Universiteit Leiden
The Netherlands

Semi-partitioned scheduling and task migration in dataflow networks

Cannella, E.

Citation

Cannella, E. (2016, October 11). *Semi-partitioned scheduling and task migration in dataflow networks*. Retrieved from https://hdl.handle.net/1887/43469

Version:	Not Applicable (or Unknown)
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/43469

Note: To cite this publication please use the final published version (if applicable).

Cover Page

Universiteit Leiden

The handle http://hdl.handle.net/1887/43469 holds various files of this Leiden University dissertation

Author: Cannella, Emanuele
Title: Semi-partitioned scheduling and task migration in dataflow networks
Issue Date: 2016-10-11

Chapter 5

Semi-partitioned Scheduling of CSDF-modeled Streaming Applications

Most of the work presented in this chapter has been published in [CBS14].

THIS chapter and the following one, Chapter 6, present two methodologies that exploit semi-partitioned scheduling algorithms, in the context of **hard real-time streaming systems**, using the scheduling analysis proposed in [BS11, BS12] as a basis and research driver. In particular, the semi-partitioned approach proposed in this chapter is aimed at reducing the number of processors required to schedule those applications which incur bin-packing issues under the partitioned scheduling approach of [BS11, BS12]. We recall that by bin-packing issues we mean that those applications require, using a partitioned scheduling approach, more processors than the minimum achievable by a global optimal scheduler.

In order to clarify the contributions of this chapter, we depict in Figure 5.1 the scheduling framework proposed in [BS11, BS12]. As input to this framework, the designer provides the application model, in the form of a (C)SDF graph with N actors (see *Analysis Model* in Figure 5.1). Then, Step ① converts the N actors of the input application into N periodic real-time tasks and derives the minimum size of the buffers which implement inter-task communication. Throughout this chapter, we refer to this conversion as *scheduling analysis of [BS11, BS12]*. This conversion is described in Section 2.3, and assumes that a hard real-time (HRT) scheduler will be used to execute the derived task set. In particular, Step ① derives:

- I The worst-case execution time (WCET) C_i of each task, using Equation (2.26) on page 43.
- II The period T_i of each task, using Equation (2.27) on page 43.

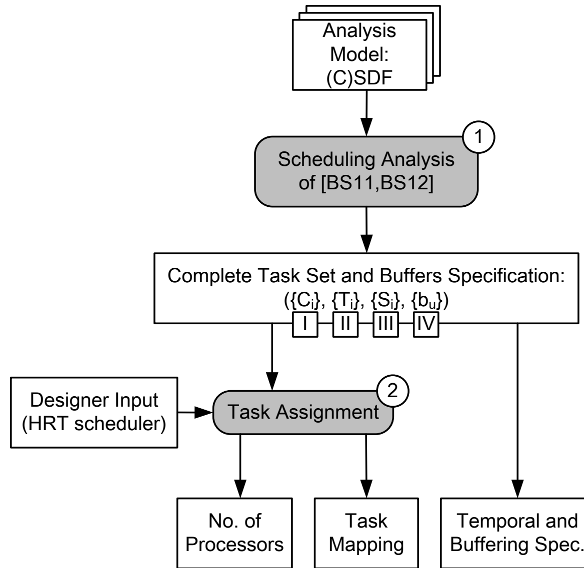


Figure 5.1: *Scheduling framework proposed in [BS11,BS12]. Step ① of the framework converts the input application, modeled as a CSDF graph, to a corresponding set of real-time periodic tasks. The obtained real-time periodic task set is completely specified, i.e., the WCET* I*, period* II*, and start time* III *of each task are known, together with the required size of buffers* IV *through which the tasks communicate. Then, based on the WCET and period of tasks, and on the designer's choice of HRT scheduling algorithm, Step ② derives the required number of processors and the assignment of task to processors. At this point, the system is completely specified.*

III The start time S_i of each task, using Equation (2.29) on page 44.
IV The size b_u of each buffer, which implements the communication over edge e_u = (A_i, A_j). This size is obtained using Equation (2.31) on page 46.

The next step, Step ② (*Task Assignment*), derives the minimum number of processors required to schedule the application and the assignment of tasks of the application to processors, using the HRT partitioned approach (see Section 2.2.5) chosen by the designer. The assignment is based on the WCET and period of each task (parameters I and II above) derived in Step ①.

At the end of Step ②, the system is completely specified. The system specification consists of the following components.

- **Number of Processors.** It represents the number of processors required to schedule the application (obtained in Step ②).
- **Task Mapping.** It describes the assignment of application's tasks to processors (obtained in Step ②).
- **Temporal and Buffering Specification.** It describes the parameters of the task set, together with the size of buffers which implement inter-task data communication (all of which are obtained in Step ①).

5.1 Proposed Extension of the Scheduling Framework of [BS11, BS12]

So far, the scheduling framework of [BS11, BS12] (see Figure 5.1) considers only **hard real-time *partitioned* scheduling algorithms**. *Partitioned* scheduling algorithms incur neither migration overhead nor memory overhead because each task is statically allocated to a single processor. However, these algorithms are affected by *bin-packing issues* [Joh73], as described in Section 2.2.5. That is, if no limit on the maximum task utilization of a task set is imposed, partitioned algorithms may require twice as many processors compared to an optimal global scheduler [LDG04]. Therefore, for some applications, the number of processors required by partitioned approaches is larger than the number of processors required by optimal global schedulers.

However, also optimal global schedulers present drawbacks. Recall, from Section 2.2.5, that under optimal *global* scheduling algorithms all the tasks can migrate among all the processors. Such algorithms can fully exploit the available computational resources (refer again to Section 2.2.5, and in particular to Expression (2.13) on page 34). However, their optimality comes at the cost of high scheduling overheads due to excessive task preemptions and migrations. Moreover, modern MPSoCs typically have distributed memories in order to avoid the unpredictability of accessing shared resources. Therefore, using a global scheduler on such distributed memory systems implies that the code of each task has to be replicated[1] on all the processors, incurring a large memory overhead.

Semi-partitioned algorithms represent a middle ground between global and partitioned scheduling algorithms. In fact, under semi-partitioned approaches, most of the tasks are statically allocated to processors, and only a subset of the tasks is allowed to migrate among different processors. Migrating tasks follow a migration pattern derived at design-time. By allowing a (usually small) subset of tasks to migrate, semi-partitioned scheduling algorithms can mitigate the bin-packing effects that affect partitioned approaches. As a result, semi-partitioned algorithms require less processors than partitioned algorithms to schedule certain task sets. At the same time, these algorithms do not incur large memory overheads and task migration/preemption overheads like global algorithms. For these reasons, **in this chapter we extend the scheduling framework of [BS11, BS12] in order to support a *semi-partitioned* approach**. In the next section we explain the reasons why, among the various semi-partitioned schedulers, our proposed approach uses EDF-fm [ABD08]. We recall that the name EDF-fm comes from the fact that the algorithm is based on EDF and allows tasks to be either *fixed* or *migrating*.

5.1.1 Choice of the EDF-fm Semi-partitioned Algorithm

Several semi-partitioned scheduling algorithms have been proposed [ABD08, DYGR10, GCS11, KY08, AT06]. These algorithms can be classified based on *when* migrations

[1] We assume task migration using code replication, as shown in Chapter 4, because in distributed memory MPSoC systems it guarantees faster completion of the migration procedure.

are allowed to occur. In *restricted-migration* approaches [ABD08, DYGR10, GCS11] migrations can happen at job boundaries only. In *unrestricted-migration* (or *portioned*) approaches, migrations can happen at any time during a job execution. We argue that the restricted-migration class of semi-partitioned schedulers is the most suitable for distributed memory MPSoCs. This is because migrating at job boundaries reduces the amount of data (state) to be transferred from one processor to the next. Moreover, if the task does not keep an internal state between two successive jobs (i.e., it corresponds to a *stateless* dataflow actor, as defined in Definition 2.3.6 on page 47), no state migration is needed.

Within the class of restricted-migration semi-partitioned approaches, EDF-fm [ABD08] is particularly suited to distributed memory systems because in that approach a migrating task is allowed to migrate only between two processors (contrary to [DYGR10, GCS11], in which migrating tasks may span among all the processors). This property reduces substantially the memory overhead caused by replicating the task code. In addition, EDF-fm uses a fast utilization-based schedulability test (contrary to [DYGR10, GCS11]), that can be easily executed at run-time for incoming applications. For the reasons explained above we employ EDF-fm in the work presented in this chapter.

5.1.2 Implications of Using EDF-fm

Although EDF-fm can have great benefits for distributed memory MPSoCs, it provides hard real-time guarantees only for migrating tasks and soft real-time (SRT) guarantees for fixed tasks. Recall that, by Definition 2.2.9 on page 38, this means that fixed tasks can miss their deadlines by a bounded value called *tardiness*. As a consequence, the scheduling analysis of [BS11, BS12] can not be used directly because it assumes that a hard real-time (HRT) scheduler will schedule the derived task set, such that all task deadlines are met. It follows that the scheduling framework depicted in Figure 5.1 has to be modified.

Although our proposed semi-partitioned approach uses a SRT scheduling algorithm, in this chapter we provide a technique which ensures that the input/output interfaces with the environment are not affected by the deadline misses which may occur to the tasks of the application. That is, we can provide HRT guarantees to the input and output interfaces of the application with the environment.

5.2 Problem Statement

The scheduling analysis of [BS11, BS12] shows that an application, modeled as an acyclic CSDF graph, can be scheduled using a **hard real-time partitioned scheduling algorithm** as a set of real-time periodic tasks. In this chapter, we investigate the applicability of the **soft real-time *semi-partitioned* scheduling algorithm EDF-fm** to the scheduling analysis of [BS11, BS12]. In order to do so, we need to modify that scheduling analysis in a way that soft real-time scheduling algorithms can be supported. Overall, our semi-partitioned approach is aimed at reducing the number

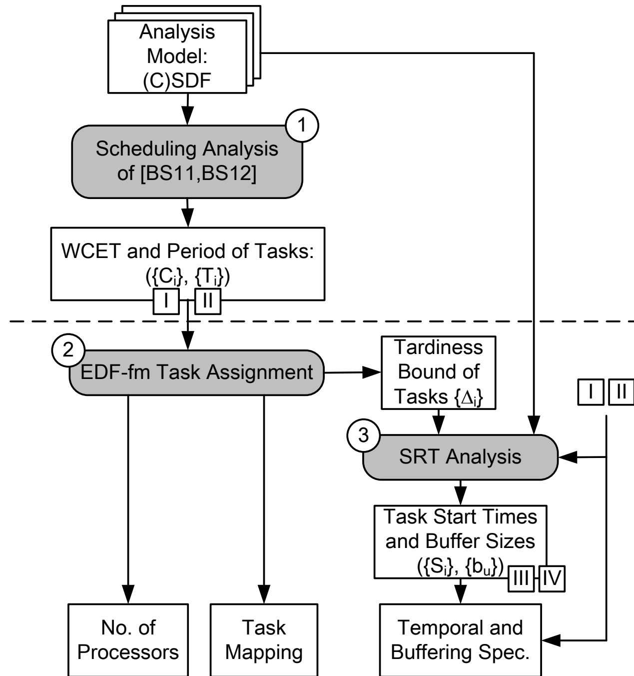


Figure 5.2: *Scheduling framework proposed in this chapter, which assumes that the SRT scheduling algorithm* EDF-fm *will schedule the derived task set, instead of the HRT partitioned schedulers considered in Figure 5.1. The part of the scheduling framework above the dashed line is identical to the scheduling framework in Figure 5.1. However, in Step ① only the WCET* I *and period* II *of each task are derived. This is because the start times* III *of tasks and required size of buffers* IV *through which the tasks communicate can only be derived when the tardiness bound* Δ_i *of each task* τ_i *is known. Step ② of the scheduling framework derives the minimum number of processors, the assignment of tasks to processors, and the tardiness bounds of tasks. Based on these tardiness bounds, Step ③ derives the task start times and buffer sizes. At this point, the system is completely specified.*

of processors required to schedule the applications that incur bin-packing issues under the partitioned scheduling approach of [BS11, BS12]. We recall that by *bin-packing issues* we mean that these applications require more processors under the partitioned scheduler compared to a global optimal scheduler.

5.3 Contributions

In order to address the problem stated in Section 5.2, we propose the scheduling framework shown in Figure 5.2. This scheduling framework is a modification of the one used in [BS11, BS12] and reported in Figure 5.1. The changes in the design flow are necessary in order to use the SRT semi-partitioned scheduler EDF-fm instead of a HRT partitioned approaches, as assumed in [BS11, BS12].

Consider Step ① of the design flow in Figure 5.1. That step converts the input application, modeled as a CSDF graph, into a set of real-time periodic tasks. In particular, it derives the complete specification of the tasks set (WCET [I], period [II], start time [III] of each task) and the size of the buffers [IV] which implement inter-task data dependencies. This analysis assumes that the derived periodic task set will be scheduled by a HRT scheduling algorithm, i.e., no task will miss any deadline. Therefore, as shown in Figure 5.1, the scheduling analysis of [BS11, BS12] can derive the complete *Temporal and Buffering Specification* of the system - composed of parameters [I], [II], [III], and [IV] - in a single step. In what follows, we will refer to such scheduling analysis, which assumes hard real-time scheduling, as *HRT approach*.

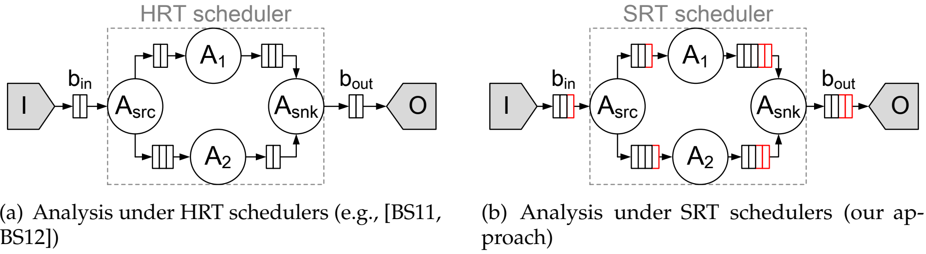
As a **first contribution** of this chapter, we show that this scheduling analysis can be extended to support SRT schedulers, once the tardiness bound[2] Δ_i of each task τ_i is known. In practice, the components of the *Temporal and Buffering Specification* of the system ([I], [II], [III], and [IV]) cannot be derived in a single step, but in the following three steps of Figure 5.2.

- Step ① derives the WCET and period of tasks (components [I] and [II]), based on the input application model. These components are not affected by the tardiness of tasks.
- Step ② assigns the tasks to processors, according to the rules of the chosen SRT scheduling algorithm (in this case, EDF-fm). The outputs of this step are the required number of processor, the assignment of tasks to processors, and the tardiness bound Δ_i of each task τ_i.
- Step ③, given the value of Δ_i of each task τ_i, derives the earliest task start times and minimum buffer sizes ([III] and [IV] in Figure 5.2) that guarantee the existence of a valid schedule of the given application. Valid schedule means that, even in presence of task tardiness, tasks can be released periodically and neither buffer underflow nor overflow can occur.

In what follows, we will refer to the scheduling analysis which assumes a SRT scheduler as *SRT approach*. In this chapter, we show that the SRT approach achieves the same throughput of the HRT approach, albeit requiring larger buffers and increased application latency. In Figure 5.3 we compare the HRT and SRT approaches. The mentioned increase in the size of buffers is visualized in Figure 5.3(b) using red color. By appropriately increasing the size of buffers, the analysis presented in this chapter guarantees that the interfaces of the environment with the application (see I and O in Figure 5.3(b)) can execute in a strictly periodic way with neither underflow nor overflow on input and output buffers (see b_{in} and b_{out} in Figure 5.3(b)), also when SRT schedulers are used. This means that the **input/output interfaces are not affected by the deadline misses which may occur to the tasks of the application, i.e., I and O in Figure 5.3(b) can execute with HRT guarantees.**

Then, using the result of the first contribution, we focus on a specific SRT semi-partitioned scheduling algorithm, namely EDF-fm [ABD08]. As mentioned in Section 5.2, we consider EDF-fm instead of the partitioned approaches adopted in [BS11,BS12] because we want to reduce the number of processors required to schedule

[2]See Definition 2.2.10 on page 38.



(a) Analysis under HRT schedulers (e.g., [BS11, BS12])

(b) Analysis under SRT schedulers (our approach)

Figure 5.3: *Scheduling framework under HRT (a) and SRT (b) schedulers. Sub-figure (a) represents the scheduling analysis of [BS11, BS12] which considers only HRT schedulers. By contrast, sub-figure (b) depicts the scheduling analysis when SRT schedulers are used. This kind of schedulers allow tasks to miss their deadlines up to a certain value. As shown in this chapter, real-time guarantees can be still provided to the interfaces with the environment (denoted by I and O). However, the SRT approach requires larger size of buffers (as highlighted in red in sub-figure (b)).*

the applications that incur bin-packing issues under that partitioned approach. As a **second contribution** of this chapter, we propose a novel task assignment heuristic, called FFD-SP (First Fit Decreasing followed by semi-partitioning), that replaces the ones proposed in [ABD08] that are intended for independent task sets. FFD-SP is executed in Step ② in Figure 5.2. We propose this novel task assignment heuristic for the reason described in the following paragraph.

As shown in Figure 5.2, the derivation of task start times and buffer sizes in Step ③ depends on the value of the tardiness bounds of tasks. In general, the more tasks are affected by tardiness, the more task start times need to be postponed. This has a direct effect on the latency of the application. Moreover, as the number of tasks with tardiness increases, so does the size of buffers required to implement inter-task communication. To summarize, the number of tasks affected by tardiness has a direct impact on the overhead in application latency and buffer sizes of the SRT approach compared to the HRT approach. This effect is not considered by the task assignment heuristics proposed in [ABD08] because those heuristics are intended for independent tasks. Our proposed FFD-SP heuristic is aimed at reducing the number of required processors, compared to the HRT approach, while keeping a low buffer size and latency overhead when the EDF-fm algorithm is used.

Finally, as a **third contribution**, we show on a set of real-life benchmarks that our SRT approach can lead to significant benefits by reducing the number of processors required to schedule the applications that incur in bin-packing issues under the HRT approach of [BS11, BS12]. At the same time, both our SRT approach and HRT approaches achieve the same application throughput. However, our experiments show that the SRT approach incurs an increase in memory requirements and application latency. Therefore, our SRT semi-partitioned approach is especially appealing for systems in which the throughput constraint is more important than memory or latency constraints.

5.4 Related Work

To the best of our knowledge, real-time semi-partitioned scheduling algorithms have never been studied when mapping streaming applications with inter-task data dependencies to MPSoCs. In fact, existing semi-partitioned solutions [ABD08, DYGR10, GCS11, KY08, AT06] only consider sets of independent tasks. In the real-time community, however, techniques different from pure partitioning to assign data-dependent application tasks to a multiprocessor platform have already been devised. Existing approaches which are close to our work are [LA09] and [LA10] by Liu and Anderson. These approaches use a global scheduler which, similar to our case, satisfies soft real-time requirements. In particular, [LA09] describes a way to guarantee bounded tardiness of an application specified as a pipeline of tasks under a SRT global scheduler. A strong limitation in [LA09] is that only simple pipeline application topologies are handled, contrary to our approach that can handle more complex topologies like CSDF graphs. In [LA10], the same authors extend their analysis to guarantee bounded task tardiness in more complex application graph topologies, such as Processing Graph Method (PGM) graphs. However, the work in [LA10] does not address the calculation of minimum buffer sizes, which is an important metric to evaluate the practicability of the approach. In contrast, the calculation of buffer sizes is supported by our approach.

5.5 Soft Real-time Scheduling Analysis

In this section, we present the first main contribution of this chapter. Our contribution is based on the scheduling analysis of [BS11, BS12] (see Section 2.3), which converts a CSDF to a set of periodic tasks, assuming that a HRT scheduler is used to schedule the derived task set. We show that such scheduling analysis can be modified in a way that SRT schedulers can be used to execute the derived periodic task set. The SRT scheduler considered in this chapter is the EDF-fm algorithm, whose per-task tardiness bound is given by Equation (2.22) on page 40. Note, however, that the results obtained in this section are valid for any SRT scheduler which provides bounded task tardiness. Our solution extends the analysis of [BS11, BS12] by deriving new earliest start times for each task (Section 5.5.1) and minimum buffer sizes (Section 5.5.2) that can handle task tardiness and still allow a periodic release of each task. Within the design flow proposed in Figure 5.2, this derivation of task start times and buffer sizes is performed in Step ③.

5.5.1 Earliest Start Times in Presence of Tardiness

In order to derive the earliest start times in presence of tardiness, we leverage some concepts which are explained in Chapter 2 of this thesis. We summarize these concepts below.

- Under hard real-time scheduling of acyclic CSDF graphs (Section 2.3), when computing the earliest start times of actors we use the cumulative produc-

tion/consumption functions defined in Definitions 2.3.1 and 2.3.2 on page 44, namely:

  – $\text{prd}^{\text{S}}{}_{[t_s,t_f)}(A_i, e_u)$, which represents the total number of tokens produced by actor S_i to edge e_u during the time interval $[t_s, t_f)$;
  – $\text{cns}^{\text{S}}{}_{[t_s,t_f]}(A_j, e_u)$, which represents the total number of tokens consumed by actor A_j from edge e_u during the time interval $[t_s, t_f]$.

- By Definition 2.2.10 on page 38, under a SRT scheduler, a task τ_i does not miss each of its deadlines by more than its tardiness bound Δ_i.

In what follows, we will use the concept of *as late as possible (ALAP) completion schedule* in case of tardiness, which is defined below.

Definition 5.5.1. *(ALAP completion schedule in case of tardiness)*. The ALAP completion schedule considers that all invocations $A_{i,j}$ (jobs $\tau_{i,j}$) of an actor A_i (task τ_i) incur the maximum tardiness Δ_i, therefore complete at $z_{i,k} = d_{i,k} + \Delta_i$ (where $d_{i,k}$ represents the absolute deadline of job $\tau_{i,j}$, as defined in Section 2.2.1).

Then, consider that actor A_j has a data dependency from actor A_i through edge e_u. In addition, assume that both A_i and A_j may be affected by tardiness. In order to derive the earliest start time of actor A_j, Equation (2.30) on page 44 has to be modified in order to capture the worst-case schedule of A_i and A_j, as shown in the following proposition.

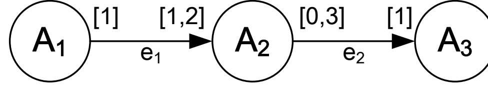
Proposition 5.5.1. *In presence of task tardiness, bounded by Δ_i for source actor A_i and by Δ_j for destination actor A_j, the earliest start time $S_{i\to j}$ of actor A_j due to its dependency from A_i through edge e_u, under a valid schedule, is given by:*

$$S_{i\to j} = \min_{t\in[0,S_i+\Delta_i+H]} \Big\{ t : \underset{[S_i+\Delta_i,\max(S_i+\Delta_i,t)+k)}{\text{prd}^{\text{S}}}(A_i, e_u) \geq \underset{[t,\max(S_i+\Delta_i,t)+k]}{\text{cns}^{\text{S}}}(A_j, e_u) \quad \forall k = 0, 1, \cdots, H \Big\} \quad (5.1)$$

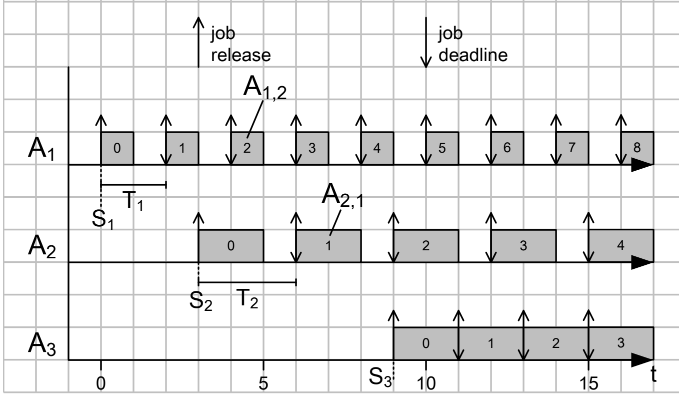
where H is the iteration defined by Equation (2.28).

Proof. If actors A_i and A_j may be affected by tardiness, Equation (2.30) on page 44 can not be applied in its original form to derive earliest actor start times. In order to illustrate this fact, we copy in Figure 5.4(a) the CSDF graph used as an example in Chapter 2. The corresponding hard real-time schedule, derived using the methodology of Section 2.3 in absence of tardiness, is shown in Figure 5.4(b). Note that in Figure 5.4(b) the start times of actors are calculate using Equation (2.29), which in turn exploits Equation (2.30).

Now, for instance, assume that actor A_1 may be affected by tardiness because it is scheduled by a SRT scheduler. Then, if invocation $A_{1,2}$ in Figure 5.4(b) completes later than its deadline, invocation $A_{2,1}$ of A_2 (that depends on the completion of $A_{1,2}$) cannot be released at time $t = 6$. It follows that the start time of actor A_2 has to be changed.



(a) Example of a CSDF graph (extracted from Section 2.1.2).



(b) Real-time periodic task set (extracted from Section 2.3)) obtained from the above CSDF graph.

Figure 5.4: *Example of the conversion of a set of CSDF actors to a real-time periodic task set using the methodology proposed in [BS11, BS12]. As explained in Section 2.3, the three actors of the CSDF graph depicted in sub-figure (a) can be scheduled as three real-time periodic tasks, as shown in sub-figure (b).*

The worst-case scenario of the execution of A_i and A_j to derive the earliest start time $S_{i \to j}$ in case of tardiness occurs when the source actor A_i completes its jobs *as late as possible*, i.e., according to its ALAP completion schedule (see Definition 5.5.1).

As shown in Figure 5.5, the ALAP completion schedule of actor A_i can be represented by a fictitious actor $\tilde{A}_i$, which has the same period as A_i, no tardiness, and start time $\tilde{S}_i = S_i + \Delta_i$. At run-time, any invocation of A_i, even if delayed by the maximum allowed tardiness Δ_i, will never complete later than the corresponding invocation of $\tilde{A}_i$. Notice that invocations $\tilde{A}_{i,k}$ of $\tilde{A}_i$ are strictly periodic, because they incur no tardiness.

By contrast, in the worst-case scenario to determine $S_{i \to j}$, A_j is executed as early as possible, so we assume that all invocations $A_{j,k}$ of A_j are not affected by tardiness. Then, the earliest start time that guarantees the absence of blocking of A_j in its execution, even for the worst-case production and consumption patterns of A_i and A_j, is found by evaluating Equation (2.30) with $\tilde{A}_i$ as source actor and A_j as destination actor. This scenario is captured by Equation (5.1). Note that any completion of an invocation $A_{i,k}$ of A_i earlier than its corresponding worst-case $\tilde{A}_{i,k}$ results in an earlier production of tokens, such that the inequality in Equation (5.1) still holds for all $k \in [0, 1, \cdots, H]$. Similarly, if any of the invocations of A_j is affected by tardiness, the token consumption is executed later and Equation (5.1) guarantees that enough

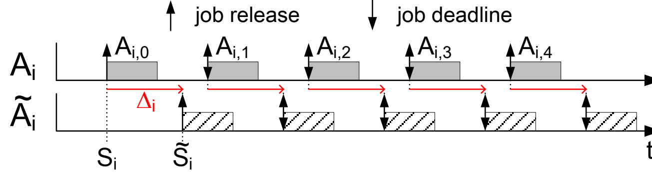


Figure 5.5: *Worst-case scheduling of source actor A_i, with tardiness Δ_i, when deriving the start time $S_{i\to j}$ of destination actor A_j. In the worst case, all invocations of actor A_i incur the maximum tardiness Δ_i. This schedule can be represented by a fictitious actor $\tilde{A}_i$, which has the same period as A_i, no tardiness, and start time $\tilde{S}_i = S_i + \Delta_i$.*

tokens will be available to be read. ■

Note also, from Equation (5.1), that the start time $S_{i\to j}$ of actor A_j due to its dependency from A_i is only affected by the tardiness bound Δ_i of the source actor. In addition, when actor A_j has several predecessors, the start time S_j has to be set to the maximum of the start times $S_{i\to j}$ given by Equation (5.1) considering each predecessor in isolation, as captured by Equation (2.29) on page 44.

5.5.2 Minimum Buffer Sizes in Presence of Tardiness

Similarly to Section 5.5.1, in order to derive minimum buffer sizes we also utilize the concept of tardiness bound under a SRT scheduler, defined in Definition 2.2.10 on page 38. In addition, we use the cumulative production and consumption functions of CSDF actors defined in Definitions 2.3.3 and 2.3.4 on page 46, namely:

- ${\rm prd^B}_{[t_s,t_f]}(A_i, e_u)$, which represents the total number of tokens produced by actor A_i to edge e_u during the time interval $[t_s, t_f]$;
- ${\rm cns^B}_{[t_s,t_f)}(A_j, e_u)$, which represents the total number of tokens consumed by actor A_j from edge e_u during the time interval $[t_s, t_f)$.

Then, similar to Section 5.5.1, consider that actor A_j has a data dependency from actor A_i through edge e_u, and both A_i and A_j may be affected by tardiness. Based on the above definitions, and given the actor start times calculated leveraging Proposition 5.5.1, the following proposition captures the worst-case scheduling of A_i and A_j when deriving the minimum buffer sizes in case of task tardiness.

Proposition 5.5.2. *In presence of task tardiness, bounded by Δ_i for source actor A_i and by Δ_j for destination actor A_j, the minimum buffer size b_u of a communication channel e_u connecting A_i and A_j, under a valid schedule, is given by:*

$$b_u(A_i, A_j) = \max_{k\in[0,1,\cdots,H]} \Big\{ \underset{[S_i,\max(S_i,S_j+\Delta_j)+k]}{\rm prd^B} (A_i, e_u) - \underset{[S_j+\Delta_j,\max(S_i,S_j+\Delta_j)+k)}{\rm cns^B} (A_j, e_u) \Big\} \quad (5.2)$$

Proof. To get the minimum buffer size in presence of task tardiness, we consider the worst-case scenario that would result in the maximum buffer requirement for channel e_u. This worst-case scenario occurs when: (i) all the invocations $A_{j,k}$ of the destination actor A_j complete with the maximum tardiness (i.e., A_j is executed according to its ALAP schedule, see Definition 5.5.1); (ii) none of the invocations of the source actor are affected by tardiness.

We can then prove Proposition 5.5.2 with a procedure similar to the one used in the proof of Proposition 5.5.1. We associate the worst-case completion of all the invocations $A_{j,k}$ to a fictitious actor $\tilde{A}_j$. Actor $\tilde{A}_j$ is strictly periodic, with no tardiness, constant period $\tilde{T}_j = T_j$ and start time $\tilde{S}_j = S_j + \Delta_j$. Then, the minimum buffer requirement of the communication channel e_u is found by evaluating Equation (2.31) on page 46 with A_i as source actor and $\tilde{A}_j$ as destination actor. This scenario is captured by Equation (5.2).

Note that any earlier completion of any of the iterations of A_j would not increase the buffer size requirement. This is because an earlier completion of A_j would results in an earlier consumption of tokens from channel e_u. Similarly, any delayed completion of an iteration of A_i would result in a delayed production of tokens to the considered channel. Thus, the derived value of b_u is sufficient. ■

Note that Equations (5.1) and (5.2) can also be used to analyze the interfaces between the external data provider and consumer (I and O in Figure 5.3(b)) and the input and output actors of the application (A_{in}, A_{out}). Compared to the HRT approach shown in Figure 5.3(a), in the SRT approach of Figure 5.3(b) A_{in} and/or A_{out} may experience tardiness. In this case, Equations (5.1) and (5.2) derive delayed start time of the external consumer O and larger buffer sizes of b_{in} and b_{out} such that both I and O can execute strictly periodically with neither buffer overflow nor underflow occurring on b_{in} and b_{out}.

5.6 FFD-SP Task Assignment Heuristic

The analysis provided in Section 5.5 extends the scheduling framework of [BS11, BS12] by calculating different task start times and buffer sizes, depending on the tardiness bounds of tasks. This way, the derived task set can be scheduled by any SRT scheduling algorithm. In this section, we present the second main contribution of this paper, which is focused on *a particular* SRT scheduling algorithm, namely EDF-fm [ABD08]. We recall that the most of the theoretical results regarding the EDF-fm algorithm are summarized in Section 2.2.7 of this thesis.

In our contribution, we propose a task assignment heuristic that does not follow the sequential approach common to all the heuristics proposed in [ABD08]. In fact, as explained in Section 2.2.7, the heuristics in [ABD08] assign tasks to processors in a sequential way, which means that in most cases processors have migrating tasks assigned to them. In turn, this makes most tasks in the system affected by tardiness. Actor tardiness imposes larger buffer sizes (according to Proposition 5.5.2) and postponed start times of successor actors (according to Proposition 5.5.1). Overall,

this leads to larger memory requirements and increased application latency. Our proposed heuristic is executed in Step ② of the design flow in Figure 5.2.

In contrast to the heuristics in [ABD08], our proposed heuristic, called FFD-SP, starts to consider semi-partitioning only when the *First-fit Decreasing* (FFD) heuristic [Joh73] (see Section 2.2.6) fails to assign a certain task in the system. The proposed task assignment heuristic accepts as input the number of processors M onto which the task set Γ has to be assigned. Then, the assignment of tasks to processors in FFD-SP proceeds as follows. At first, the set of stateful actors Γ_s is constructed and tried to be assigned to the processors using FFD. Stateful actors are considered first in our heuristics because this way they are fixed to a processor and at run-time there is no need to migrate their state.

Then, FFD-SP tries to assign the remaining (stateless) actors using FFD. Only when FFD fails, semi-partitioning is considered. This way, the number of processors with migrating tasks is likely to be less. Recall from Section 2.2.7 that, under EDF-fm, on processors which runs migrating tasks, fixed tasks are affected by tardiness. Therefore, by reducing the number of processors with migrating tasks, FFD-SP tries to reduce the number of fixed tasks with tardiness. When a task τ has to be semi-partitioned, its utilization is divided into two shares, $s_1(\tau)$ and $s_2(\tau)$. In particular, FFD-SP tries to assign the largest share possible $s_1(\tau)$ from the remaining available utilization on processors; then, it tries to find the *best fit* for the remaining share $s_2(\tau)$, in order to leave larger "chunks" of processor available utilizations to remaining (unallocated) tasks.

Our proposed FFD-SP assignment heuristic is reported in Algorithm 1. As mentioned earlier, at first Algorithm 1 builds Γ_s, the set of stateful actors, which are then assigned using the FFD heuristic (lines 1-4).

Then, considering task $\tau \in (\Gamma - \Gamma_s)$, the algorithm tries to assign task τ to one of the processors using FFD (lines 6-8). If FFD does not succeed, the algorithm tries to divide the utilization of task τ in two shares, $s_1(\tau)$ and $s_2(\tau)$. Traversing the processor list in decreasing order of available utilization, a share $s_1(\tau) = 1 - \sigma(\pi')$ is tried to be mapped on processor π' (lines 9-12). In line 10, the term $\sigma(\pi')$ denotes the total utilization assigned to π'. For the sake of clarity, this notation is slightly different from the one given in Equation (2.20) on page 39. If the assignment of $s_1(\tau)$ is successful, the algorithm attempts to map share $s_2(\tau) = u(\tau) - s_1(\tau)$ by traversing the list of processors in increasing order of available utilization (lines 13-17).

Note that our FFD-SP heuristic may fail to assign tasks to the considered set of processors. In fact, at the first execution of Algorithm 1, the number of processors M is set to M_{OPT}, the number of processors required by an optimal scheduler (see Equation (2.14) on page 34). If the task set cannot be assigned to M processors, M is increased by one and Algorithm 1 is executed again until a successful assignment is found.

The algorithm makes use of the *sp_assign* function to try and assign task shares. As shown in Algorithm 2, this function checks three conditions (see line 1):

1. There must be enough available utilization on the processor to accommodate the share. Similarly to Algorithm 1, the term $\sigma(\pi)$ denotes the total utilization assigned to π.

Algorithm 1: FFD-SP task assignment heuristic.

```
   Input: The number of processors M, a task set Γ = {τ1, τ2, ··· , τN} of N periodic tasks.
   Result: True and an M-partition describing the task assignment onto M processors if Γ is
           schedulable, False otherwise.
1  Find Γs = {τ : τ ∈ Γ ∧ τ is stateful};
2  Assign tasks in Γs using FFD;
3  if Γs cannot be assigned then
4      return False;
5  for τ ∈ (Γ − Γs, sorted in decreasing utilization) do
6      Try to assign task τ using First-Fit heuristic;
7      if First-Fit is successful then
8          continue;
9      for π' ∈ (Π sorted in decr. available utilization) do
10         s1(τ) = 1 − σ(π');
11         Assigned = False;
12         if sp_assign(s1(τ), π')==True then
13             s2(τ) = u(τ) − s1(τ);
14             for π'' ∈ (Π sorted in incr. available utilization) do
15                 if sp_assign(s2(τ), π'')==True then
16                     Assigned = True;
17                     break;
18             if Assigned==False then
19                 Revert assignment of s1(τ) to π';
20             else
21                 break;
22     if Assigned==False then
23         return False;
24 Optimize the obtained partition;
25 return True;
```

Algorithm 2: *sp_assign* function.

```
   Input: The share s of task τ to be assigned, a processor π.
   Result: True if s can be assigned to π, False otherwise.
1  if (σ(π) + s ≤ 1) and (σ^mig_current(π) + u(τ) ≤ 1) and (n_mig_tasks < 2) then
2      Assign s to π;
3      return True;
4  else
5      return False;
```

2. In case another migrating task has already been mapped on processor π, Condition (2.21) in Section 2.2.7 must be satisfied. Note that in Algorithm 2 the term $\sigma^{\text{mig}}_{\text{current}}(\pi)$ denotes the total utilization of migrating tasks assigned to π.
3. The number of migrating tasks $n_{\text{mig_tasks}}$ assigned to processor π must be less than 2 (as required by the EDF-fm algorithm).

When an M-partition (see Definition 2.2.6 on page 35) has been successfully found,

the FFD-SP heuristic tries to *optimize* it (line 24 in Algorithm 1). The optimization consists in re-assigning the migrating task shares, whenever possible, to processors to which less fixed tasks are assigned. This way, less fixed tasks are affected by tardiness, leading to lower application latency and buffer size requirements. Note that in Algorithm 1 the first share of a migrating task $s_1(\tau)$ is set to the largest possible value, given the current available utilization of processors. This in turns makes the second share of each migrating task $s_2(\tau)$ as small as possible, making the process of optimization of the partition more effective.

5.7 Evaluation

We evaluate our semi-partitioned scheduling approach using the StreamIt benchmarks considered in [ZBS13], for which we employ the unfolding technique described in [ZBS13] to derive larger CSDF graphs with improved throughput. Among these benchmarks, seven applications require, under the partitioned FFD allocation scheme, more processors than an optimal scheduler. This set of applications is listed in Table 5.1. In this section we compare the number of required processors, memory requirements, and application latencies obtained with three different allocation/scheduling approaches: (i) Partitioned EDF with FFD heuristic; (ii) Semi-partitioned EDF-fm, with our proposed FFD-SP heuristic; (iii) Semi-partitioned EDF-fm, with the LUF (Lowest Utilization First) heuristic proposed in [ABD08]. These approaches are denoted in Table 5.1 with *FFD*, *FFD-SP*, and *LUF*, respectively.

Note that *all* the approaches in Table 5.1 lead to the same application throughput. This is because the throughput of an application depends on the period of its sink actor, which is unchanged in our analysis even in presence of task tardiness. In addition, we choose to compare the results of the LUF heuristic with our FFD-SP heuristic because, among the heuristics proposed in [ABD08], LUF achieves the smallest number of processors.

The M_{OPT} column in Table 5.1 lists the number of processors required by an optimal scheduler (for instance [BCPV96]) to execute the considered applications. M_{OPT} is obtained using Equation (2.14).

Let us focus on the comparison between the partitioned approach (FFD, described in Section 2.2.6) and our proposed semi-partitioned approach (FFD-SP, proposed in Section 5.6). We note that the FFD approach results in a number of processors (M_{FFD}) which is on average 17.6% greater than the number required by an optimal scheduler (see column $\frac{M_{\mathrm{FFD}}}{M_{\mathrm{OPT}}}$). In contrast, our FFD-SP algorithm requires on average only 2.1% more processors (see column $\frac{M_{\mathrm{SP}}}{M_{\mathrm{OPT}}}$), while maintaining the same throughput. This means that our proposed approach can exploit the available processors more efficiently, getting significantly closer to the results obtained by optimal schedulers (see columns M_{SP} and M_{OPT}). However, this comes at two costs.

The **first cost** is the increase of memory requirements. For each benchmark, column mem_{FFD} reports the memory required by the partitioned approach, expressed in bytes.

Table 5.1: *Comparison of different allocation/scheduling approaches.*

Benchmark	**OPT**	**Partitioned (FFD)**				**Semi-partitioned (FFD-SP)**				**Semi-partitioned (LUF)**		
	M_{OPT}	M_{FFD}	$\frac{M_{FFD}}{M_{OPT}}$	mem_{FFD}[B]	L_{FFD}[c.c.]	M_{SP}	$\frac{M_{SP}}{M_{OPT}}$	$\frac{mem_{SP}}{mem_{FFD}}$	$\frac{L_{SP}}{L_{FFD}}$	M_{LUF}	$\frac{mem_{LUF}}{mem_{FFD}}$	$\frac{L_{LUF}}{L_{FFD}}$
FFT	24	30	1.25	144680	192512	26	1.083	1.413	1.483	26	1.485	1.676
Beamformer	26	28	1.077	14492	60912	26	1.0	1.145	1.474	26	1.229	1.606
TDE	20	25	1.25	516282	1127175	20	1.0	1.560	1.396	21	1.722	1.860
DES	26	33	1.269	3381	33088	27	1.038	1.138	1.218	28	1.684	1.862
MPEG2	8	9	1.125	61909	138240	8	1.0	1.290	1.217	9	3.014	3.432
Bitonic	11	13	1.182	2374	2275	11	1.0	1.139	1.185	11	1.413	1.395
Serpent	39	42	1.077	59815	370296	40	1.026	1.012	1.074	39	1.068	1.479
average	-	-	1.176	-	-	-	1.021	1.243	1.292	-	1.659	1.902

It is derived using the following expression:

$$\text{mem}_{\text{FFD}} = \sum_{i=1}^{N} \text{CSS}(\tau_i) + \sum_{i=1}^{|E|} b_u^{HRT} \tag{5.3}$$

where N is the number of tasks, $\text{CSS}(\tau_i)$ 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. Compared to FFD, in FFD-SP the memory requirements increase due to both the size of buffers, that have to be enlarged to handle task tardiness, and the code and stack size overhead of task replicas, which are necessary in case of migrating tasks. The memory requirement in FFD-SP is denoted by mem_{SP} and calculated as follows.

$$\text{mem}_{\text{SP}} = \sum_{i=1}^{M_{\text{SP}}} \sum_{\tau_j \in \Gamma_i} \text{CSS}(\tau_j) + \sum_{i=1}^{|E|} b_u^{SRT} \tag{5.4}$$

where M_{SP} is the number of processors required by the semi-partitioned approach, 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 (5.4) differs from Equation (5.3) because in the SP approach a task can have shares on different processors.

In Table 5.1 the overhead of our proposed FFD-SP over FFD, in terms of memory requirements, is expressed by the ratio $\frac{\text{mem}_{\text{SP}}}{\text{mem}_{\text{FFD}}}$. On average, our proposed approach requires 24.3% more memory compared to FFD.

The **second cost** is the increase in applications' latency, due to the postponement of task start times needed to handle task tardiness. Column L_{FFD} shows the applications' latency, expressed in clock cycles, under FFD. These values are derived using the latency analysis described in Section 4.7 of [Bam14]. In order to derive the application latency under FFD-SP, denoted by L_{SP}, we use the same analysis from [Bam14], considering the task start times obtained by our SRT approach (described in Section 5.5.1). Then, we add to that latency value the tardiness (which can be potentially null) of the output actor of the application. The latency increase of our FFD-SP over FFD is on average 29.2% (see column $\frac{L_{\text{SP}}}{L_{\text{FFD}}}$).

Finally, to evaluate the efficiency of our proposed FFD-SP heuristic, we compare its results with LUF. The memory requirements and application latencies under LUF are derived following the same procedures used for FFD-SP. We can see from the last two columns of Table 5.1 that over the considered benchmarks the EDF-fm approach with the LUF heuristic incurs a much larger memory overhead (on average +65.9%, see column $\frac{\text{mem}_{\text{LUF}}}{\text{mem}_{\text{FFD}}}$) and latency increase (on average +90.2%, see column $\frac{L_{\text{LUF}}}{L_{\text{FFD}}}$) compared to FFD. Moreover, we note that for most applications the number of required processors is equal or greater when using LUF (M_{LUF}) compared to our FFD-SP (M_{SP}), with the exception of the *Serpent* application. Only in that example, the LUF outperforms our FFD-SP due to the characteristics of the task set. This means

that for most of the benchmarks our FFD-SP heuristic is equally or more efficient than LUF in exploiting the available processing resources.

5.8 Discussion

The theoretical analysis provided in Section 5.5 proves that streaming applications modeled as acyclic CSDF graphs can be scheduled using any soft real-time scheduler, providing hard real-time guarantees on the input/output interfaces between the application and the environment.

Using the theoretical results of Section 5.5, in Section 5.6 we propose a novel heuristic that is aimed at reducing the number of required processors while keeping a low buffer size and latency overhead when the EDF-fm SRT scheduling algorithm is used. Section 5.7 shows that on a set of real-life applications, our approach can reduce the number of processors required to schedule these applications, guaranteeing the same throughput. However, compared to a HRT partitioned approach, our semi-partitioned SRT approach incurs an overhead in terms of memory requirements (on average, 24.3%) and application latency (on average, 29.2%).