Gamma \rceil, \hat{M}]$ that yielded to the lowest energy consumption. For PAR (SP), these values are shown in column M^o_{PAR} (M^o_{SP}). Note that the optimal values of M_{ON} for PWM are identical to M^o_{SP}, therefore they are not included.

In the following discussion, we will identify rows in Table 2 with the couple (App, $\hat{M}$). For each of these rows, under PAR, the table shows: the optimal number of active processors M^o_{PAR}; the total memory requirement TM_{PAR} (including code, stack, buffers); the application latency L_{PAR}; the energy consumption E^H_{PAR}.

We see from Table 2 that SP consumes significantly lower energy than PAR, see column $E^H_{\text{SP}}/E^H_{\text{PAR}}$. On average, we obtain an energy saving of 36 %. The energy saving goes up to 64 %, see row (JP2, 8). These energy savings, however, come at a cost. The total memory requirements (see column $\text{TM}_{\text{SP}}/\text{TM}_{\text{PAR}}$) and application latencies (see column $L_{\text{SP}}/L_{\text{PAR}}$) are increased. Memory requirements increase due to *i)* more task replicas (with their code

Table 2 Comparison of different share allocation/scheduling approaches

App	U_Γ	R	$\hat{M}$	PAR				SP				PWM			
		[tkns/s]		M^o_{PAR}	TM_{PAR}[kB]	L_{PAR}[ms]	E^H_{PAR}[J]	M^o_{SP}	$\frac{TM_{SP}}{TM_{PAR}}$	$\frac{L_{SP}}{L_{PAR}}$	$\frac{E^H_{SP}}{E^H_{PAR}}$	α_{opt}	$\frac{TM_{PWM}}{TM_{PAR}}$	$\frac{L_{PWM}}{L_{PAR}}$	$\frac{E^H_{PWM}}{E^H_{PAR}}$
DCT	2.43	12,000	4	3	22.7	0.667	9.67×10^{-5}	4	1.37	1.62	0.77	0.61	2.18	3.85	0.66
			8	3	22.7	0.667	9.67×10^{-5}	5	1.64	2.22	0.64	0.49	3.02	5.95	0.61
			12	3	22.7	0.667	9.67×10^{-5}	9	3.14	5.25	0.37	0.27	na	na	na
JP2	1.23	333.33	4	2	114	17.5	1.78×10^{-3}	3	1.42	1.28	0.62	0.41	1.9	3.45	0.51
			8	2	114	17.5	1.78×10^{-3}	5	2.28	2.13	0.36	0.25	na	na	na
			12	2	114	17.5	1.78×10^{-3}	5	2.28	2.13	0.36	0.25	na	na	na
MJPEG	1.22	6000	4	2	62.7	0.833	7.50×10^{-5}	3	1.31	1.42	0.84	0.41	1.94	4.73	0.69
			8	2	62.7	0.833	7.50×10^{-5}	5	1.8	3.17	0.47	0.24	na	na	na
			12	2	62.7	0.833	7.50×10^{-5}	5	1.8	3.17	0.47	0.24	na	na	na
MPEG2	6.81	12,000	8	8	345	1.50	2.70×10^{-4}	7	1.44	1.55	0.99	0.97	1.84	2.00	1.00
			12	8	345	1.50	2.70×10^{-4}	12	2.01	2.21	0.63	0.57	2.45	3.01	0.65
TDE	6.28	3000	8	7	840	9.67	9.90×10^{-4}	7	1.00	1.00	1.00	0.9	1.42	1.42	0.94
			12	9	840	9.67	7.60×10^{-4}	11	1.68	1.76	0.83	0.57	1.68	1.78	0.84

and stack memory) needed by the semi-partitioned approach and *ii)* more buffers due to task tardiness. Similarly, application latency increases because task tardiness postpones the start times of the tasks of the application.

The rightmost part of Table 2 presents the results under PWM. It shows that this approach can provide higher energy savings compared to SP (compare columns $E^H_{\text{PWM}}/E^H_{\text{PAR}}$ and $E^H_{\text{SP}}/E^H_{\text{PAR}}$). The additional energy saving can grow up to 18 % (see rows (JP2,4) and (MJPEG,4)) compared to SP. Rows with *na* (not applicable) values indicate that the value of α_{opt} (see the corresponding column) is lower than the minimum speed in A_{A9}. Therefore, the PWM scheme is not applicable. Note that in three rows the value of $E^H_{\text{PWM}}/E^H_{\text{PAR}}$ is higher than $E^H_{\text{SP}}/E^H_{\text{PAR}}$. This means that, in those cases, PWM is less effective than SP. The largest inefficiency is obtained in row (MPEG2,12). In all these cases, the value of α_{opt} is extremely close to one of the speeds in A_{A9}, therefore the energy overhead incurred by the PWM scheme renders PWM disadvantageous. Finally, note that PWM incurs more total memory and latency overheads compared with SP, see columns $\text{TM}_{\text{PWM}}/\text{TM}_{\text{PAR}}$ and $L_{\text{PWM}}/L_{\text{PAR}}$. This is due to the higher number of task replicas, and higher values of tardiness, incurred under PWM.

Note that our experimental results, summarized in Table 2, evaluate our proposed approaches SP and PWM using PAR as a reference point. However, we also made a second comparison, by evaluating our SP and PWM against the results that can be obtained by using the EDF-os scheduling algorithm as a reference. We do not show the results of the comparison against EDF-os in a separate table because that table would be nearly identical to Table 2. This is because PAR and EDF-os achieve nearly the same results, for the following reason. Since our designs are aimed at achieving the maximum throughput of the considered applications, the utilization of at least one of the tasks of each application is close to one. In this scenario, as shown in Sect. 3, EDF-os cannot distribute the utilization of such "heavy" tasks on multiple processors, therefore the operating frequency of the system cannot be lowered without compromising the schedulability of the system. Because of this, EDF-os does not outperform the PAR approach in our experiments, with the exceptions of the (MPEG2,8) and (MPEG2,12) cases. In both these two cases, EDF-os requires 7 processors to schedule the tasks set (one processor less than PAR) which results in a slightly reduced total energy of $2.67 \times 10^{-4} J$ (compared to $E^H_{\text{PAR}} = 2.70 \times 10^{-4} J$). Since the difference between PAR and EDF-os involves only the MPEG2 benchmark, and is in fact minimal, we choose not to show explicitly in a separate table the comparison of our proposed SP and PWM against EDF-os to avoid redundancy.

Finally, we made a third comparison by evaluating the approaches proposed in this paper (SP and PWM) with the semi-partitioned approach used in [7], in terms of energy savings. Similar to the comparison against EDF-os, described in the above paragraph, we do not report the results of this comparison in a separate table because that table would be nearly identical to Table 2. This is because the approach of [7] and of PAR, which is used as reference point in Table 2, achieve nearly identical results. In fact, the approach of [7] yields the same results as EDF-os, which are considered and explained in the above paragraph.

6 Conclusions

In this paper, we propose a novel soft real-time scheduling algorithm that exploits the presence of stateless tasks in streaming applications, to achieve higher energy efficiency. Our approach provides significant energy savings compared to a pure partitioned scheduling approach and an existing semi-partitioned one, EDF-os, guaranteeing the same throughput. However, this comes at the cost of higher memory requirements and application latency.

References

1. Anderson JH et al (2005) An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In: ECRTS
2. Anderson JH et al (2014) Optimal semi-partitioned scheduling in soft real-time systems. In: RTCSA. doi:10.1109/RTCSA.2014.6910532
3. Aydin H, Yang Q (2003) Energy-aware partitioning for multiprocessor real-time systems. In: IPDPS . doi:10.1109/IPDPS.2003.1213225
4. Bamakhrama M, Stefanov T (2011) Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In: EMSOFT
5. Baruah S et al (1996) Proportionate progress: a notion of fairness in resource allocation. Algorithmica, Hyderabad
6. Bini E et al (2009) Minimizing CPU energy in real-time systems with discrete speed management. Trans Embed Comput Syst 8(4):31. doi:10.1145/1550987.1550994
7. Cannella E et al (2014) System-level scheduling of real-time streaming applications using a semi-partitioned approach. In: DATE
8. Colin A, Kandhalu A, Rajkumar R (2014) Energy-efficient allocation of real-time applications onto heterogeneous processors. In: Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on, pp 1–10
9. Davis R, Burns A (2011) A survey of hard real-time scheduling for multiprocessor systems. ACM Comput Surv 43(4):1–35
10. Devadas V, Aydin H (2010) Coordinated power management of periodic real-time tasks on chip multiprocessors. In: IGCC. doi:10.1109/GREENCOMP.2010.5598261
11. Devi UC (2006) Soft real-time scheduling on multiprocessors. Ph.D. thesis
12. Erickson JP, Anderson JH (2011) Response time bounds for G-EDF without intra-task precedence constraints. In: OPODIS
13. Howard J et al (2010) A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In: ISSCC. doi:10.1109/ISSCC.2010.5434077
14. Huang H et al (2010) Leakage-aware reallocation for periodic real-time tasks on multicore processors. In: FCST doi:10.1109/FCST.2010.105
15. Huang P et al (2013) Throughput-constrained voltage and frequency scaling for real-time heterogeneous multiprocessors. In: SAC doi:10.1145/2480362.2480645
16. Johnson DS (1973) Near-optimal bin packing algorithms. Ph.D. thesis, MIT
17. Kong F, Yi W, Deng Q (2011) Energy-efficient scheduling of real-time tasks on cluster-based multicores. In: Design, Automation test in europe conference exhibition (DATE), pp 1–6
18. Lee E, Messerschmitt D (1987) Synchronous data flow. Proc IEEE 75(9):1235–1245
19. Lee WY (2009) Energy-saving dvfs scheduling of multiple periodic real-time tasks on multi-core processors. In: DS-RT doi:10.1109/DS-RT.2009.12
20. Park S et al (2013) Accurate modeling of the delay and energy overhead of dynamic voltage and frequency scaling in modern microprocessors. Comput-Aid Design Integr Circ Syst IEEE Trans 32(5):695–708. doi:10.1109/TCAD.2012.2235126
21. Seo E et al (2008) Energy efficient scheduling of real-time tasks on multicore processors. Parallel Distrib Syst IEEE Trans 19(11):1540–1552. doi:10.1109/TPDS.2008.104
22. Singh AK et al (2013) Energy optimization by exploiting execution slacks in streaming applications on multiprocessor systems. In: DAC
23. Thies W, Amarasinghe S (2010) An empirical characterization of stream programs and its implications for language and compiler design. In: Proceedings of the 19th international conference on parallel architectures and compilation techniques, pp 365–376 doi:10.1145/1854273.1854319

24. Wang Y et al (2011) Overhead-aware energy optimization for real-time streaming applications on multiprocessor system-on-chip. ACM Trans Design Autom Electron Syst 16(2):14
25. Wei YH et al (2010) Energy-efficient real-time scheduling of multimedia tasks on multi-core processors. In: SAC doi:10.1145/1774088.1774142
26. Yang K, Anderson JH (2014) Optimal gedf-based schedulers that allow intra-task parallelism on heterogeneous multiprocessors. In: ESTIMedia
27. Zheng L (2007) A task migration constrained energy-efficient scheduling algorithm for multiprocessor real-time systems. In: WiCom doi:10.1109/WICOM.2007.759
28. Zhu Y, Reddi VJ (2013) High-performance and energy-efficient mobile web browsing on big/little systems. In: HPCA doi:10.1109/HPCA.2013.6522303