Semi-partitioned scheduling and task migration in dataflow networks

Cannella, E.

Chapter 6

Energy Efficient Semi-Partitioned Scheduling of SDF Graphs

Most of the work presented in this chapter has been published in [CS16].

As mentioned in Section 1.2.4, energy efficiency is one of the emerging challenges of the modern embedded MPSoCs design, 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 energy efficiency challenge many techniques have been proposed in the past decade in the embedded system community. As explained in Section 1.2.4, these techniques exploit Voltage/Frequency Scaling (VFS) of processors and have been applied to both streaming applications and periodic independent real-time tasks sets. These VFS techniques can be classified as *offline* and *online*. Offline VFS uses parameters such as the worst-case execution time (WCET) and the 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.

In this chapter, we devise an approach to exploit VFS of processors, to minimize the energy consumption of streaming applications with throughput constraints. We do so by reusing the scheduling analysis proposed in Chapter 5, which considers the soft real-time (SRT) EDF-fm scheduling algorithm. However, in this chapter we propose a novel SRT semi-partitioned scheduling algorithm, different from EDF-fm, 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. In particular, our proposed SRT semi-partitioned algorithm is based on *restricted migrations*, for the practical reasons explained in the introduction of Chapter 5.

Although the scheduling algorithm used in our VFS scheduling approach is SRT, our proposed approach can provide HRT guarantees to the input/output interfaces of the application with the environment. This property is ensured by the scheduling analysis proposed in Chapter 5, which is reused in this chapter. Therefore, the results of this chapter can be applied in the context of **hard real-time streaming systems**.

6.1 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 chapter, we study the problem of energy minimization when mapping streaming applications with throughput constraints using such semi-partitioned approach. Our technique considers homogeneous multiprocessor systems in which voltage and frequency scaling is supported with a discrete set of operating voltage/frequency modes.

6.2 Contributions

As the main contribution of this chapter, we propose a VFS semi-partitioned scheduling technique aimed at streaming applications with throughput constraints. Our proposed schedling technique is depicted in Figure 6.1. As mentioned earlier, our technique builds upon the results of Chapter 5. In that chapter, we showed that a SRT semi-partitioned scheduler (EDF-fm) can be used to schedule actors of a (C)SDF graph as real-time periodic tasks.

The dependencies of the technique proposed in this chapter with the scheduling analysis of Chapter 5 are highlighted by dashed boxes in Figure 6.1. These boxes include steps that are identical to the scheduling framework in Figure 5.2. In particular, in both figures:

- In Step ①, we use the scheduling analysis of [BS11, BS12] to derive the WCET (I) and period (II) of each task, based on the characteristics of the input application model. Throughout this chapter, we will refer to such derivation as *scheduling analysis of [BS11, BS12]*. Recall, from Chapter 5, that parameters I and II do not depend on the (potential) tardiness of tasks.
- In Step ③, we assume that the periodic tasks will be scheduled by a SRT scheduling algorithm, which provides a certain tardiness bound Δ_i of each task τ_i. Then, based on these tardiness bound values, Step ③ derives valid start times of tasks (III), and sizes of the buffers (IV) which implement inter-task communication.

The differences of our VFS semi-partitioned technique shown in Figure 6.1, with regard to semi-partitioned approach of Chapter 5, are the following.

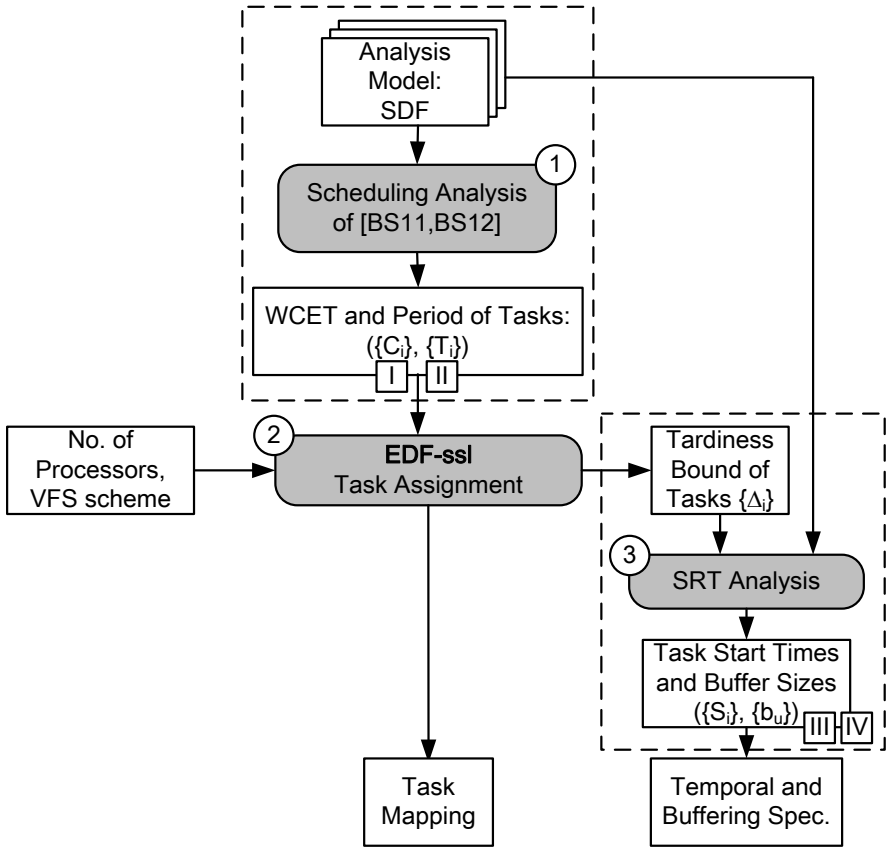


Figure 6.1: *Energy-efficient scheduling technique proposed in this chapter. Our proposed scheduling technique starts with Step ①, where the scheduling analysis of [BS11, BS12] is used to derive the WCET (I) and period (II) of each task, based on the characteristics of the input application model. Step ② uses as input the derived WCET and period of tasks, together with the considered number of active processors and VFS scheme (parameters provided by the designer). This step derives the Task Mapping and the tardiness bound of each task, based on the EDF-ssl scheduling algorithm proposed in this chapter. In turn, given the tardiness bound of tasks, Step ③ derives valid start times of tasks (III) and sizes of the buffers IV that implement inter-task communication. Therefore, after Step ③, the complete Temporal and Buffering Specification is known. The steps enclosed in the dashed boxes reuse part of the scheduling framework shown in Figure 5.2 and represent the dependency of this chapter from the theoretical results of Chapter 5.*

1. The scheduling algorithm used to schedule the tasks on the system is different. Chapter 5 uses the EDF-fm SRT semi-partitioned scheduling algorithm. By contrast, in this chapter, we propose a novel semi-partitioned scheduling algorithm, called EDF-ssl (Earliest Deadline First based semi-partitioned stateless), which is targeted at streaming applications where some of the tasks may be

stateless[1]. In the presence of stateless tasks, our proposed EDF-ssl scheduler can be effectively used to achieve higher energy efficiency compared to existing partitioned and semi-partitioned approaches.

2. The scheduling framework shown in Figure 6.1 does not support VFS (i.e., all processors run at the highest frequency), and provides as *output* the minimum number of processors required to schedule the application. By contrast, in Figure 6.1, the number of available processors, and the VFS mode used in the system, are provided by the designer (see inputs of Step ②). This is because, to achieve higher energy efficiency, it may be beneficial to distribute the tasks of the application on a number of processors greater than the minimum required.
3. The scheduling analysis described in Chapter 5 can accept, as input, applications modeled as CSDF graphs (see Analysis Model in Figure 5.2). By contrast, we restrict the VFS scheduling approach presented in this chapter only to SDF graphs, which are a subset of CSDF graphs. This is because our semi-partitioned technique can only be beneficial if there are actors for which successive invocations can be executed in parallel. Actors that possess such property are, by definition, not allowed in the CSDF model of computation. However, they are allowed in the SDF model.

Note that, in order to perform the SRT Analysis of Step ③ in Figure 6.1, the tardiness bound of each task is required. Therefore, in this chapter we derive the tardiness bounds guaranteed by our proposed EDF-ssl scheduler. In particular, we derive these bounds 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 and 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.

6.3 Scope of work

The assumptions and limitations which define the scope of our work are listed in what follows.

6.3.1 Assumptions

Our work is built on some assumptions that we describe and motivate below.

[1]See Definition 2.3.6 on page 47.

First, we consider systems with distributed program and data memory. As mentioned in Section 1.1.2, this choice of memory subsystem is needed to ensure predictability of the execution at run-time (since PEs do not have to access shared resources to perform the computation), and scalability.

Second, we consider semi-partitioned scheduling, which is a hybrid between two extremes, *partitioned* and *global* scheduling. As shown in Chapter 5, semi-partitioned scheduling can ameliorate the bin-packing issues of partitioned scheduling when applied to streaming applications. At the same time, semi-partitioned scheduling does not incur the excessive memory and migration overheads of global scheduling.

Third, we assume that the system's communication infrastructure is predictable, i.e., it provides guaranteed communication latency. This assumption is needed because Step ① in Figure 6.1 uses the scheduling analysis of [BS11,BS12] (see Section 2.3) to derive WCET and period of each task, based on the characteristics of the input analysis model. This derivation requires the worst-case communication latency to compute the WCET of a task. The WCET of a task 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 Section 2.3 (see, in particular, Equation (2.26) on page 43).

6.3.2 Limitations

The research problem addressed in this chapter, described in Section 6.1, 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 VFS scheduling technique. In what follows, we list the limitations considered in our proposed approach.

First, 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 [TA10] has shown that around 90% of streaming applications can be modeled as acyclic SDF graphs.

Second, we assume that the hardware platform supports a discrete set of operating VFS modes. Moreover, we assume that the operating voltage/frequency mode can only be changed globally over the considered set of processors. Our technique, therefore, finds applicability in hardware platforms that apply the same voltage/frequency mode to all the processors of the system (e.g., the OMAP 4460, as in [ZR13]). 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 [DA10].

Third, 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. Since we consider applications in which 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 μs. Since the VFS transition delay overhead of modern embedded systems is in the range of tens of μs [P$^+$13], 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.

6.4 Related work

Several techniques addressing energy minimization for streaming applications have already been proposed in the literature. Among these, the closest to our work are [WLL$^+$11, SDK13, HMGM13]. [WLL$^+$11] considers applications modeled as Directed Acyclic Graphs, applies certain transformation on the initial graph and then generates task schedules using a genetic algorithm, assuming *per-core* VFS. [SDK13] assumes that applications are modeled as SDF graphs, and is composed of an offline and online VFS phases, to achieve energy optimization. As shown in Section 6.7, our approach exploits results from the real-time scheduling theory that allow, in the presence of stateless tasks, to set the global system frequency to the lowest value which guarantees schedulability and is supported by the system. Both [WLL$^+$11] and [SDK13] 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, [HMGM13] considers both per-core and global VFS but assumes applications modeled as Homogeneous SDF graphs, and that the 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 [DA10] and [SJPL08] are closely related to our approach because they consider global VFS. The authors in [DA10] 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 chapter, 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 [SJPL08] 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 [HXW+10] and [Zhe07]. In [HXW+10], the authors reallocate 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 [Zhe07] 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 [Zhe07] 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 [Zhe07] 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 [Zhe07]. 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 Section 6.7. Moreover, the applicability of the analysis proposed in [Zhe07] to task sets with data dependencies, as in our case, is questionable. In fact, the semi-partitioned scheduling algorithm underlying [Zhe07] is identical to the one proposed by Anderson et al. in [ABD08]. 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 [Zhe07].

As mentioned earlier, the approach we propose in our work exploits the concurrent execution on different processors of jobs of the same task. In a similar fashion, related works that exploit parallel execution of a task on different processors to achieve energy efficiency are [W+10] and [Lee09]. In [W+10] 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 [W+10], 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 [W+10]. An approach similar to [W+10] has been proposed by the authors in [Lee09]. The technique presented in [Lee09] also divides computation-intensive tasks to sub-tasks which can be concurrently executed on multiple cores.

As in [W+10], 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 [Lee09] 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 [Lee09] employs a processor frequency switching scheme to obtain this lowest frequency and still meet all deadlines. However, our analysis is different from [Lee09] in several aspects. First, when assigning sub-tasks load to the available processors, [Lee09] considers only *symmetric* distribution of the load of a task to different processors. In contrast, in our proposed approach, as shown in Example 6.7.2 in Section 6.7, 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 [Lee09] 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 [Lee09] 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 [Lee09].

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 [ABD08] and EDF-os [AEDC14], which are described in Sections 2.2.7 and 2.2.8 of this thesis. Our EDF-ssl algorithm inherits some properties from EDF-fm and EDF-os. 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 Section 6.7, 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 (Section 6.9.3). Similar to our work, analyses of scheduling algorithms that allow jobs within a single task to run concurrently are presented in [EA11, YA14]. However, both these works consider global scheduling algorithms which, as mentioned earlier, entail high overheads especially in distributed memory architectures. In addition, in both [EA11] and [YA14] the potential of exploiting job parallelism to achieve higher energy efficiency is not explored.

6.5 System Model

In this section, we define the system model used in this chapter. As in Chapter 5, we consider a system composed of a set $\Pi = \{\pi_1, \pi_2, \cdots, \pi_M\}$ of M homogeneous processors. In this chapter, however, we assume that processors are endowed with VFS capability. In particular, as explained in the beginning of this chapter, we consider the problem of mapping applications to systems 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 energy. We assume that the system supports only a discrete set $\Phi = \{F_1, F_2, \cdots, F_N\}$ 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 6.5.1. *(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 chapter. Given the set of supported frequencies Φ, by applying Def. 6.5.1 we obtain a set of supported normalized system speeds $\mathcal{NS} = \{\alpha_1, \alpha_2, \cdots, \alpha_N\}$, where $\alpha_N = \alpha_{\max} = 1$.

6.6 Example of SRT Scheduling of an SDF Graph

In this section, we provide an example of the scheduling technique shown in Figure 6.1. This example will be used in the remainder of this chapter. We recall that the dashed boxes in Figure 6.1 represent the dependencies of the scheduling analysis proposed in this chapter from the theoretical results of Chapter 5. Based on the characteristics of the input application model, the steps contained in the dashed boxes are used to obtain the complete *temporal and buffering specification* of the task set, i.e., the WCET, period, and start time of actors, together with the size of the buffers that implement inter-task communication. Altogether, these steps convert the SDF model of the input application to a set of real-time periodic tasks which can be scheduled by an SRT scheduler.

In particular, Step ① in Figure 6.1 uses the scheduling analysis of [BS11, BS12] (described in Section 2.3) to derive the WCET and period of tasks. More in detail, it uses Equation (2.26) on page 43 to derive the WCET of tasks and Equation (2.29) on page 44 to calculate their periods.

Step ② in Figure 6.1 requires as input the obtained WCET and period of actors. In addition, it assumes that the number of available processors in the system and the VFS scheme are given (for instance, by the designer). Using these inputs, Step ② derives the mapping of tasks to processors and the tardiness bound of each task. In

the next sections, we will describe how the task mapping and tardiness bound of tasks are derived under our proposed EDF-ssl scheduler.

Assuming that the tardiness bound of each task is known, Step ③ applies the theoretical results from Chapter 5 to derive the earliest start times of tasks and minimum sizes of the buffers that implement inter-task data communication.

In the example provided below, we describe in greater detail how periods and start times of tasks are derived, in Step ① and Step ③ in Figure 6.1, respectively.

Example 6.6.1. Consider the SDF graph shown in Figure 6.2(a), which has three actors (A_1, A_2, A_3) with WCET indicated between parentheses (C_1=2, C_2=3, C_3=2) and production/consumption rates indicated above the corresponding edges. In Step ① in Figure 6.1, using Equation (2.27) on page 43, we derive the following minimum periods: T_1=T_3=6 and T_2=3, as shown in Figure 6.2(b). Then, suppose that the underlying SRT scheduling algorithm guarantees tardiness bounds Δ_1=1, Δ_2=2 (as indicated in Figure 6.2(a) and visualized in Figure 6.2(b)), whereas Δ_3=0.

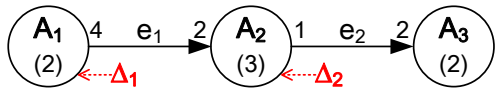
In Step ③ in Figure 6.1, using these tardiness bounds, we apply Proposition 5.5.1 on page 95 and derive the earliest start times S_i shown in Figure 6.2(b). For instance, note that S_2=7 ensures that any invocation of A_2 will always have enough data to read as soon as it is released. This holds even when all the invocations of A_1 incur the largest tardiness Δ_1, i.e., they execute according to the ALAP completion schedule (see Definition 5.5.1 on page 95).

6.7 Proposed Semi-partitioned Algorithm: EDF-ssl

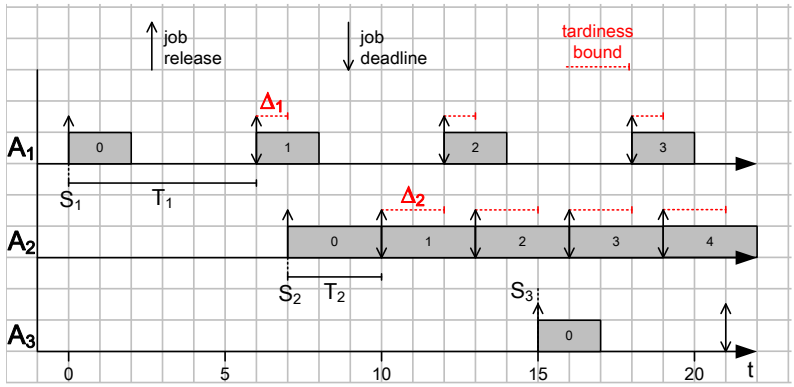
In this section we describe our proposed semi-partitioned scheduler, called EDF-ssl. In EDF-ssl, only stateless tasks (recall Definition 2.3.6 on page 47) are allowed to be migrated. We enforce this condition because migrating the internal state of a stateful task can be prohibitive in a distributed 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. For instance, in Figure 6.4(b), jobs $\tau_{1,0}$ and $\tau_{1,1}$ are executed on two different processors and can execute in parallel.

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., [AY03]) 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 6.7.1. 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



(a) Simple example of an SDF graph. The WCET of actors are indicated between parentheses. The production/consumption rates of actors are indicated above the corresponding edges.



(b) Derived periodic task set and minimum start times. Up arrows represent job releases, down arrows represent task deadlines. The schedule of all actors continues indefinitely.

Figure 6.2: *Example of the approach proposed in Chapter 5 to schedule an SDF graph with a scheduler that provides SRT guarantees. The SDF actors in sub-figure (a) are scheduled as real-time periodic tasks in sub-figure (b). Task periods (T_1, T_2, T_3) are derived using the methodology of [BS11, BS12]. Then, the analysis proposed in Chapter 5 considers the tardiness bounds guaranteed by the SRT scheduler to each task. In this figure, the tardiness bounds of actors τ_1 and τ_2 are Δ_1 and Δ_2, respectively. Using these bounds, the analysis proposed in Chapter 5 derives valid start times (S_1, S_2, S_3) of actors such that all tasks can be released periodically without any buffer underflow.*

effective, because 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 the schedulability test given by Inequality (2.23) on page 42 has to be changed according to the current normalized processor speed.

We enforce therefore $\sigma_1 \leq \alpha$ and $\sigma_2 \leq \alpha$. The resulting assignment of shares of τ_1 is shown in Figure 6.3.

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 Figure 6.4(a), 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,

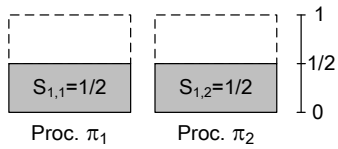
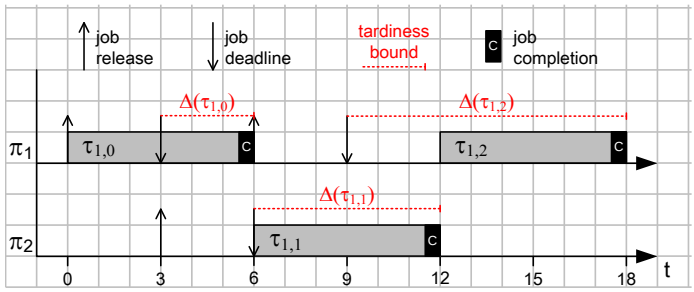
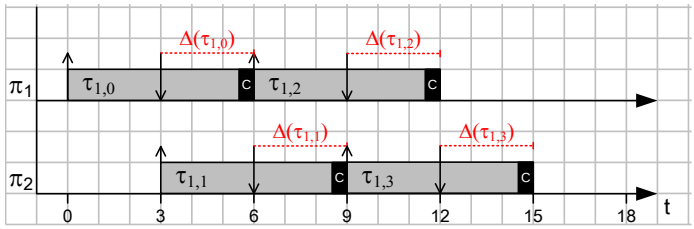


Figure 6.3: *Share assignment considered in Example 6.7.1. The utilization $u_1 = 1$ of a single migrating task τ_1 is split into two shares $s_{1,1} = 1/2$ and $s_{1,2} = 1/2$. Shares $s_{1,1}$ and $s_{1,2}$ are assigned to processors π_1 and π_2, respectively.*



(a) Job executions according to EDF-os rules. Since EDF-os does not consider job parallelism, the tardiness of successive jobs of task τ_1 grows unboundedly, as shown in red in this figure.



(b) Job executions according to EDF-ssl rules. EDF-ssl does consider job parallelism, allowing jobs released by migrating task τ_1 to execute in parallel. This leads to bounded tardiness for all jobs of τ_1.

Figure 6.4: *Job executions of $\tau_1 = (C_1 = 3, T_1 = 3)$, as defined in Example 6.7.1, according to the share assignment of Figure 6.3. Up arrows indicate job releases, down arrows indicate job deadlines. Black rectangles indicate job completion. Although the WCET of τ_1 is 3 time units, each job of τ_1 takes 6 time units to complete because processors π_1 and π_2 run at normalized processor speed $\alpha = 1/2$.*

has to wait until time 6 to start executing. As shown in Figure 6.4(a), 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 Figure 6.4(b), this leads to bounded tardiness for all jobs of τ_1.

Under our EDF-ssl, necessary (but not sufficient) conditions to guarantee schedu-

lability 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: $\alpha \geq u_{s,max}$, where $u_{s,max}$ is the utilization of the heaviest stateful task in Γ. This is because stateful tasks are fixed, and any processor to which the utilization $u_{s,max} > \alpha$ is assigned will be overloaded. We merge the above two conditions in the following expression, which provides necessary higher and lower bounds for α:

$$\max\{U_\Gamma / M,\, u_{s,max}\} \leq \alpha \leq 1 \tag{6.1}$$

We now proceed with a detailed description of our EDF-ssl. As in all semi-partitioned approaches (e.g., [ABD08, AEDC14]), EDF-ssl is composed of two phases, an assignment phase and an execution phase, which are described in Section 6.7.1 and Section 6.7.2, respectively. Tardiness bounds guaranteed under EDF-ssl are derived in Section 6.7.3, for the case of processors running at a fixed normalized speed α. Finally, Section 6.7.4 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.

6.7.1 Assignment Phase

The assignment phase of EDF-ssl tries to find an assignment of tasks to processors that reduces the number of tasks with tardiness. This is because, as described in Chapter 5, many tasks with tardiness result in high overheads in terms of application latency and buffer sizes.

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 the schedulability test given by Condition (2.23) on page 42 as follows:

$$\sigma_k \leq \alpha,\ \forall\, \pi_k \in \Pi \tag{6.2}$$

where σ_k is the total share assignment on any processor π_k. Expression (6.2) implies that σ_k cannot exceed the normalized processor speed. Moreover, note that searching a valid assignment makes only sense if Condition (6.1) is satisfied.

The assignment phase of EDF-ssl consists mainly of 3 steps, which we explain below.

First step. In this step, we find the set of stateful tasks Γ_s within the original task set Γ. Then, we use the First-Fit Decreasing Heuristic (FFD) [Joh73] (see Section 2.2.6) 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: $s_{i,k} = u_i$. On all the other processors, task τ_i has no shares.

Second step. 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} (not assigned), which are assigned in the next step.

Algorithm 3: Share assignment heuristic.

Input: A set of M processors $\Pi = \{\pi_1, \pi_2, \cdots, \pi_M\}$, their normalized speed α, a set of N periodic tasks $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_N\}$.

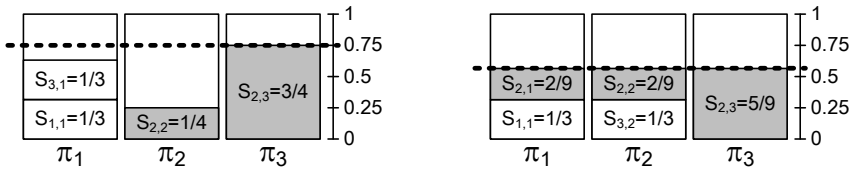
Result: An M-partition describing the share assignment onto M processors if Γ is schedulable, *False* otherwise.

1 Find $\Gamma_s = \{\tau : \tau \in \Gamma \wedge \tau$ is stateful$\}$;
2 **for** $\tau_i \in (\Gamma_s$, *sorted by decreasing utilization*$)$ **do**
3 Try to assign $s_{i,k} = u_i$ of task τ_i on a single π_k using FF;
4 **if** *FF fails for all* $\pi_k \in \Pi$ **then**
5 **return** False;
6 $\Gamma_{\text{na}} = \emptyset$ (the set of unassigned tasks, initially empty)
7 **for** $\tau_i \in (\Gamma - \Gamma_s$, *sorted by decreasing utilization*$)$ **do**
8 Try to assign $s_{i,k} = u_i$ of task τ_i on π_k using FF;
9 **if** *FF fails for all* $\pi_k \in \Pi$ **then**
10 $\Gamma_{\text{na}} = \Gamma_{\text{na}} \cup \tau_i$;
11 k = M (start share assignment from processor π_M to π_1);
12 **for** $\tau_i \in \Gamma_{na}$ **do**
13 $u_{\text{remaining}} = u_i$;
14 **while** $u_{remaining} > 0$ **do**
15 $s_{i,k} = \min(u_{\text{remaining}}, (\alpha - \sigma_k))$;
16 $\sigma_k = \sigma_k + s_{i,k}$;
17 $u_{\text{remaining}} = u_{\text{remaining}} - s_{i,k})$;
18 **if** $\sigma_k = \alpha$ **then**
19 k = k-1

Third step. The final step assigns all the remaining tasks, which could not be allocated as fixed tasks. Considering the processor list in reversed order $\{\pi_M, \pi_{M-1}, \cdots \pi_1\}$, task $\tau_i \in \Gamma_{\text{na}}$ 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.

The three steps described above are implemented in Algorithm 3. In particular, the first step is represented in lines (1-5), the second step in lines (6-10), the third and final step in lines (11-17).

Example 6.7.2. Consider the SDF graph example in Figure 6.2(a). In Example 6.6.1, we derived the corresponding task set $\Gamma = \{\tau_1 = (2, 6), \tau_2 = (3, 3), \tau_3 = (2, 6)\}$. Tasks τ_1 and τ_2 are stateful, whereas task τ_3 is stateless. The total utilization of the task set is



(a) $\alpha = 0.75$. (b) $\alpha = \alpha_{\text{opt}} = 5/9$.

Figure 6.5: *Share assignments considered in Example 6.7.2. Values of the normalized system speed α are denoted by a dashed line. Migrating tasks are indicated in gray. In sub-figure (a), the normalized system speed α is set to the lowest value that guarantees schedulability and is supported by the system, i.e., $\alpha = 0.75$. In sub-figure (b), we use the periodic frequency switching technique described in Section 6.7.4 to get the normalized speed $\alpha = \alpha_{opt} = 5/9$ in the long run. The technique represented in sub-figure (b) leads to additional energy savings due to an even distribution of tasks shares among processors.*

$U_\Gamma = 1/3 + 1 + 1/3 = 5/3$. Assume that we want to execute this task set on $M = 3$ processors. By Condition (6.1), $\alpha \geq U_\Gamma / M = 5/9$, therefore the lowest α which could provide schedulability is $\alpha_{\text{opt}} = 5/9$. 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 Section 6.7.3, which considers fixed processors speed.

However, suppose that the considered system supports a set of normalized speeds $\mathcal{NS} = \{0.25, 0.5, 0.75, 1\}$. Note that $\alpha_{\text{opt}} \notin \mathcal{NS}$. In this case, we have two choices. *Choice 1)* We set the system speed to the lowest $\alpha \in \mathcal{NS}$ such that $\alpha > \alpha_{\text{opt}}$, condition which could provide schedulability: $\alpha = 0.75$. We can then refer again to Section 6.7.3 to derive tardiness bounds in this scenario. Figure 6.5(a) 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 Section 6.7.4 to get the normalized speed α_{opt} in the long run, and we derive the corresponding tardiness bounds. Figure 6.5(b) shows the assignment obtained when $\alpha = \alpha_{\text{opt}} = 5/9$.

6.7.2 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 (and, in turn, from EDF-fm) the job releasing techniques for migrating tasks (see Section 2.2.8). In particular, the job releasing technique uses the concept of *task fraction*, defined in Definition 2.2.12 on page 39, to avoid overloading the processors in the long run. For an example, refer to Figure 2.5 on page 41. That figure shows the job release pattern of tasks τ_3 on processors π_1 and π_2. Since task τ_3 has task fractions $\varphi_{3,1} = 3/4$ on processor π_1, and $\varphi_{3,2} = 1/4$ on processor π_2, in the long run the number of jobs of τ_3 released on π_1 is three times the number of jobs released on π_2. Moreover,

Inequality (2.24) on page 42, which provides an upper bound of the number of jobs released on a processor as a function of the migrating task fraction, is still valid. This result will be instrumental to the derivation of tardiness bounds under our EDF-ssl.

Job prioritization rules. Jobs of fixed and migrating tasks released on a certain processor are scheduled using a local EDF scheduler. As shown in Example 6.7.1, 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.

6.7.3 Tardiness Bounds under Fixed Processor Speed

Given the rules and properties of our EDF-ssl, described in Section 6.7.1 and Section 6.7.2, we now derive its tardiness bounds, which are provided by Theorem 6.7.1 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 6.7.1. *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*

$$\Delta^{\pi_k} = \frac{2\,(C_i + C_j)}{\alpha} \tag{6.3}$$

where C_i and C_j are the worst-case execution time of τ_i and τ_j, respectively, and α follows Definition 6.5.1.

Proof. We prove Theorem 6.7.1 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 (see Definition 2.2.1) 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 Inequality (2.24) on page 42.

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 Inequality (2.24) the demand[2] dmd from migrating tasks in $[t_0, t_c)$ is upper bounded by:

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

Given the definition of task fraction $\varphi_{i,k}$ (see Definition 2.2.12 on page 39), we obtain:

$$\mathrm{dmd}(\gamma_m, t_0, t_c) \leq (t_d - t_0)(s_{i,k} + s_{j,k}) + 2(C_i + C_j) \tag{6.4}$$

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

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

From Condition (6.2), we obtain:

$$\mathrm{dmd}(\gamma_f, t_0, t_c) \leq (t_d - t_0)(\alpha - s_{i,k} - s_{j,k}) \tag{6.5}$$

Combining Inequality (6.4) and (6.5), we derive an upper bound for the total demand of fixed and migrating tasks in $[t_0, t_c)$:

$$\mathrm{dmd}(\gamma_f \cup \gamma_m, t_0, t_c) \leq \alpha(t_d - t_0) + 2(C_i + C_j) \tag{6.6}$$

To ease our analysis, we now express the total demand from tasks in clock cycles. In fact, 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 $CC_a = C_a \cdot F_{\max}$. This is because the worst-case execution *time* C_a of τ_a is obtained at the maximum processor frequency, $F_{\max}$ (see definition of $F_{\max}$ in Section 6.5).

Then, from Inequality (6.6) we get:

$$\mathrm{dmd\_cc}(\gamma_f \cup \gamma_m, t_0, t_c) \leq F_{\max}\left(\alpha(t_d - t_0) + 2(C_i + C_j)\right) \tag{6.7}$$

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

[2] See Definition 2.2.1.

$[t_0, t_c)$ 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 Inequality (6.7), whereas the amount of clock cycles provided by processor π_k is $\alpha \cdot F_{\max}(t_c - t_0)$, because π_k runs at frequency $\alpha \cdot F_{\max}$. Therefore, we have:

$$\alpha \cdot F_{\max}(t_c - t_0) < F_{\max}\left(\alpha(t_d - t_0) + 2(C_i + C_j)\right) \tag{6.8}$$

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

$$t_c < t_d + 2(C_i + C_j)/\alpha \;\Rightarrow\; t_c < t_d + \Delta^{\pi_k} \tag{6.9}$$

Expression (6.9) contradicts the earlier definition of $t_c = t_d + \Delta^{\pi_k}$, therefore Theorem 6.7.1 holds. ■

Note that the tardiness bound given by Equation (6.3) differs from the tardiness bounds of EDF-os given by Equation (3) and (10) in [AEDC14]. This is caused by the differences in the *execution* phase between the two scheduling algorithm described in Section 6.7.2.

6.7.4 Tardiness Bounds under PWM Scheme

The optimal normalized speed α_{opt}, which can minimize the energy consumption while guaranteeing schedulability, is derived from the lower bound in Expression (6.1). This α_{opt}, however, often may not be supported by the system. Example 6.7.2 shows such a case. Recall that by Def. 6.5.1, 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 [B$^+$09], which we summarize in the following subsection. Note that other research works have considered the problem of providing the optimal processor speed in processors that only provide a discrete set of frequencies. See, for instance, [IY98]. However, in our work we choose the technique proposed in [B$^+$09] because it is accurate (it considers the overheads incurred during voltage/frequency switching) and it uses the real-time periodic task model (contrary to [IY98]).

PWM Scheme

The PWM scheme presented in [B$^+$09] is aimed at uniprocessor systems with HRT constraints. The execution of the scheme at run-time is sketched in Figure 6.6. 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, [B$^+$09] 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.

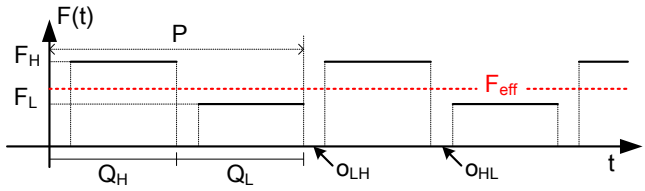


Figure 6.6: *Execution of the PWM scheme. The scheme switches periodically between a lower frequency F_L and a higher frequency F_H, in order to provide an effective frequency F_{eff} in the long run. The period of the PWM scheme is denoted by P.*

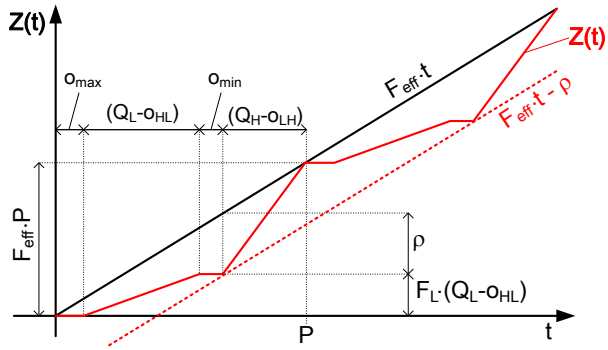


Figure 6.7: *Supply function $Z(t)$. This function provides the minimum number of cycles executed by the processor under the PWM scheme in every time interval of length t.*

As shown in Figure 6.6, 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, [B+09] denotes the amount of clock cycles lost during frequency transitions as $\Delta_{\text{LH}} = F_L \cdot o_{\text{HL}} + F_H \cdot o_{\text{LH}}$.

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 [B+09]:

$$F_{\text{eff}} = \lambda_L F_L + \lambda_H F_H - \Delta_{\text{LH}}/P \tag{6.10}$$

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 Figure 6.7, 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.

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 6.7.2. *Consider a processor π_k, on which the PWM scheme described in Section 6.7.4 is applied to obtain an effective frequency F_{eff}. Assume that two migrating tasks, τ_i (with 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*

$$\Delta_{PWM}^{\pi_k} = \frac{2(C_i + C_j)}{\alpha_{eff}} + \frac{\rho}{F_{eff}} \tag{6.11}$$

with $\rho = (F_{eff} - F_L)Q_L + F_L \cdot o_{HL} + F_{eff} \cdot o_{LH}$ and α_{eff} is derived from F_{eff} using Definition 6.5.1.

Proof. To prove Theorem 6.7.2, 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 the interval $[0, t]$ by a processor running at F_{eff}, and $Z(t)$. From Figure 6.7 we get:

$$\rho = (F_{\text{eff}} - F_L)Q_L + F_L \cdot o_{\text{HL}} + F_{\text{eff}} \cdot o_{\text{LH}} \tag{6.12}$$

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

$$\check{Z}(t) = F_{\text{eff}} \cdot t - \rho \tag{6.13}$$

$\check{Z}(t)$ is depicted in Figure 6.7 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 6.7.1. 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 the same as in the proof of Theorem 6.7.1. The demand from fixed and migrating tasks, expressed in clock cycles, is still bounded by Expression (6.7). However, we have to change the left-hand side of Inequality (6.8) with $Z(t_c - t_0)$, obtaining:

$$F_{\text{eff}} \cdot (t_c - t_0) - \rho + e(t_c - t_0) < F_{\text{max}} \left(\alpha_{\text{eff}}(t_d - t_0) + 2(C_i + C_j)\right) \tag{6.14}$$

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

$$(t_c - t_d) < \frac{2(C_i + C_j)}{\alpha_{\text{eff}}} + \frac{\rho - e(t_c - t_0)}{F_{\text{eff}}} = \Delta_{\text{PWM}}^{\pi_k} - \frac{e(t_c - t_0)}{F_{\text{eff}}} \tag{6.15}$$

Even with $e(t_c - t_0) = 0$, which represents the worst case, Expression (6.15) contradicts the definition of t_c, therefore Theorem 6.7.2 holds. ■

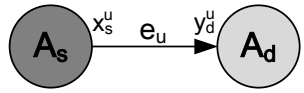


Figure 6.8: *SDF actors A_s (source) and A_d (destination) with dependency over edge e_u. The production rate of A_s over e_u is denoted by x_s^u. The consumption rate of A_d over e_u is denoted by y_d^u.*

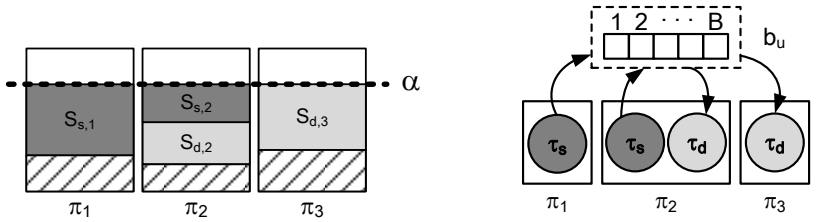
Note that by Equation (6.11) it follows that tardiness can be experienced even on processors with no migrating tasks, given the fact that the term ρ depends only on the parameters of the PWM scheme.

6.8 Start times and buffer sizes under EDF-ssl

As mentioned in Section 6.6, the analysis described in this chapter leverages the results of Chapter 5 to schedule the applications using our EDF-ssl soft real-time scheduler. However, compared to the EDF-fm scheduling algorithm used in Chapter 5, our EDF-ssl is different in certain aspects. In order to maintain the scheduling analysis valid for our proposed EDF-ssl, we must take into account these differences between our EDF-ssl and EDF-fm.

Let us consider the data-dependent actors A_s (source) and A_d (destination) shown in Figure 6.8. We recall that in our analysis A_s and A_d are converted into two periodic tasks τ_s and τ_d using the methodology described in Section 2.3. 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 Figure 6.9(a). 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 Figure 6.9(a), 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 Equation (6.3) 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 Chapter 5, 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 Figure 6.9(b), 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 Figure 6.9(b), buffer b_u of size B implements the communication



(a) Considered share assignment of τ_s and τ_d.

(b) Scheme of the communication between τ_s and τ_d over b_u, according to the share assignment given in sub-figure (a).

Figure 6.9: *Analysis of the communication between data-dependent actors when both source (τ_s) and destination (τ_d) actors are implemented as migrating tasks. In sub-figure (a), 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. Sub-figure (b) represents the access to the shared communication buffer b_u by jobs of τ_s and τ_d. Under our EDF-ssl, jobs of τ_s may execute in parallel if released onto different processors and be affected by a different tardiness depending on which processor execute them. The same holds for jobs of τ_d. It follows that jobs of τ_s may write out-of-order to buffer b_u. Similarly, jobs of the destination task τ_d may read from b_u out-of-order. This phenomenon is taken into account by our analysis.*

over edge e_u of Figure 6.8. 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 Chapter 5 to our EDF-ssl. Second, we define a 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 2.2.10 on page 38, 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:

$$\Delta_i = \begin{cases} \max_{k \mid s_{i,k}>0}\{\Delta^{\pi_k}\} & \text{under fixed processor speed} \\ \max_{k \mid s_{i,k}>0}\{\Delta^{\pi_k}_{PWM}\} & \text{under PWM scheme} \end{cases} \tag{6.16}$$

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 Section 6.7.3 and Section 6.7.4, respectively. For each processor π_k, Δ^{π_k} and $\Delta^{\pi_k}_{PWM}$ are obtained using Equation (6.3) and Equation (6.11), respectively. Finally, in Equation (6.16), $s_{i,k}$ represents the share of τ_i on π_k.

By using the tardiness bound Δ_i expressed by Equation (6.16), we can represent the ALAP completion schedule (see Definition 5.5.1 in Section 5.5.1) of actor A_i (corresponding to task τ_i) as a fictitious actor $\tilde{A}_i$, which has the same period as A_i, no tardiness, and start time $\tilde{S}_i = S_i + \Delta_i$. From Equation (6.16) it follows that at run time

Algorithm 4: Write pattern of job j of source task τ_s.

Input: Number of produced tokens x_s^u, job index j, buffer size B.
1 bgn $= [(x_s^u \cdot j) \mod B] + 1$;
2 end $= (x_s^u \cdot (j+1)) \mod B$;
3 **if** *bgn* < *end* **then**
4 write x_s^u tokens from b_u[bgn] to b_u[end]
5 **else**
6 write *(bgn-B+1)* tokens from b_u[bgn] to b_u[B];
7 write *remaining* tokens from b_u[1] to b_u[end];

any invocation $A_{i,j}$ of actor A_i will never be completed later than the corresponding invocation $\tilde{A}_{i,j}$ of actor $\tilde{A}_i$, regardless of which processor is executing that invocation. Therefore, the analysis for start times and buffer sizes in the presence of tardiness described in Chapter 5 can be applied considering the tardiness bounds given by Equation (6.16) and it is correct for our EDF-ssl.

Part 2 - Reading/writing pattern to/from b_u. Let us focus on the source actor A_s in Figure 6.8, and let assume the share assignment shown in Figure 6.9(a). Under our EDF-ssl, jobs of τ_s, which correspond to invocations of A_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 Figure 6.9(b). 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 the scenario described above, 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 A_s/A_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_s^u, 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 memory locations that would be written if the jobs of τ_s wrote in-order to a FIFO buffer of size B implemented as a circular buffer. This writing pattern is implemented in Algorithm 4. Lines 5-7 in the algorithm handle the case in which the x_s^u tokens are "wrapped" in the buffer. Note that by replacing, in Algorithm 4, x_s^u with y_d^u and write operations with read operations we obtain the reading pattern corresponding to job j of destination task τ_d.

Finally, note that under EDF-ssl, as in EDF-os, the job index j is maintained by the scheduling algorithm in order to release a migrating task on the right processor, in order to follow the job releasing rules mentioned in Section 6.7.2. Therefore, the value of j, used in Algorithm 4, is known when a job is executed on a processor.

6.9 Evaluation

In this section, we evaluate the effectiveness of our EDF-ssl semi-partitioned scheduling 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 [AY03]. The authors in [AY03] also show that the most balanced distributions are derived when Worst Fit Decreasing (WFD) heuristic (see Section 2.2.6) 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 Chapter 5. 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 Chapter 5, 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 soft real-time scheduling technique of Chapter 5, 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.

6.9.1 Power Model

As mentioned in Section 6.5, 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 [ZR13]. 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:

$$\mathcal{NS}_{\text{A9}} = \{0.292, 0.583, 0.767, 1.0\}$$

We use the power model of a similar dual Cortex-A9 core system, considered in [P+13], which we normalize to a single core:

$$p_{\text{cpu}} = p_{\text{dyn}} + p_{\text{sta}} = (0.223 V_{\text{cpu}}^2 F_{\text{cpu}}) + (K_1 V_{\text{cpu}} + K_2) \tag{6.17}$$

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. Note that the power model given in Expression (6.17) has been validated with actual power measurements in [P+13]. 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 also that Expression (6.17) assumes that the processor runs at one of the supported normalized speeds $\alpha_i \in \mathcal{NS}_{\text{A9}}$. From this α_i, we can derive the processor frequency $F_{\text{cpu}} = F_i$ by Definition 6.5.1. 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 $p_{\text{dyn}}(\alpha_i)$, $p_{\text{sta}}(\alpha_i)$, and $p_{\text{cpu}}(\alpha_i)$.

6.9.2 Energy per Iteration Period

Based on the power model expressed by Equation (6.17), 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 Equation (2.28) on page 44). 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, Equation (2.28)). 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:

$$E_{\text{sta}}^{H,\text{FIX}} = H \cdot p_{\text{sta}}^{\text{FIX}} = H \cdot M_{\text{ON}} \cdot p_{\text{sta}}(\alpha_i) \tag{6.18}$$

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 $p_{\text{dyn}}(\alpha_i)$. Therefore, the dynamic energy consumed by task τ_j during one iteration period H is:

$$E_{\text{dyn}}^{H,\text{FIX}}(\tau_j) = q_j \frac{C_j}{\alpha_i} p_{\text{dyn}}(\alpha_i) \tag{6.19}$$

From Equation (6.19) we derive the dynamic energy consumed in one iteration period H by the whole task set as follows:

$$E_{\text{dyn}}^{H,\text{FIX}} = \sum_{\tau_j \in \Gamma} q_j \frac{C_j}{\alpha_i} p_{\text{dyn}}(\alpha_i) = \frac{p_{\text{dyn}}(\alpha_i)}{\alpha_i} \sum_{\tau_j \in \Gamma} q_j C_j \tag{6.20}$$

Total energy of (PAR, SP). From Equation (6.18) and (6.20) we derive the total energy consumed during one iteration period H under (PAR, SP) by:

$$E_{\text{tot}}^{H,\text{FIX}} = H \cdot M_{\text{ON}} \cdot p_{\text{sta}}(\alpha_i) + \frac{p_{\text{dyn}}(\alpha_i)}{\alpha_i} \sum_{\tau_j \in \Gamma} q_j C_j \tag{6.21}$$

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 Equation (6.21) 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 Equation (6.17), 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 [B+09], reported below.

$$p_{\text{cpu}}^{\text{PWM}} = \lambda_L p_L + \lambda_H p_H + E_{\text{SW}}/P \tag{6.22}$$

where $E_{\text{SW}} = e_{\text{LH}} - p_H \cdot o_{\text{LH}} + e_{\text{HL}} - p_L \cdot o_{\text{HL}}$, 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 Section 6.7.4, 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 $o_{\text{LH}} = o_{\text{HL}} = 10\,\mu s$. Note that these values are compatible with the findings in [P+13], where the time and energy overheads due to frequency switching have been derived using actual measurements. Now, given the number of active cores M_{ON}, we can express the total energy per iteration period H under PWM as:

$$E_{\text{tot}}^{H,\text{PWM}} = H \cdot M_{\text{ON}} \cdot p_{\text{cpu}}^{\text{PWM}} \tag{6.23}$$

Note that Equation (6.23) depends on Equation (6.22), 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 Equation (6.10), 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 Equation (6.10); 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 Equation (6.23).

6.9.3 Experimental Results

In Table 6.1, we show the results obtained using the considered approaches (PAR, SP, PWM) on a set of real-life applications (see column App). For each application, column U_Γ reports the cumulative utilization of the corresponding task set. In addition,

Table 6.1: *Comparison of different share allocation/scheduling approaches.*

App	U_Γ	R [tkns/s]	$\hat{M}$	PAR				SP				PWM			
				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	12000	4	3	22.7	0.667	$9.67 \cdot 10^{-5}$	4	1.37	1.62	0.77	0.61	2.18	3.85	0.66
			8	3	22.7	0.667	$9.67 \cdot 10^{-5}$	5	1.64	2.22	0.64	0.49	3.02	5.95	0.61
			12	3	22.7	0.667	$9.67 \cdot 10^{-5}$	9	3.14	5.25	0.37	0.27	na	na	na
JP2	1.23	333.33	4	2	114	17.5	$1.78 \cdot 10^{-3}$	3	1.42	1.28	0.62	0.41	1.9	3.45	0.51
			8	2	114	17.5	$1.78 \cdot 10^{-3}$	5	2.28	2.13	0.36	0.25	na	na	na
			12	2	114	17.5	$1.78 \cdot 10^{-3}$	5	2.28	2.13	0.36	0.25	na	na	na
MJPEG	1.22	6000	4	2	62.7	0.833	$7.50 \cdot 10^{-5}$	3	1.31	1.42	0.84	0.41	1.94	4.73	0.69
			8	2	62.7	0.833	$7.50 \cdot 10^{-5}$	5	1.8	3.17	0.47	0.24	na	na	na
			12	2	62.7	0.833	$7.50 \cdot 10^{-5}$	5	1.8	3.17	0.47	0.24	na	na	na
MPEG2	6.81	12000	8	8	345	1.50	$2.70 \cdot 10^{-4}$	7	1.44	1.55	0.99	0.97	1.84	2.00	1.00
			12	8	345	1.50	$2.70 \cdot 10^{-4}$	12	2.01	2.21	0.63	0.57	2.45	3.01	0.65
TDE	6.28	3000	8	7	840	9.67	$9.90 \cdot 10^{-4}$	7	1.00	1.00	1.00	0.9	1.42	1.42	0.94
			12	9	840	9.67	$7.60 \cdot 10^{-4}$	11	1.68	1.76	0.83	0.57	1.68	1.78	0.84

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 Equation (2.26) on page 43, 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 6.1 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 $\alpha_i \in \mathcal{NS}_{\text{A9}}$ which guarantees schedulability ($\min_{\alpha_i \in \mathcal{NS}_{\text{A9}}}\{\alpha_i \geq \sigma_k, \forall \text{ active } \pi_k\}$). Knowing M_{ON} and α_i, we determine the total energy per iteration period E^H_{PAR} by Equation (6.21).
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 $\mathcal{NS}_{\text{A9}}$, which we denote with α_i. We run Algorithm 3 with this speed value α_i and $M = M_{\text{ON}}$. If Algorithm 3 is successful, we determine the total energy per iteration period E^H_{SP} by Equation (6.21) with the considered α_i and M_{ON}.
In PWM, we calculate $\alpha_{\text{opt}} = U_\Gamma / M_{\text{ON}}$ and we run Algorithm 3 with speed value α_{opt} and $M = M_{\text{ON}}$. If Algorithm 3 is successful, we use $\alpha_H = \min_{\alpha_i \in \mathcal{NS}_{\text{A9}}}\{\alpha_i \geq \alpha_{\text{opt}}\}$ and $\alpha_L = \max_{\alpha_i \in \mathcal{NS}_{\text{A9}}}\{\alpha_i \leq \alpha_{\text{opt}}\}$ and derive the total energy per iteration period E^H_{PWM} by Equation (6.23).

For each valid task share assignment, we derive earliest start times of actors and buffer size requirements by using the formulas derived in Chapter 5 with, for each task τ_l, one of the following tardiness bound values: $\Delta_l = 0$ in PAR; Δ_l obtained by Equation (6.16) in SP and PWM.

At the end of the design space exploration, for PAR and SP, we report in Table 6.1 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 6.1 with the couple $(\text{App}, \hat{M})$. 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}, calculated using the latency analysis described in Section 4.7 of [Bam14]; the energy consumption E^H_{PAR}. In particular, the

total memory requirement in the PAR approach is calculated as follows.

$$\mathrm{TM}_{\mathrm{PAR}} = \sum_{i=1}^{N} \mathrm{CSS}(\tau_i) + \sum_{i=1}^{|E|} b_u^{HRT} \tag{6.24}$$

where N is the number of tasks, $\mathrm{CSS}(\tau_i)$ is the code and stack size of task τ_i (which represents actor A_i of the input CSDF graph G), E is the set of edges in G, b_u^{HRT} is the size of the buffer that implements the communication over edge e_u. The value of b_u^{HRT} assumes no task tardiness and is obtained using Equation (2.31) on page 46.

By contrast, for the semi-partitioned approach SP, the total memory requirement $\mathrm{TM}_{\mathrm{SP}}$ is derived using the following expression.

$$\mathrm{TM}_{\mathrm{SP}} = \sum_{i=1}^{M_{\mathrm{SP}}^o} \sum_{\tau_j \in \Gamma_i} \mathrm{CSS}(\tau_j) + \sum_{i=1}^{|E|} b_u^{SRT} \tag{6.25}$$

where M_{SP}^o is the number of processors (derived in the design space exploration), Γ_j is the set of tasks with non-zero shares on processor π_j, and b_u^{SRT} is the size of the buffer that implements the communication over edge e_u, calculated using Equation (5.2). Note that Equation (6.25) differs from Equation (6.24) because in the SP approach a task can have shares on different processors. In addition, in order to derive the application latency under SP, denoted by L_{SP}, we use the analysis in Section 4.7 of [Bam14], considering the task start times obtained by our SRT approach. Then, we add to that latency value the tardiness (which can be potentially null) of the output actor of the application.

For the PWM approach, the total memory requirement $\mathrm{TM}_{\mathrm{PWM}}$ and application latency L_{PWM} are derived following the same procedure used for the SP approach.

We see from Table 6.1 that SP consumes significantly lower energy than PAR, see column $E_{\mathrm{SP}}^H / E_{\mathrm{PAR}}^H$. On average, we obtain an energy saving of 36%. The energy saving goes up to 64%, see row $(\mathrm{JP2}, 8)$. These energy savings, however, come at a cost. The total memory requirements (see column $\mathrm{TM}_{\mathrm{SP}} / \mathrm{TM}_{\mathrm{PAR}}$) and application latencies (see column $L_{\mathrm{SP}} / L_{\mathrm{PAR}}$) are increased. Memory requirements increase due to *i)* more task replicas (with their code 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 6.1 presents the results under PWM. It shows that this approach can provide higher energy savings compared to SP (compare columns $E_{\mathrm{PWM}}^H / E_{\mathrm{PAR}}^H$ and $E_{\mathrm{SP}}^H / E_{\mathrm{PAR}}^H$). 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 $\mathcal{NS}_{\mathrm{A9}}$. Therefore, the PWM scheme is not applicable. Note that in three rows the value of $E_{\mathrm{PWM}}^H / E_{\mathrm{PAR}}^H$ is higher than $E_{\mathrm{SP}}^H / E_{\mathrm{PAR}}^H$. 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 $\mathcal{NS}_{\mathrm{A9}}$, 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 TM_{PWM}/TM_{PAR} and L_{PWM}/L_{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 6.1, 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 6.1. 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 Section 6.7, 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 $2.67 \cdot 10^{-4} J$ (compared to $E^{H}_{PAR} = 2.70 \cdot 10^{-4} J$). 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.

6.10 Discussion

In this chapter, we have proposed EDF-ssl, a soft real-time semi-partitioned scheduling algorithm aimed at reducing the energy consumption of embedded multiprocessor streaming systems with throughput constraints. Our EDF-ssl exploits the presence of some stateless tasks in the application, allowing the execution of different jobs of the same task in parallel, and achieving 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.

As shown in Section 6.9, our semi-partitioned scheduling approach achieves significant energy savings compared to a purely partitioned scheduling approach and an existing semi-partitioned one, EDF-os. The energy savings are on average 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, an additional energy saving up to 18% is obtained. 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.