Semi-partitioned scheduling and task migration in dataflow networks

Cannella, E.

Chapter 2

Background

IN this chapter, we introduce the mathematical notations, definitions, concepts, and existing theoretical results that are necessary to understand the contributions of this thesis.

We first provide in Table 2.1 a summary of the mathematical notations used in this thesis.

Symbol	Meaning
$\mathbb{N}$	The set of natural numbers excluding zero
$\mathbb{N}_0$	$\mathbb{N} \cup \{0\}$
$\mathbb{Z}$	The set of integers
$\lvert x \rvert$	The cardinality of a set x
$\hat{x}$	The maximum value of x
mod	The integer modulo operator
${}^{x}V$	An x-partition of a set V (see Definition 2.2.6)

Table 2.1: *Summary of mathematical notation.*

Then, in Section 2.1, we describe in further detail the MoCs used in this dissertation. A preliminary introduction of these MoCs was given earlier in Section 1.1.1. In addition, in Section 2.2 we present results and definitions from hard real-time scheduling theory that are instrumental to understand our contributions in the context of HRT systems (Chapters 5 and 6). Finally, in Section 2.3 we describe the methodology for hard real-time scheduling of CSDF graphs proposed by [BS11, BS12]. In fact, our contributions presented in Chapters 5 and 6 represent an extension of the framework proposed in [BS11, BS12], therefore a thorough introduction to that framework is necessary.

2.1 Dataflow Models of Computation

As mentioned in Section 1.1.1, dataflow MoCs represent a good match for streaming applications because they allow to express the parallelism available in these kind of applications in a natural way. In this thesis we consider the MoCs described in Sections 2.1.1-2.1.3, namely (H)SDF, CSDF, and PPN. In both (C | H)SDF[1] and PPN MoCs applications are specified in the form of directed graphs in which graph nodes represent the tasks (active entities) of the application and graph edges represent inter-task data dependencies. In (C | H)SDF, nodes of the application graph are called *actors*, whereas nodes in a PPN are called *processes*. The actor-based MoCs considered in this thesis, (H)SDF and CSDF, are presented in Sections 2.1.1 and 2.1.2, respectively. These MoCs are used to specify the input applications in the techniques for HRT systems proposed in Chapters 5 and 6. The process-based MoC considered in this thesis, PPN, is described in Section 2.1.3 and is used to specify applications in the approaches proposed for best-effort (BE) systems, which are presented in Chapters 3 and 4.

2.1.1 (Homogeneous) Synchronous Dataflow ((H)SDF)

The **Synchronous Dataflow (SDF)** MoC is introduced in [LM87b]. Under this MoC, an application is modeled as a directed multigraph $G = (A, E)$ where A is the set of actors and E is the set of edges. Actors represent tasks of the application and edges represent inter-task data dependencies. Actors communicate over edges generating a stream of data, which is divided in atomic data objects called *tokens*. An edge $e_u \in E$ represents a first-in first-out (FIFO) buffer and is defined by a tuple $e_u = (A_i, A_j)$. This tuple means that the edge is directed from actor A_i (called *source*) to actor A_j (called *destination*). We define input and output actors of graph G as follows.

Definition 2.1.1. *(Input actor).* An input actor of graph G is an actor that receives the input stream of the application from the environment.

Definition 2.1.2. *(Output actor).* An input actor of graph G is an actor that produces the output stream of the application to the environment.

An execution of an actor $A_i \in A$ is called a **firing** or **invocation**. In this thesis we denote the jth invocation (with $j \in \mathbb{N}_0$) of actor A_i as $A_{i,j}$. Invocation $A_{i,j}$ can begin its execution only if enough input data is present on all its input edges. During one invocation, actor A_i consumes input data from all its input edges, processes this data according to a function f_i, and writes the output data to its output edges. The amount of data read/written from/to each input/output edge is fixed, known at compile-time and it is called consumption/production *rate*. For each $A_i \in A$ we can define a set of *predecessor* and *successor* actors, denoted by $\mathsf{prec}(A_i)$ and $\mathsf{succ}(A_i)$, respectively. These sets are defined as follows.

$$\mathsf{prec}(A_i) = \{A_j \in A : \exists e_u = (A_j, A_i)\} \quad (2.1)$$

$$\mathsf{succ}(A_i) = \{A_j \in A : \exists e_u = (A_i, A_j)\} \quad (2.2)$$

[1] For the sake of brevity, we identify CSDF, SDF and HSDF MoCs with the acronym (C | H)SDF.

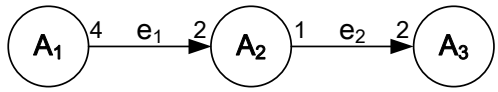

Figure 2.1: *Example of an SDF graph composed of actors A_1, A_2, A_3 and edges e_1, e_2. Numbers over the edges indicate the production/consumption rates of source/destination actors of that edge. For instance, each invocation of actor A_2 consumes 2 tokens from e_1 and produces 1 token to e_2.*

We assume that any input actor A_{in} has no predecessors and any output actor A_{out} has no successors, i.e., $\mathsf{prec}(A_{\text{in}}) = \emptyset$ and $\mathsf{succ}(A_{\text{out}}) = \emptyset$. Moreover, we can define for each $A_i \in A$ a set of *input* and *output* edges, denoted by $\text{inp}(A_i)$ and $\text{out}(A_i)$, respectively.

An example of an SDF graph is shown in Figure 2.1. The numbers over the edges indicate the production/consumption rates of source/destination actors of that edge. Consider edge $e_u = (A_i, A_j)$. The production rate of source actor A_i over edge e_u is denoted by x_i^u. Conversely, the consumption rate of destination actor A_j over edge e_u is denoted by y_j^u. For instance, each invocation of actor A_1 in Figure 2.1 produces 4 tokens to e_1, and each invocation of actor A_2 consumes 2 tokens from the same edge. Therefore, $x_1^1 = 4$ and $y_2^1 = 2$.

A special case of the SDF MoC is the **Homogeneous SDF (HSDF)**, that is an SDF in which all the production/consumption rates of all the actors are equal to one. An example of an HSDF graph is given in Figure 1.2(a) on page 5, where production/consumption rates of actors are omitted because they are all equal to one.

Since streaming applications process continuous streams of data, we are interested in determining a schedule of the SDF graph that can continue indefinitely, using a finite amount of memory to implement the FIFO channels corresponding to the edges of the graph. Such a schedule can be derived at compile-time if the SDF graph is **consistent** and **deadlock-free**.

An SDF graph G is consistent [LM87a] if its *balance equation*, given below, has a positive integer solution.

$$\mathbf{\Gamma_G} \cdot \vec{q} = \vec{0} \tag{2.3}$$

In the expression above, $\mathbf{\Gamma_G} \in \mathbb{Z}^{|E| \times |A|}$ is called **topology matrix** and $\vec{q}$ is called **repetition vector**. The topology matrix is constructed as follows:

$$\Gamma_{uj} = \begin{cases} x_j^u & \text{if actor } A_j \text{ produces on edge } e_u \\ -y_j^u & \text{if actor } A_j \text{ consumes from edge } e_u \\ 0 & \text{otherwise.} \end{cases} \tag{2.4}$$

In particular, the repetition vector with the smallest norm is called **basic repetition vector**. In this thesis, unless otherwise specified, we will utilize the basic repetition vector of a graph to perform our analyses. The meaning of the repetition vector is the following. If every actor A_i of the graph is fired q_i times, where q_i is the ith component of the repetition vector, then the net change of the number of tokens in the FIFO channels is zero.

For the example in Figure 2.1, the topology matrix $\mathbf{\Gamma_G}$ is given below.

$$\mathbf{\Gamma_G} = \begin{bmatrix} 4 & -2 & 0 \\ 0 & 1 & -2 \end{bmatrix}$$

Its (basic) repetition vector $\vec{q_G}$, derived using Equation (2.3), is:

$$\begin{aligned} \vec{q_G} &= [q_{A_1}, q_{A_2}, q_{A_3}]^T \\ &= [1, 2, 1]^T \end{aligned}$$

Note that any vector $\vec{q}'$ obtained by multiplying the basic repetition vector $\vec{q_G}$ by a positive natural number is also a repetition vector of G, i.e., it satisfies Equation (2.3). Note also that the existence of a positive integer solution to Equation (2.3) is only a necessary condition to execute an SDF graph indefinitely with a periodic schedule. Another condition that must be satisfied is the absence of deadlocks, which can be verified by constructing a periodic admissible schedule [LM87a] of the graph. An SDF graph that has no deadlock is called *deadlock-free*, or *live*. An important property of the SDF MoC is that the consistency and liveness of an SDF graph can be verified at compile-time. In this thesis we consider only consistent and live SDF graphs.

2.1.2 Cyclo-Static Dataflow (CSDF)

The Cyclo-static Dataflow (CSDF) MoC [BELP96] is a generalization of SDF. Similar to SDF, a CSDF graph $G = (A, E)$ also consists of a set of actors A and a set of edges E. However, contrary to SDF, the behavior of CSDF actors is cyclic, as explained in the following.

Each CSDF actor A_i has a certain number of *phases*, denoted by Ω_i. The execution of each phase φ is associated with a certain function $f_i(\varphi)$. Therefore, we can define an **execution sequence** $[f_i(0), f_i(1), \cdots, f_i(\Omega_i - 1)]$ which links each phase to the corresponding executed function. Moreover, production/consumption rates for each output/input edge are also defined for each phase. Thus, for each actor A_i, the following sequences can be defined.

- **Consumption sequence** for each input edge e_u:

$$[y_i^u(0), y_i^u(1), \cdots, y_i^u(\Omega_i - 1)]$$

- **Production sequence** for each output edge e_u:

$$[x_i^u(0), x_i^u(1), \cdots, x_i^u(\Omega_i - 1)]$$

Notice that the length of all these sequences is Ω_i, the number of phases.

Phases of a CSDF actor A_i are executed in a cyclic fashion. That is, during invocation $A_{i,n}$ (with $n \in \mathbb{N}$) of actor A_i, function $f_i((n) \bmod \Omega_i)$ is executed. Similarly, for each input edge e_u, $y_i^u((n) \bmod \Omega_i)$ tokens are consumed and for each output edge e_u,

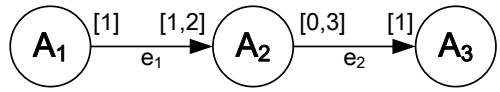

Figure 2.2: *Example of a CSDF graph.*

$x_i^u((n) \bmod \Omega_i)$ tokens are produced. The cumulative number of tokens consumed by invocations $A_{i,0}$ to $A_{i,n}$ of actor A_i from its input edge e_u is denoted by:

$$Y_i^u(n) = \sum_{l=0}^{n} y_i^u(l) \tag{2.5}$$

Similarly, the cumulative number of tokens produced by invocations $A_{i,0}$ to $A_{i,n}$ of actor A_i to its output edge e_u is denoted by:

$$X_i^u(n) = \sum_{l=0}^{n} x_i^u(l) \tag{2.6}$$

Similar to the SDF case, we are interested in finding an indefinite, periodic schedule of a CSDF graph G. As shown in [BELP96], a repetition vector $\vec{q}$ of G is given by:

$$\vec{q} = \mathbf{\Theta} \cdot \vec{r}, \qquad \text{with} \qquad \Theta_{ik} = \begin{cases} \Omega_i & \text{if } i = k \\ 0 & \text{otherwise} \end{cases} \tag{2.7}$$

where $\vec{r} = [r_1, r_2, \cdots, r_{|A|}]^T$ is a positive integer solution of the balance equation

$$\mathbf{\Gamma} \cdot \vec{r} = \vec{0} \tag{2.8}$$

and where the *topology matrix* $\mathbf{\Gamma} \in \mathbb{Z}^{|E| \times |A|}$ is defined by

$$\Gamma_{uj} = \begin{cases} X_i^u(\Omega_j - 1) & \text{if actor } A_i \text{ produces on edge } e_u \\ -Y_i^u(\Omega_j - 1) & \text{if actor } A_i \text{ consumes from edge } e_u \\ 0 & \text{otherwise.} \end{cases} \tag{2.9}$$

Example 2.1.1. An example of a CSDF graph is shown in Figure 2.2. The graph indicates the production/consumption sequences of the actors over the edges of the graph. For instance, actor A_2 has consumption sequence $[1, 2]$ over e_1 and production sequence $[0, 3]$ over e_2. From Equations (2.7)-(2.9), we derive the repetition vector $\vec{q}$ as shown below.

$$\mathbf{\Gamma} = \begin{bmatrix} 1 & -3 & 0 \\ 0 & 3 & -1 \end{bmatrix}, \vec{r} = \begin{bmatrix} 3 \\ 1 \\ 3 \end{bmatrix}, \mathbf{\Theta} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \text{and } \vec{q} = \begin{bmatrix} 3 \\ 2 \\ 3 \end{bmatrix}$$

Based on this repetition vector $\vec{q}$, we can derive a static non-preemptive schedule for the CSDF graph in Figure 2.2 that can be repeated forever using bounded buffers. The following schedule possess this property.

$$\text{Schedule 1:} \quad A_1 A_1 A_1 \, A_2 A_2 \, A_3 A_3 A_3 = 3A_1 \, 2A_2 \, 3A_3 \tag{2.10}$$

Note that alternative schedule exists.

For a consistent and live (H)SDF or CSDF graph $G = (A, E)$, given the repetition vector $\vec{q}$ of the graph, we can define the concept of actor iteration and graph iteration as shown below.

Definition 2.1.3. *(Actor iteration).* An **actor iteration** is the invocation of an actor A_i for q_i times.

Definition 2.1.4. *(Graph iteration).* A **graph iteration** is the invocation of every actor A_i for q_i times.

Stateless Actors

In this thesis, we will use the concept of *stateless* and *stateful* actors. This concept is common to both the (H)SDF and CSDF MoCs. Formally, any (C | H)SDF actor is stateless because the relation between tokens consumed and produced during an invocation is defined by a function, as mentioned earlier. Needless to say, a function does not have a state. However, sometimes it is necessary to model actors for which the result of the current invocation is dependent from the data produced in the previous invocations. In the (C | H)SDF MoC, these dependencies from previous invocations are modeled using self-edges. On these self-edges, an invocation can write data tokens that represent the actor "state" and that can be read by successive invocations. A formal definition of stateless actors is given below.

Definition 2.1.5. *(Stateless actor).* A (C | H)SDF actor is called *stateless* if it has no self-edges used to model its state.

Consequently, stateful actors are defined as follows.

Definition 2.1.6. *(Stateful actor).* A (C | H)SDF actor is called *stateful* if it has self-edges used to model its state.

2.1.3 Polyhedral Process Network (PPN)

The Polyhedral Process Network (PPN) [VNS07] MoC is used in this thesis mainly in the context of best-effort systems (Chapters 3 and 4). The PPN MoC is a special case of the Kahn Process Network (KPN) MoC proposed in [Kah74]. A PPN is a directed multigraph G defined as a tuple $G = (\mathcal{P}, \mathcal{C})$, where:

- $\mathcal{P} = \{P_1, P_2, \cdots, P_{|\mathcal{P}|}\}$ is a set of processes;
- $\mathcal{C} = \{\text{ch}_1, \text{ch}_2, \cdots, \text{ch}_{|\mathcal{C}|}\}$ is a set of FIFO channels.

Processes in $\mathcal{P}$ represent tasks of an application and they communicate among each other using FIFO channels in $\mathcal{C}$. An example of PPN is depicted in Figure 2.3(a). Each PPN process has a set of *input ports* (for instance, process P_2 has input ports IP_1 and IP_2) and a set of *output ports* (P_2 has only one output port, OP_1), through which the process reads and writes data. Channels connected to the input and output ports of a process P are called *input* and *output channels*, and denoted by IC_P and OC_P, respectively.

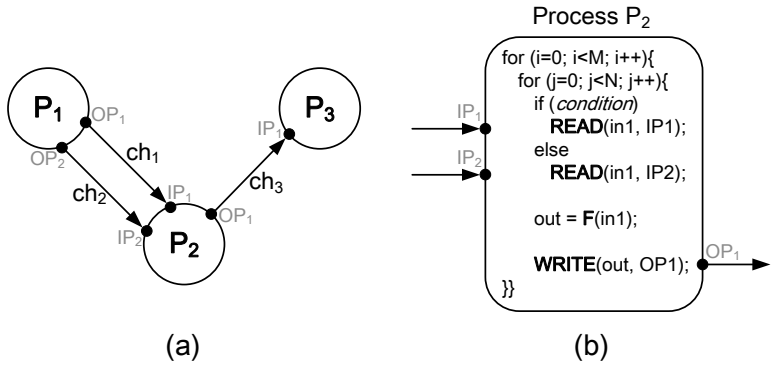

Figure 2.3: *In sub-figure (a), an example of a PPN topology composed of processes P_1, P_2, P_3 and FIFO channels ch_1, ch_2, ch_3. Processes read/write data tokens from/to channels using input/output ports, which are denoted by dots. Sub-figure (b) shows the internal structure of process P_2 of sub-figure (a). As in all PPN processes, the structure of P_2 is based on nested for-loops.*

Similar to KPN processes, PPN processes are synchronized through the FIFO channels, that is, processes that attempt to read from an empty FIFO will block (*blocking read*). However, contrary to KPNs, in PPNs FIFO buffers have bounded size, therefore processes are also forced to block when trying to write to a full FIFO (*blocking write*).

Note that in PPNs control and synchronization are completely distributed, which allows to change the mapping of processes to PEs at run-time with minor effort. We leverage this advantage in our proposed techniques aimed at achieving system adaptivity in NoC based MPSoCs, see Chapters 3 and 4.

Another restriction with respect to the KPN MoC is that in PPNs processes have a precise structure. As shown in Figure 2.3(b) for process P_2, the execution of a PPN process is defined using nested *for*-loops. Each execution of a PPN process corresponds to a certain value of the *for*-loop iterators. The value of these iterators can be represented as a vector $\vec{I}$, called *iteration vector*. For P_2, the iteration vector is $\vec{I} = [i, j]$.

Each PPN process executes as follows. First, the process reads data from (a subset of) its input ports. The subset of input ports from which data is read depends on the value of the iteration vector. For instance, process P_2 reads data from IP_1 if the *condition*[2] in Figure 2.3(b) is satisfied, otherwise data is read from IP_2. Then, the input data is processed by a function (in P_2, this is represented by the line $out = F(in1)$). This function represents the computational behavior of the process. Finally, the process writes the produced data to (a subset of) its output ports. The subset of output ports to which data is written depends, again, on the value of the iteration vector.

[2]Conditions used to determine whether an input/output port has to be read/written can contain any affine relation of the loop iterators, static parameters, and constants.

Note that, similar to the actor-based MoCs presented in Sections 2.1.1 and 2.1.3, the relation between input and output data of a process is defined by a function, which is by definition stateless. In order to model processes for which the result of the current iteration is dependent from the data produced in the previous iterations, one can use self-channels. On these self-channels, an iteration can write data tokens that represent the process "state" and that can be read by successive iterations. This state is therefore stored outside the process itself. However, note that the set of input/output ports which is read/written by the PPN process is derived based on its iteration vector $\vec{I}$. Therefore, the iteration vector is in fact the only state of the PPN process.

Automatic derivation from SANLPs

The restrictions imposed by the PPN MoC compared to the KPN MoC lead to the following important property: any sequential application specified as a Static Affine Nested Loop Program (SANLP) can be automatically converted to an equivalent parallel PPN specification [VNS07]. An SANLP can be defined as follows (from [Mei10]).

Definition 2.1.7. *(Static Affine Nested Loop Program (SANLP)).* An SANLP is a program where each program statement is enclosed by one or more loops and if-statement, and where:

- loops have a constant step size;
- loops have bounds that are affine expressions of the enclosing loop iterators, static program parameters, and constants;
- if-statements have affine conditions in terms of the loop iterators, static program parameters, and constants;
- index expression of array references are affine expressions of the enclosing loop iterators, static program parameters, and constants;
- data flow between statements in the loop is explicit, which prohibits that two statements that contain function calls communicate through shared variables invisible at the SANLP level.

In particular, in this thesis we use the `pn` compiler [VNS07] to automatically convert static affine nested loop programs (SANLPs) to parallel PPN specifications and to determine the buffer sizes that guarantee deadlock-free execution. Although the `pn` compiler imposes some restrictions on the specification of the input application, a large set of streaming applications can be effectively specified as SANLPs. In addition to the case studies considered in Chapters 3 and 4, SANLPs can model applications from various domains, such as image/video processing (JPEG2000, H.264), sound processing (FM radio, MP3), and scientific computation (QR decomposition, stencil, finite-difference time-domain). Moreover, a recent work [TA10] has shown that most of the streaming applications can be specified using the Synchronous Data Flow (SDF) model [LM87b]. The PPN model is more expressive than SDF, thus it can as well be used effectively to model most streaming applications.

2.2 Real-time Scheduling Theory

In this section, we introduce in a formal way the real-time periodic task model and important schedulability results for multiprocessor systems. This task model and analysis techniques are instrumental to the approaches we present in Chapters 5 and 6. Finally, we describe the notation and theoretical results of two semi-partitioned scheduling algorithms which are leveraged in our work: EDF-fm [ABD08] and EDF-os [AEDC14].

2.2.1 Real-time periodic and sporadic task models

Under the real-time **periodic** task model, a **task** is defined by a 4-tuple $\tau_i = (C_i, T_i, S_i, D_i)$, where C_i is the worst-case execution time (WCET) of the task, T_i is the task period, S_i is the start time of the task, and D_i is the deadline of the task. A periodic task τ_i starts at time S_i and is recurrent, with a constant inter-arrival time T_i. That is, a periodic task τ_i is invoked at time instants $r_{i,k} = S_i + kT_i$, for all $k \in \mathbb{N}_0$. Each invocation of τ_i is called a **job**. The kth job of τ_i is denoted by $\tau_{i,k}$. Job $\tau_{i,k}$ must complete its execution before time $d_{i,k} = r_{i,k} + D_i$. In this thesis, we assume that tasks have *implicit deadlines*, i.e., $D_i = T_i$ for each task τ_i. In this case, the absolute deadline of job $\tau_{i,k}$ is $d_{i,k} = S_i + (k+1)T_i$ is coincident with the arrival of job $\tau_{i,k+1}$. We denote the actual completion time of $\tau_{i,k}$ as $z_{i,k}$. We assume that tasks can be preempted at any time. The *demand* of a real-time periodic task is defined as follows.

Definition 2.2.1. *(Demand of a periodic real-time task).* The demand of a periodic real-time task τ_i in the interval $[t_0, t_c)$ is the total time in which jobs of τ_i are executed in $[t_0, t_c)$. This demand is denoted by $\mathrm{dmd}(\tau_i, t_0, t_c)$.

In Section 2.3, we summarize the scheduling technique [BS11, BS12] on which our proposed approaches aimed at HRT systems are based. That scheduling technique considers an input application modeled as an acyclic (C)SDF graph with N actors. Then, this (C)SDF model of the application is converted to a set $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_N\}$ of N real-time periodic tasks. In general, tasks in Γ do not have the same start time, i.e., Γ is an *asynchronous* task set. The **utilization** of task $\tau_i \in \Gamma$ is $u_i = C_i/T_i$ and the total utilization of the task set Γ is $U_\Gamma = \sum_{\tau_i \in \Gamma} u_i$.

The **sporadic** task model is a generalization of the periodic task model. Jobs released by a sporadic task must be separated in time by a minimum inter-arrival interval T_i.

2.2.2 System model

In this thesis, we consider *homogeneous* multiprocessor systems. That is, in the considered systems all the processors are identical and the speed of execution of tasks on processors is the same. In particular, we consider a system composed of a set $\Pi = \{\pi_1, \pi_2, \cdots, \pi_M\}$ of M homogeneous processors.

2.2.3 Multiprocessor Real-Time Scheduling Algorithms

In this section we describe some concepts and results from real-time scheduling analysis which are instrumental to the approaches proposed in this thesis. We focus on scheduling algorithms which handle periodic real-time task sets.

Multiprocessor scheduling algorithms try to solve two problems [DB11]:

1. The *allocation problem*, namely on which processor(s) jobs of a task should execute;
2. The *priority problem*, or when, and in what order with respect to jobs of other tasks, each job should execute.

Based on the way in which scheduling algorithms approach the **allocation problem**, they can be categorized in:

- *No migration.* Each task is allocated to only one processor, and no migration is allowed.
- *Task-level migration.* Different jobs of the same task can execute on different processors. However, each job can only be executed on one processor. These approaches are said to have **restricted migrations**.
- *Job-level migration.* A single job can migrate and be executed on different processors. However, parallel execution of a job is not allowed, i.e., the same job cannot be executed in parallel on two or more processors.

Algorithms that allow any task to migrate, either at task-level or at job-level, are termed **global**. By contrast, the algorithms that do not allow migration at any level are called **partitioned**.

Depending on how scheduling algorithms solve the **priority problem**, they can be classified in:

- *Fixed task priority.* Each task has a single fixed priority applied to all of its jobs.
- *Fixed job priority.* The jobs of a task may have different priorities, but each job has a single static priority. An example of this class is the Earliest Deadline First (EDF) [LL73] scheduling described in Section 2.2.4.
- *Dynamic job priority.* A single job may have different priorities during its execution. An example of this class is Least Laxity First (LLF) scheduling.

We proceed our discussion by introducing some useful definitions from [DB11].

Definition 2.2.2. *(Feasibility of a task set).* A task set is said to be *feasible* with respect to a given system if there exist some scheduling algorithm that can schedule all possible sequences of jobs that may be generated by the task set on that system without missing any deadline.

Definition 2.2.3. *(Optimal scheduling algorithm).* A scheduling algorithm is said to be *optimal* with respect to a system and a task model if it can schedule all of the task sets that comply with the task model and are feasible on the system.

Definition 2.2.4. *(Schedulability of a task and of a task set).* A task τ is referred to as *schedulable* according to a given scheduling algorithm $\mathcal{A}$ if its worst-case response time under $\mathcal{A}$ is less than or equal to its deadline. Similarly, a task set is referred to as *schedulable* under a given scheduling algorithm if all of its tasks are schedulable.

Real-time scheduling theory provides analytical *schedulability tests* to verify the schedulability of a task set Γ under scheduling algorithms $\mathcal{A}$. A schedulability test is termed *sufficient* if all task sets that are deemed schedulable according to the test are in fact schedulable [DB11]. Similarly, a schedulability test is termed *necessary* if all the task sets that are deemed unschedulable according to the test are in fact unschedulable. Finally, a schedulability test that is both sufficient and necessary is called *exact*.

For implicit deadline periodic task sets, an useful performance metric of both uniprocessor and multiprocessor scheduling algorithms is the worst-case utilization bound, as defined below.

Definition 2.2.5. *(Worst-case utilization bound (from [DB11]))*. The worst-case utilization bound $U_{\mathcal{A}}$ for a scheduling algorithm $\mathcal{A}$ is the minimum utilization of any implicit deadline task set that is only *just* schedulable under $\mathcal{A}$.

From this definition, it follows that every implicit deadline task set Γ with total utilization $U_\Gamma \leq U_{\mathcal{A}}$ is schedulable under $\mathcal{A}$. Therefore, the condition:

$$U_\Gamma \leq U_{\mathcal{A}} \tag{2.11}$$

can be used as a sufficient (not necessary) schedulability test for task set Γ under scheduling algorithm $\mathcal{A}$.

2.2.4 Uniprocessor Schedulability Analysis

Arguably, the two most popular scheduling algorithms for uniprocessor systems are Earliest Deadline First (EDF) and Rate Monotonic (RM). These two scheduling algorithms are described below.

Earliest Deadline First (EDF)

The EDF scheduling algorithm was proposed in the seminal paper [LL73] of Liu and Layland. Under EDF a task is assigned the highest priority if the deadline of its current job is the nearest. Ties are broken arbitrarily. An exact schedulability test under EDF for implicit deadline periodic task sets is given in the following theorem.

Theorem 2.2.1. *Under EDF, an implicit deadline periodic task set Γ is schedulable on one processor if the total utilization of Γ is less than or equal to one:*

$$U_\Gamma = \sum_{\tau_i \in \Gamma} u_i \leq 1 \tag{2.12}$$

Note that EDF is *optimal* on uniprocessor systems. That is, if a task set is feasible on such a system, it is also schedulable under EDF.

Rate Monotonic (RM)

Under the Rate Monotonic (RM) scheduling algorithm, each task has a fixed priority. In particular, for any two tasks τ_i and τ_j, if the period of τ_i is shorter than the period of τ_j then the priority of τ_i is higher that that of τ_j.

Such a priority assignment is optimal in the sense that no other fixed task priority assignment rule can schedule a task set which cannot be scheduled by RM [LL73]. However, contrary to EDF, RM is in general not optimal on uniprocessors (see Definition 2.2.3) for real-time periodic task sets.

2.2.5 Multiprocessor Schedulability Analysis

As mentioned in Section 1.2.2, the scheduling problem on multiprocessors is much more complex than on uniprocessor systems. In order to find a solution to this problem, a plethora of scheduling algorithms for hard real-time multiprocessor systems have been proposed in the literature [DB11, BBB15]. Each scheduling algorithm has its advantages and its drawbacks compared to the others, and in fact there is no scheduling algorithm that outperforms the rest in all aspects.

Optimal Global Scheduling Algorithms

On a system comprised of M homogeneous processors, hard real-time scheduling algorithms that achieve a worst-case utilization bound of M exploit job-level migrations and dynamic job priority (recall the classification of scheduling algorithms given in Section 2.2.3). Examples of such algorithms include PFAIR [BCPV96] and LLREF [CRJ06]. Under these optimal global scheduling algorithms, an exact schedulability test for an implicit deadline periodic task set Γ on M processors is:

$$U_\Gamma = \sum_{\tau_i \in \Gamma} u_i \leq M \tag{2.13}$$

that is, any implicit deadline periodic task set with total utilization less than or equal to M is schedulable on M processors. Based on the above equation, we can derive the minimum number of processors M_{OPT} required by an optimal scheduling algorithm to schedule an implicit deadline periodic task set Γ:

$$M_{\text{OPT}} = \lceil U_\Gamma \rceil \tag{2.14}$$

Note that other global scheduling algorithms do not achieve optimality, for instance Global EDF (GEDF).

Partitioned Scheduling Algorithms

Unfortunately, optimal global scheduling algorithms entail high migration and preemption overheads. To avoid such overheads, designers often choose partitioned approaches, where no migration is allowed. Partitioned scheduling approaches are

composed of two phases, an assignment phase and an execution phase. Under partitioned approaches, as the name suggests, in the **first phase** a *schedulable partition* of the initial task set is created. In general, a partition of a set V is defined as a grouping of its elements into non-empty subsets, in such a way that every element is included in one and only one of the subsets. We provide a definition using mathematical notation below.

Definition 2.2.6. *(Partition of a set).* Let V be a set. An x-partition of V is a set, denoted by ${}^{x}V$, where:

$$ {}^{x}V = \{{}^{x}V_1, {}^{x}V_2, \cdots, {}^{x}V_x\}, $$

such that each subset ${}^{x}V_i \subseteq V$, and:

$$ {}^{x}V_i \neq \emptyset \quad \forall\, {}^{x}V_i \quad \text{and} \quad \bigcap_{i=1}^{x} {}^{x}V_i = \emptyset \quad \text{and} \quad \bigcup_{i=1}^{x} {}^{x}V_i = V. $$

As mentioned earlier, in the case of partitioned scheduling algorithms, we are interested in obtaining a *schedulable partition* of a task set, which is defined below.

Definition 2.2.7. *(Schedulable partition of a task set).* Let Γ be a set of periodic real-time tasks. A schedulable partition ${}^{x}\Gamma$ is a partition of Γ that complies with Definition 2.2.6 and guarantees that each subset of ${}^{x}\Gamma$ is schedulable on one processor under the considered local scheduling algorithm.

In particular, consider a task set Γ and an x-partition ${}^{x}\Gamma$ of Γ. Assume that each subset ${}^{x}\Gamma_j \in {}^{x}\Gamma$ is assigned to a separate processor and it is scheduled by a local uniprocessor scheduler $\mathcal{A}$. Then, using the schedulability test provided by Condition (2.11), we have that Γ is schedulable using $\mathcal{A}$ on each processor if:

$$ \sum_{\tau_i \in {}^{x}\Gamma_j} u_i \leq U_{\mathcal{A}}, \quad \forall\, {}^{x}\Gamma_j \in {}^{x}\Gamma \tag{2.15} $$

where $U_{\mathcal{A}}$ is the worst-case utilization bound of $\mathcal{A}$, as defined in Definition 2.2.5. For instance, the worst-case utilization bound of EDF is $U_{\text{EDF}} = 1$ [LL73], therefore we have that Γ is schedulable using Partitioned EDF (PEDF) if:

$$ \sum_{\tau_i \in {}^{x}\Gamma_j} u_i \leq 1, \quad \forall\, {}^{x}\Gamma_j \in {}^{x}\Gamma \tag{2.16} $$

Then, in the **second phase**, at run-time, the local (uniprocessor) scheduler $\mathcal{A}$ is used to schedule the subset of the partition which is assigned to each processor.

From the above discussion, it is clear that the first phase of a partitioned scheduling approach is in fact an instance of the classical *bin-packing* problem [Joh73]. In the bin-packing problem, items of different sizes must be packed into the least amount of bins, which have a certain capacity. In partitioned scheduling, in an analog way, tasks with different utilizations must be partitioned into the least amount of processors. The "capacity" of each processor is determined by the worst-case utilization bound

$U_{\mathcal{A}}$ of the local scheduling algorithm. The equivalence of partitioning schemes and the bin-packing problem leads to the following two limitations.

Limitation 1. An optimal solution to the bin-packing problem is one that minimizes the number of bins required to pack the items. Analogously, an optimal partitioning of the set of tasks is one that requires the least amount of processors to assign all tasks, while guaranteeing schedulability on all processors. In both cases, finding an optimal solution is NP-hard [GJ79]. In order to tackle the NP-hardness of the problem, several heuristics have been proposed [Joh74] to find approximate solutions. We provide an overview of the most commonly used heuristics in Section 2.2.6. These heuristics are rather simple and fast, but they do not guarantee the optimality of the provided solution.

Limitation 2. Consider a system composed of M processors. Even if we determine the optimal partitioning of tasks to processors, no partitioned scheduling algorithm can guarantee the worst-case utilization bound of M (recall Equation (2.13)) provided by optimal global algorithms. In general, the worst-case utilization bound of a partitioned scheduling algorithm on M processors can reach at most $(M+1)/2$ [ABD08]. This phenomenon is termed *utilization loss* and implies that partitioned algorithms cannot, in general, exploit the available processing resources in an optimal way. In fact, a partitioned approach may require twice as many processors to schedule certain task sets compared to an optimal global scheduler.

In the rest of this dissertation, we refer to the two above limitations as *bin-packing issues*.

2.2.6 Partitioning Heuristics

Consider a set Γ of N tasks (items) and a set Π of M homogeneous processors (bins). Each processor uses a local scheduler $\mathcal{A}$ with worst-case utilization bound $U_{\mathcal{A}}$, and each task τ_i has utilization u_i. As mentioned in **Limitation 1** of Section 2.2.5, an optimal partitioning of the task set Γ is a partitioning that uses the least amount of processors and satisfies Condition (2.15). Deriving such optimal partitioning, which is an instance of the bin-packing problem, is NP-hard. Given the complexity of this problem, several heuristics have been proposed to solve it. In the following, we summarize some of the most common heuristics used to solve the partitioning problem. These heuristics assign each task $\tau_i \in \Gamma$ to a certain processor $\pi_k \in \Pi$ by considering one task at a time, following a certain sequence. For First-Fit, Best-Fit and Worst-Fit, in particular, the current task to be assigned is determined by following the order of tasks appearance in Γ, e.g., τ_1 is assigned first and τ_N last.

All the partitioned heuristics described in what follows utilize the concept of processor utilization, defined below.

Definition 2.2.8. *(Utilization of a processor).* Let Γ_k denote the set of tasks currently assigned to processor π_k. Then, the utilization σ_k of processor π_k is equal to the sum of the utilizations of the tasks assigned to π_k, i.e.:

$$\sigma_k = \sum_{\tau_i \in \Gamma_k} u_i \tag{2.17}$$

Note that, in the beginning of all partitioned heuristics listed below, $\Gamma_k = \emptyset$ and $\sigma_k = 0$ for each π_k.

- **First-Fit (FF).** A task τ_i is assigned to the lowest-indexed processor π_k that can contain it. That is, the index k of π_k is determined by:

$$k = \min\{j \mid u_i + \sigma_j \leq U_{\mathcal{A}}\}$$

  If the condition enclosed by the braces is not satisfied by any processor, a new processor is instantiated and τ_i is assigned to it.
- **Best-Fit (BF).** A task τ_i is assigned to a processor π_k such that π_k will have the *minimal* residual utilization ($U_{\mathcal{A}} - \sigma_k$) after the assignment. That is, the index k of π_k is determined by:

$$k = \min\{j \mid (u_i + \sigma_j) \text{ is closest to, without exceeding, } U_{\mathcal{A}}\}$$

  If the condition enclosed by the braces is not satisfied by any processor, a new processor is instantiated and τ_i is assigned to it.
- **Worst-Fit (WF).** A task τ_i is assigned to a processor π_k such that π_k will have the *maximal* residual utilization ($U_{\mathcal{A}} - \sigma_k$) after the assignment.

$$k = \min\{j \mid (u_i + \sigma_j) \text{ is minimal and does not exceed } U_{\mathcal{A}}\}$$

  If the condition enclosed by the braces is not satisfied by any processor, a new processor is instantiated and τ_i is assigned to it.

Recall that for EDF the worst-case utilization bound $U_{\mathcal{A}}$ is $U_{\mathsf{EDF}} = 1$ (see Equation (2.16)).

As shown in [Joh73], these heuristics achieve better performance if they are preceded by a sorting of the input task set. Usually, the input task set is sorted by decreasing utilization. The approaches composed of a first phase, which sorts the input task set by decreasing utilization, and a second phase, which applies one of the aforementioned heuristics (FF, BF, WF), are termed **First-Fit Decreasing (FFD)**, **Best-Fit Decreasing (BFD)**, and **Worst-Fit Decreasing (WFD)**, respectively.

The performance of a partitioning heuristic can be measured by its *approximation ratio*. Let $OPT(\Gamma)$ be the number of processors needed by an optimal partitioning scheme to assign a certain task set Γ. Consider a certain partitioning heuristic H, which requires $H(\Gamma)$ processors to assign the same task set Γ. Then, the approximation ratio of H, denoted by $\mathcal{R}_H$, ensures that for *any* task set Γ:

$$H(\Gamma) \leq \mathcal{R}_H \cdot OPT(\Gamma) \tag{2.18}$$

The approximation ratios for FF, BF, and FFD are $17/10$, $17/10$, and $11/9$, respectively [CGJ96, GJ79, Yue91].

2.2.7 EDF-fm Semi-partitioned Algorithm

As summarized in Section 2.2.5, global scheduling algorithms can be optimal for multiprocessor systems leading to a full exploitation of the available processors.

However, they incur high migration and preemption overheads, which may limit their applicability. Moreover, they incur significant memory overhead in distributed memory systems, as explained in Section 1.3.2. By contrast, partitioned approaches as PEDF show low preemption overheads and neither migration nor memory overheads. However, they are affected by bin-packing issues and in general may not exploit the available processing resources in an optimal way.

In Chapters 5 and 6 of this thesis we consider semi-partitioned scheduling algorithms, which represent a middle ground between global and partitioned algorithms. As mentioned in Chapter 1, under semi-partitioned algorithms 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. Semi-partitioned algorithms aim at ameliorating the bin-packing issues of partitioned scheduling without incurring the excessive overheads of global scheduling. In particular, in Chapter 5 we exploit the EDF-fm scheduling algorithm [ABD08], which is described in the rest of this section. 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*. By contrast, in Chapter 6 we propose a novel scheduling algorithm, EDF-ssl which is based on some concepts and properties of the EDF-os scheduling algorithm [AEDC14] summarized in Section 2.2.8.

As mentioned in Section 1.4.2, EDF-fm can have great benefits for distributed memory MPSoCs. However, it provides only soft real-time guarantees to the scheduled tasks. Since many definitions of soft real-time behavior exist, we provide below the definition of a SRT algorithm adopted in this thesis.

Definition 2.2.9. *(Soft Real-Time (SRT) scheduling algorithm).* A scheduling algorithm is said to be SRT when it allows tasks to miss their deadlines by a bounded value called *tardiness*.

Note that EDF-fm falls into this definition of SRT algorithm. In particular, under EDF-fm we can compute a bound of the tardiness of each task. A definition of tardiness bound is given below.

Definition 2.2.10. *(Tardiness bound).* A task τ_i is said to have a *tardiness bound* Δ_i if each job $\tau_{i,k}$ of τ_i does not miss its deadline $d_{i,k}$ by more than Δ_i. That is, denoting the completion time of job $\tau_{i,k}$ by $z_{i,k}$:

$$z_{i,k} \leq (d_{i,k} + \Delta_i), \forall k \in \mathbb{N}_0$$

We describe now the EDF-fm scheduling algorithm, as presented in [ABD08], in greater detail. In EDF-fm, tasks can be either fixed or migrating. Migrating tasks migrate between exactly two processors, with the restriction that migration can only happen at job boundaries. The EDF-fm approach consists of two phases: the *assignment phase* and the *execution phase*, which are summarized in what follows.

Assignment phase

Consider the following definitions:

Definition 2.2.11. *(Task share).* A task τ_i is said to have a share $s_{i,k}$ on π_k when a part $s_{i,k}$ of its utilization u_i is assigned to π_k.

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

Definition 2.2.12. *(Task fraction).* Given $s_{i,k}$, π_k executes a fraction $\varphi_{i,k} = \frac{s_{i,k}}{u_i}$ of τ_i's total execution requirement.

In the assignment phase each task is assigned to either one processor (fixed task) or two processors (migrating task). In particular, the assignment phase assigns tasks in sequence to processors. Since EDF-fm uses EDF as local scheduling algorithm, the capacity of each processor π_k (the maximum utilization that can be assigned to it) is 1 (see Equation (2.12) on page 33), therefore the condition:

$$\sigma_k \leq 1, \ \forall \, \pi_k \in \Pi \tag{2.19}$$

must be satisfied.

In particular, in the assignment phase tasks are assigned to a processor π_k until its capacity is exhausted. Recall that σ_k denotes the total utilization assigned to processor π_k (see Definition 2.2.8). In the case of EDF-fm, σ_k is equal to the sum of *shares* assigned to π_k:

$$\sigma_k \triangleq \sum_{\tau_i \in \Gamma_k} s_{i,k} \tag{2.20}$$

where Γ_k is the set of tasks with non-zero shares on π_k.

If a task τ_i cannot entirely fit on processor π_k, then a share $s_{i,k} = 1 - \sigma_k$ of its utilization is assigned to π_k. This makes sure that, after this assignment, $\sigma_k = 1$, i.e., π_k is fully utilized. The remaining utilization $s_{i,k+1} = (u_i - s_{i,k})$ of τ_i is assigned to the next processor, π_{k+1}. The assignment phase of EDF-fm ensures that at most two migrating tasks are assigned to any processor (see an example in Figure 2.4).

Moreover, on a processor with two migrating tasks (τ_i and τ_j), EDF-fm requires that the sum of the migrating tasks' utilization (denoted by σ_k^{mig}) does not exceed one:

$$\sigma_k^{\text{mig}} = u_i + u_j \leq 1, \tag{2.21}$$

This condition is automatically satisfied if the maximum utilization of any task is limited to $1/2$, given the fact that at most two migrating tasks can be assigned to a single processor. However, tasks that exceed this utilization limit can still be scheduled by EDF-fm, provided that Condition (2.21) is respected on all the processors. Note that if no limit on maximum task utilizations is set, EDF-fm is not optimal, because it cannot fully exploit the available processors for all possible input task sets.

Example 2.2.1. Given the task set $\{\tau_1 = (C_1 = 3, T_1 = 10), \tau_2 = (2,5), \tau_3 = (2,5), \tau_4 = (1,2), \tau_5 = (1,2), \tau_6 = (2,5), \tau_7 = (1,2)\}$, the EDF-fm algorithm derives the task assignment shown in Fig. 2.4. For instance, task τ_3 cannot entirely fit onto π_1 in Fig. 2.4, thus its utilization is split between π_1 and π_2 with shares $s_{3,1} = 3/10$ and $s_{3,2} = 1/10$, respectively.

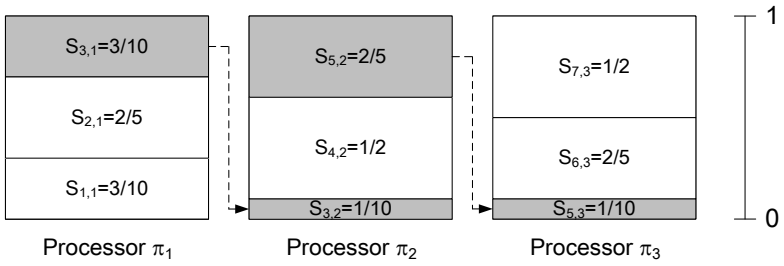

Figure 2.4: *EDF-fm assignment of the task set considered in Example 2.2.1. Tasks τ_1, τ_2, τ_4, τ_6, and τ_7 are fixed, i.e., their whole utilization is assigned to a single processor. For instance, task τ_1 has utilization $u_1 = 3/10$ and the share $s_{1,1}$ of τ_1 on π_1 is equal to its whole utilization, that is, $s_{1,1} = u_1$. By contrast, tasks τ_3 and τ_5 are migrating tasks. Their shares on processors are highlighted with a shaded area. For instance, τ_3 cannot entirely fit onto π_1, thus its utilization is split between π_1 and π_2 with shares $s_{3,1} = 3/10$ and $s_{3,2} = 1/10$, respectively.*

Execution phase

The execution phase employs a simple online scheduling algorithm that is derived from EDF and ensures bounded tardiness with a minimal overhead compared to a canonical EDF scheduler. Let τ_i be a migrating task that migrates between processor π_k and π_{k+1}. Then, jobs belonging to a task τ_i are assigned at run-time such that in the long run the fraction of τ_i's workload executed on π_k (π_{k+1}) is close to $\varphi_{i,k}$ ($\varphi_{i,k+1}$). This result is achieved by leveraging results from PFAIR scheduling [BCPV96]. We recall that EDF-fm allows only *restricted migrations*. As explained in Section 2.2.3, this means that different jobs of the same task can execute on different processors. However, each job can only be executed on one processor.

For instance, according to the share assignment depicted in Figure 2.4, task τ_3 releases its jobs on processors π_1 and π_2 according to the pattern shown in Figure 2.5. Task τ_3 releases a job every period T_3, either to π_1 or to π_2. On average 1 out of 4 jobs of τ_3 are assigned to π_2 and the remaining 3 jobs are assigned to π_1. In the long run (the release pattern continues indefinitely), the number of jobs released on π_1 are three times the number of jobs released on π_2. This is due to the fact that the share $s_{3,1}$ of τ_3 assigned to π_1 is three times larger than the share $s_{3,2}$ of τ_3 assigned to π_2.

Jobs released on a processor are prioritized among each other using a local EDF scheduler. The job release pattern of migrating tasks under EDF-fm, mentioned above, prevents the overloading on processors in the long run. However, it creates *temporary* overloading on processors, which in turn leads to tardiness. In particular, when two migrating tasks, τ_i and τ_j, are assigned to π_k, the tardiness bound under EDF-fm for a fixed task τ_u assigned to the same processor is given by:

$$\Delta(\tau_u) = \frac{C_i \cdot (\varphi_{i,k} + 1) + C_j \cdot (\varphi_{j,k} + 1) - T_u \cdot (1 - \sigma_k)}{1 - s_{i,k} - s_{j,k}} \tag{2.22}$$

where C_i and C_j are the worst-case execution times of τ_i and τ_j (as defined in Section 2.2.1), and T_u is the period of task τ_u. Finally, σ_k is the total utilization assigned

Figure 2.5: *Release pattern of jobs of task τ_3 between processors π_1 and π_2, according to the share assignment of τ_3 in Figure 2.4.*

to π_k, i.e., the sum of fixed tasks' utilization and migrating tasks' shares allocated to π_k (see Equation (2.20)). Note that in Equation (2.22) the tardiness bound of EDF-fm is denoted as $\Delta(\tau_u)$, whereas in Definition 2.2.10 we denote the tardiness bound of a task τ_u as Δ_u. Throughout this thesis, we will use the latter notation if the context makes clear that Δ_u is the tardiness bound of task τ_u.

In contrast with fixed tasks, in EDF-fm migrating tasks do not miss any deadline, therefore their tardiness bound is zero.

2.2.8 EDF-os Semi-partitioned Algorithm

In order to tackle the sub-optimality of the EDF-fm scheduling algorithm, Anderson et al. in [AEDC14] propose EDF-os (EDF-based optimal semi-partitioned scheduling). In what follows, we summarize the features of EDF-os which are leveraged in our EDF-ssl scheduling algorithm presented in Chapter 6.

Similar to EDF-fm, EDF-os is also a SRT scheduling algorithm (see Definition 2.2.9). These two algorithms share some definitions and concepts, but EDF-os introduces modifications to both the assignment and execution phases of EDF-fm to achieve optimality. As in EDF-fm, under EDF-os tasks can be either *fixed* or *migrating*, with migrations only allowed at job boundaries. However, in EDF-os migrating tasks are allowed to migrate among any number of processors, not only between two processors as in EDF-fm. Each task τ_i is assigned a (potentially zero) *share* $s_{i,k}$ of the available utilization of a processor π_k, following Definition 2.2.11.

If task τ_i is migrating, it has non-zero shares on several processors. If τ_i is fixed, it has non-zero shares on a single processor. The **assignment phase** in EDF-os ensures that the cumulative sum of the shares of a task over all the processors equals the task utilization, that is:

$$u_i = \sum_{k=1}^{M} s_{i,k}$$

where M is the total number of processors in the system. Similar to EDF-fm, the total utilization assigned to processor π_k is denoted by σ_k and derived using Equation (2.20).

Also under EDF-os, the total utilization assigned to a processor must always be equal to or lower than the available processor utilization (which is 1). That is, for each processor π_k:

$$\sigma_k \leq 1 \tag{2.23}$$

Condition (2.23) above is ensured by the assignment phase of EDF-os to avoid the over-utilization of any processor in the long run. In fact, Condition (2.23) is identical to Condition (2.12) used in EDF-fm.

In the **execution phase**, EDF-os enforces that, in the long run, the fraction of workload generated by task τ_i on π_k is equal to the task fraction $\varphi_{i,k}$, given by Definition 2.2.12. Similar to EDF-fm, this long-run workload distribution according to task fractions is obtained by leveraging results from Pfair scheduling [BCPV96]. In particular, out of the *first* ν consecutive jobs released by τ_i, EDF-os ensures that the number of jobs released on processor π_k is between $\lfloor \varphi_{i,k} \cdot \nu \rfloor$ and $\lceil \varphi_{i,k} \cdot \nu \rceil$ (Property 1 in [AEDC14]). In turn, out of *any* c consecutive jobs of a migrating task τ_i, the number of jobs released on π_k (indicated as $c_{i,k}$) is bounded by the following expression:

$$c_{i,k} \leq \varphi_{i,k} \cdot c + 2 \tag{2.24}$$

The above expression is given by Property 6 in [AEDC14]. For a more detailed explanation of assignment rules for jobs of migrating tasks, the reader is referred to [ABD08] and [AEDC14].

Tardiness bounds for both fixed and migrating tasks under EDF-os are derived in [AEDC14]. We do not report these bounds because they are not relevant for the contributions of this thesis.

2.3 HRT Scheduling of Acyclic CSDF Graphs [BS11, BS12]

As mentioned in Section 1.2.2 (on page 15, under Class III of scheduling algorithms), several approaches that bridge the gap between dataflow MoCs and real-time task models have been proposed in recent years. In this thesis, in particular, among these approaches we consider the scheduling technique proposed in [BS11, BS12].

Bamakhrama and Stefanov in [BS11, BS12] consider applications specified as acyclic CSDF graphs and show that the set of N actors $A = \{A_1, A_2, \cdots, A_N\}$ of an input CSDF graph G can be converted to a set of N real-time periodic tasks $\Gamma_G = \{\tau_1, \tau_2, \cdots, \tau_N\}$. This conversion allows a designer to apply algorithms from hard real-time scheduling theory to derive in a fast and analytical way the minimum number of processors that guarantee the required performance of an application and the partitioning of tasks to processors.

In particular, for each actor $A_i \in A$ of the input CSDF graph, the analysis in [BS11, BS12] derives the parameters of the corresponding real-time periodic task $\tau_i = (C_i, T_i, S_i)$, where C_i is the WCET of the task, T_i is the task period, S_i is the start time of the task (as described in Section 2.2.1). In the rest of this section, we describe how these parameters (C_i, T_i, S_i) are derived. Then, we show how the size of buffers

which implement inter-task communication over edges can be derived. Finally, we summarize the correspondence between the dataflow notation for the input CSDF graph G and the real-time theory notation for the derived periodic task set Γ_G.

WCET of Actors (C_i)

The analysis in [BS11,BS12] begins with the computation of the WCET C_i of each CSDF actor A_i. The value of C_i is derived as follows. First, the WCET $C_{i,k}$ of each phase k of actor A_i is computed. This WCET includes both the worst-case communication and computation time required by phase k of A_i, and is calculated using the following equation:

$$C_{i,k} = C^R \cdot \sum_{e_l \in \text{inp}(A_i)} y_i^l(k) + C^W \cdot \sum_{e_r \in \text{out}(A_i)} x_i^r(k) + C_i^C(k) \tag{2.25}$$

In Equation (2.25), C^R (C^W) represents the worst case time needed to read (write) a single token from (to) an input (output) channel in the considered hardware platform; $y_i^l(k)$ ($x_i^r(k)$) is the number of tokens read (written) by actor A_i from (to) edge e_l (e_r) by phase k of A_i; $\text{inp}(A_i)$ ($\text{out}(A_i)$) is the set of input (output) edges of A_i; and $C_i^C(k)$ is the worst-case computation time of phase k of actor A_i. Note that $C_i^C(k)$ includes also the worst-case overhead incurred by the underlying scheduler (e.g., EDF), following the analysis of [Dev06].

The WCET C_i of actor A_i is derived by finding the maximum value among the WCET $C_{i,k}$ of each phase k of A_i, that is:

$$C_i = \max_{k=1}^{\Omega_i} (C_{i,k}) \tag{2.26}$$

where $C_{i,k}$ is obtained using Equation (2.25).

Given the WCET C_i of each actor A_i in G, we can represent the WCETs of all actors in G using the **WCET vector** $\vec{C}$, where each component C_i of $\vec{C}$ is the WCET of actor A_i.

Minimum Period of Actors (T_i)

Based on the properties of the graph and on the WCET of actors given by Equation (2.26), the minimum period $T_i \in \mathbb{N}$ of actor A_i can be calculated using the following expression:

$$T_i = \frac{Q}{q_i} \left\lceil \frac{\eta}{Q} \right\rceil \tag{2.27}$$

where q_i is the number of repetitions of actor A_i per graph iteration, $\eta = \max_{A_i \in A}\{C_i\, q_i\}$ (recall that C_i is the WCET of A_i), and $Q = \text{lcm}\{q_1, q_2, \cdots, q_N\}$.

The period T_i of actor A_i, obtained using Equation 2.27, is minimum. However, in some cases a designer may want longer periods of actors. These longer periods can be derived by multiplying the minimum period of each actor by a positive integer factor constant among all actors.

Given the period T_i of each actor A_i in G, we can represent the periods of all actors in G using the **period vector** $\vec{T}$, where each component T_i of $\vec{T}$ is the period of actor A_i.

Example 2.3.1. Consider again the CSDF graph shown in Figure 2.2 on page 27. We have already derived in Section 2.1.2 the repetition vector of the graph $\vec{q} = [3, 2, 3]$. Assume that its WCET vector is $\vec{C} = [C_1, C_2, C_3] = [1, 2, 2]$. Then, it follows that $\eta = 6$ and $\vec{T} = [T_1, T_2, T_3] = [2, 3, 2]$.

In general, the derived period vector $\vec{T}$ satisfies the condition:

$$q_1 T_1 = q_2 T_2 = \cdots = q_N T_N = H \tag{2.28}$$

where H is referred to as *iteration period*, and represents the duration needed by the graph to complete one iteration (recall Definition 2.1.4 on page 28).

Earliest Start Times of Actors (S_i)

The earliest start time $S_j \in \mathbb{N}_0$ of an actor A_j is derived using the following expression:

$$S_j = \begin{cases} 0 & \text{if } \text{prec}(A_j) = \emptyset \\ \max_{A_i \in \text{prec}(A_j)} \left(S_{i \to j}\right) & \text{if } \text{prec}(A_j) \neq \emptyset \end{cases} \tag{2.29}$$

where:

$$S_{i \to j} = \min_{t \in [0, S_i + H]} \left\{ t : \underset{[S_i, \max(S_i, t) + k)}{\text{prd}^{\text{S}}} (A_i, e_u) \geq \underset{[t, \max(S_i, t) + k]}{\text{cns}^{\text{S}}} (A_j, e_u),\ \forall k = 0, 1, \cdots, H \right\} \tag{2.30}$$

where:

- H is the iteration period as defined by Equation (2.28);
- S_i is the start time of a predecessor actor A_i;

and the two functions (prd^{S}, cns^{S}) used in Expression 2.30 are defined as follows.

Definition 2.3.1. *(Cumulative production function for start times calculation).* The cumulative production function used to derive start times is denoted by $\text{prd}^{\text{S}}_{[t_s, t_f)}(A_i, e_u)$ and represents the total number of tokens produced by actor S_i to edge e_u during the time interval $[t_s, t_f)$.

Definition 2.3.2. *(Cumulative consumption function for start times calculation).* The cumulative consumption function used to derive start times is denoted by $\text{cns}^{\text{S}}_{[t_s, t_f]}(A_j, e_u)$ and represents the total number of tokens consumed by actor A_j from edge e_u during the time interval $[t_s, t_f]$.

Note that, for the purpose of computing the start times of actor A_j, the cumulative production function $\text{prd}^{\text{S}}(A_i, e_u)$ assumes that token production happens as **late** as

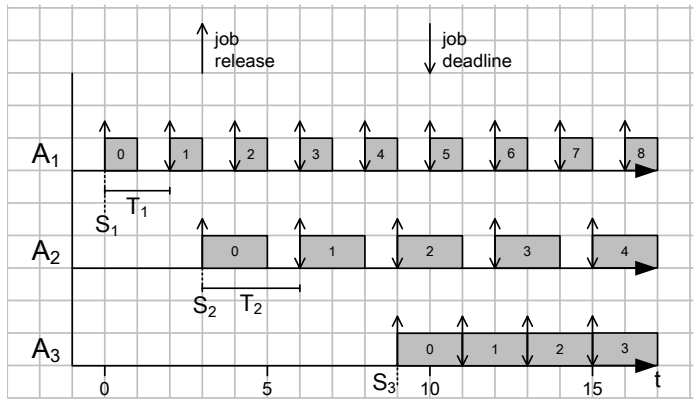

Figure 2.6: *Hard real-time scheduling of the CSDF graph in Figure 2.2 on page 27 derived using the methodology of [BS11, BS12]. For instance, the derived period of A_1 is $T_1 = 2$ and the start time of A_1 is $S_1 = 0$. This means that the first invocation of A_1 is released at time 0, and the successive invocations are released periodically, every 2 time units. The schedule continues indefinitely, only its initial part is shown.*

possible, i.e., at the deadline of each invocation of predecessor actor A_i. Conversely, the cumulative consumption function $\text{cns}^S(A_j, e_u)$ assumes that token consumption happens as **early** as possible, i.e., at the release of each invocation of actor A_j. These assumptions, which make the calculation of actor start times safe, are emphasized by the superscript S in prd^S and cns^S.

Given the start time S_i of each actor A_i in G, we can represent the start times of all actors in G using the **start time vector** $\vec{S}$, where each component S_i of $\vec{S}$ is the start time of actor A_i.

Example 2.3.2. In Example 2.3.1, assuming a WCET vector $\vec{C} = [1, 2, 2]$ we derived the period vector $\vec{T} = [2, 3, 2]$ of the graph G shown in Figure 2.2 on page 27. Then, based on Expression (2.29) we derive the earliest start time vector $\vec{S} = [0, 3, 9]$. Therefore, the real-time periodic task set corresponding to G is completely defined as:

$$\Gamma_G = \{\tau_1 = (C_1 = 1, T_1 = 2, S_1 = 0), \tau_2 = (2, 3, 3), \tau_3 = (2, 2, 9)\}$$

Given the complete specification of the obtained real-time periodic task set, a designer can apply algorithms from hard real-time scheduling theory to derive in a fast and analytical way the minimum number of processors that guarantee the required performance of an application and the partitioning of tasks to processors. For instance, the total utilization U_{Γ_G} is $13/6$, therefore even an optimal global scheduling algorithm would require at least 3 processors to schedule Γ_G (see Equation (2.14)).

The periodic schedule of G, resulting from the derived task set Γ_G, is visualized in Figure 2.6. In the figure, notice that the first invocation of each actor A_i is released at the start time S_i, obtained using Expression (2.29). Then, successive invocations of each actor A_i are released periodically, according to the actor's period T_i.

Minimum Buffer Sizes

Given the period T_i and start time S_i of each actor $A_i \in A$, the minimum size b_u of the buffer which implements the communication over edge $e_u = (A_i, A_j)$ is given by:

$$b_u = \max_{k \in [0,1,\cdots,H]} \left\{ \underset{[S_i, \max\{S_i,S_j\}+k]}{\text{prd}^{\text{B}}}(A_i, e_u) - \underset{[S_j, \max(S_i,S_j)+k)}{\text{cns}^{\text{B}}}(A_j, e_u) \right\} \tag{2.31}$$

where:

- H is the iteration period as defined by Equation (2.28);
- S_i and S_j are the start times actors A_i and A_j, respectively;

and the two functions (prd^{B}, cns^{B}) used in Expression 2.30 are defined as follows.

Definition 2.3.3. *(Cumulative production function for buffer sizes calculation).* The cumulative production function used to derive buffer sizes is denoted by $\text{prd}^{\text{B}}{}_{[t_s,t_f]}(A_i, e_u)$ and represents the total number of tokens produced by actor S_i to edge e_u during the time interval $[t_s, t_f]$.

Definition 2.3.4. *(Cumulative consumption function for buffer sizes calculation).* The cumulative consumption function used to derive buffer size is denoted by $\text{cns}^{\text{B}}{}_{[t_s,t_f)}(A_j, e_u)$ and represents the total number of tokens consumed by actor A_j from edge e_u during the time interval $[t_s, t_f)$.

Note that, for the purpose of computing the buffer size b_u, the cumulative production function $\text{prd}^{\text{B}}(A_i, e_u)$ assumes that token production happens as **early** as possible, i.e., at the release of each invocation of predecessor actor A_i. Conversely, the cumulative consumption function $\text{cns}^{\text{B}}(A_j, e_u)$ assumes that token consumption happens as **late** as possible, i.e., at the deadline of each invocation of actor A_j. These assumptions, which make the calculation of buffer sizes safe, are emphasized by the superscript B in prd^{B} and cns^{B}.

Correspondence Between Dataflow and Real-Time Theory Notations

The analysis in [BS11, BS12] creates a one-to-one correspondence between actor A_i of the input CSDF graph G and real-time periodic task τ_i of the derived periodic task set Γ_G. In this thesis, we will leverage either the dataflow notation for the (C)SDF graph G (see Sections 2.1.1 and 2.1.2) or the real-time theory notation for the periodic task set Γ_G (see Section 2.2.1), depending on the problem we want to address. For the sake of clarity, Table 2.2 shows the correspondence between these two notations. Recall that we denote the jth invocation of actor A_i as $A_{i,j}$ (with $j \in \mathbb{N}_0$), therefore $A_{i,0}$ represents the *first* invocation of actor A_i. Note that, in Table 2.2, the earliest start time and latest completion time of an invocation $A_{i,j}$ refer to the schedule generated by the methodology of [BS11, BS12].

Note that Chapter 5 and 6 of this thesis extend the methodology of [BS11, BS12]. Therefore, in those chapters the correspondence between dataflow and real-time theory notations shown in Table 2.2 is used extensively.

Dataflow notation for G	**Real-time notation for Γ_G**
Actor A_i	Task τ_i
Invocation $A_{i,j}$ of A_i	Job $\tau_{i,j}$ of τ_i
Earliest start time of $A_{i,0}$	Start time S_i of τ_i
Earliest start time of $A_{i,j}$	Release time $r_{i,j}$ of $\tau_{i,j}$
Latest completion time of $A_{i,j}$	Deadline $d_{i,k}$ of $\tau_{i,j}$

Table 2.2: *Correspondence between dataflow and real-time theory notations resulting from the methodology of [BS11, BS12].*

In addition, both Chapter 5 and 6 use the concept of *stateless* real-time periodic tasks. In general, a task is said to be *stateless* if it complies to the definition below.

Definition 2.3.5. *Stateless task (general).* A task is said to be *stateless* if it does not keep an internal state between two successive jobs.

Using the methodology of [BS11, BS12], we recall that each task τ_i corresponds to actor A_i of the input CSDF graph G. Therefore

Definition 2.3.6. *Stateless task (in [BS11, BS12]).* In the scheduling technique of [BS11, BS12], a task τ_i is said to be *stateless* if it corresponds to a (C)SDF actor A_i which is stateless (i.e., A_i complies to Definition 2.1.5 on page 28).