Chapter 2

Background

THIS chapter is dedicated to an overview of the background material needed to understand the novel research contributions of this thesis presented in the following chapters. We first provide a summary of some mathematical notations used throughout this thesis in Table 2.1.

Symbol	**Meaning**
$\mathbb{N}$	The set of natural numbers excluding zero
$\mathbb{N}_0$	$\mathbb{N} \cup \{0\}$
$\mathbb{Z}$	The set of integers
$\|x\|$	The cardinality of a set x
$\lceil x \rceil$	The smallest integer that is greater than or equal to x
$\lfloor x \rfloor$	The greatest integer that is smaller than or equal to x
$\hat{x}$	The maximum value of x
$\check{x}$	The minimum value of x
$\vec{x}$	The vector x
lcm	The least common multiple operator
mod	The integer modulo operator
${}^{x}V$	An x-partition of a set V (see Definition 2.2.1)

Table 2.1: *Summary of mathematical notations.*

Then, in Section 2.1, we present the dataflow MoCs that are used in this thesis. In Section 2.2, we present some results and definitions from the hard real-time (HRT) scheduling theory relevant to the context of this thesis. Finally, in Section 2.3 and 2.4, we describe the HRT analysis for the adopted dataflow MoCs.

2.1 Dataflow Models of Computation

As mentioned in Section 1.2.2, dataflow MoCs have been identified as the most suitable parallel MoCs to express the available parallelism in streaming applications. In this section, we present the dataflow MoCs considered in this thesis, that is, the CSDF and SDF MoCs are given in Section 2.1.1 and the MADF MoC is given in Section 2.1.2.

2.1.1 Cyclo-Static/Synchronous Data Flow (CSDF/SDF)

An application modeled as a CSDF [16] is defined as a directed graph $G = (\mathcal{A}, \mathcal{E})$. G consists of a set of actors $\mathcal{A}$, which corresponds to the graph nodes, that communicate with each other through a set of communication channels $\mathcal{E} \subseteq \mathcal{A} \times \mathcal{A}$, which corresponds to the graph edges. Actors represent computations while communication channels represent data dependencies among actors. A communication channel $E_u \in \mathcal{E}$ is a first-in first-out (FIFO) buffer and it is defined by a tuple $E_u = (A_i, A_j)$, which implies a directed connection from actor A_i (called *source*) to actor A_j (called *destination*) to transfer data, which is divided in atomic data objects called *tokens*. An actor receiving an input data stream of the application from the environment is called *input actor* and an actor producing an output data stream of the application to the environment is called *output actor*.

An actor fires (executes) when there are enough tokens on all of its input channels. Every actor $A_i \in \mathcal{A}$ has an *execution sequence* $[f_i(1), f_i(2), \cdots, f_i(\phi_i)]$ of length ϕ_i, i.e., it has ϕ_i phases. This means that the execution of each phase $1 \leq \phi \leq \phi_i \in \mathbb{N}$ of actor A_i is associated with a certain function $f_i(\phi)$. As a consequence, the execution time of actor A_i is also a sequence $[C_i(1), C_i(2), \cdots, C_i(\phi_i)]$ consisting of the worst-case execution time (WCET) values for each phase. Every output channel E_u of actor A_i has a predefined token *production sequence* $[x_i^u(1), x_i^u(2), \cdots, x_i^u(\phi_i)]$ of length ϕ_i. Analogously, token consumption from every input channel E_u of actor A_i is a predefined sequence $[y_i^u(1), y_i^u(2), \cdots, y_i^u(\phi_i)]$, called *consumption sequence*. Therefore, the $k-$th time that actor A_i is fired, it executes function $f_i(((k-1) \mod \phi_i) + 1)$, produces $x_i^u(((k-1) \mod \phi_i) + 1)$ tokens on each output channel E_u, and consumes $y_i^u(((k-1) \mod \phi_i) + 1)$ tokens from each input channel E_u. The total number of produced tokens by actor A_i on channel E_u during its first n invocations and the total number of consumed tokens from the same channel by A_j during its first n invocations are $X_i^u(n) = \sum_{l=1}^{n} x_i^u(((l-1) \mod \phi_i) + 1)$ and $Y_j^u(n) = \sum_{l=1}^{n} y_j^u(((l-1) \mod \phi_j) + 1)$, respectively.

An important property of the CSDF model is the ability to derive a schedule for the actors at design-time. In order to derive a valid static schedule for a CSDF graph at design-time, it has to be *consistent* and *live*.

Theorem 2.1.1 (From [16]). *In a CSDF graph G, a repetition vector* $\vec{q} = [q_1, q_2, \cdots, q_{|\mathcal{A}|}]^T$ *is given by*

$$\vec{q} = \Theta \cdot \vec{r} \quad with \quad \Theta_{ik} = \begin{cases} \phi_i & if\ i = k \\ 0 & otherwise \end{cases} \tag{2.1}$$

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

$$\Gamma \cdot \vec{r} = \vec{0} \tag{2.2}$$

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

$$\Gamma_{ui} = \begin{cases} X_i^u(\phi_i) & if\ actor\ A_i\ produces\ on\ channel\ E_u \\ -Y_i^u(\phi_i) & if\ actor\ A_i\ consumes\ from\ channel\ E_u \\ 0 & otherwise. \end{cases} \tag{2.3}$$

Theorem 2.1.1 shows that a repetition vector and hence a valid static schedule can only exist if the balance equation, given as Equation (2.2), has a non-trivial solution [16]. A graph G that meets this requirement is said to be consistent. An entry $q_i \in \vec{q} = [q_1, q_2, \cdots, q_{|\mathcal{A}|}]^T \in \mathbb{N}^{|\mathcal{A}|}$ denotes how many times an actor $A_i \in \mathcal{A}$ executes in every graph iteration of G. If a deadlock-free schedule can be found, G is said to be live. When every actor $A_i \in \mathcal{A}$ in G has a single phase, i.e., $\phi_i = 1$, the graph G is a Synchronous Data Flow (SDF) [52] graph, meaning that the SDF MoC is a subset of the CSDF MoC.

For example, Figure 2.2(b) shows a CSDF graph. The graph has a set $\mathcal{A} = \{A_1, A_2, A_3, A_4, A_5\}$ of five actors and a set $\mathcal{E} = \{E_1, E_2, E_3, E_4, E_5\}$ of five FIFO channels that represent the data dependencies between the actors. In this graph, there is one input actor (i.e., A_1) and one output actor (i.e., A_5). Each actor has different number of phases, an execution time sequence, and production/consumption sequences on different channels. For instance, actor A_1 has two phases, i.e., $\phi_1 = 2$, its execution time sequence (in time units) is $[C_1(1), C_1(2)] = [1, 1]$ and its token production sequence on channel E_4 is $[0, 1]$. Then, according to Equations (2.1), (2.2), and (2.3) in Theorem 2.1.1, we can derive the repetition vectors $\vec{q}$ as follows:

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

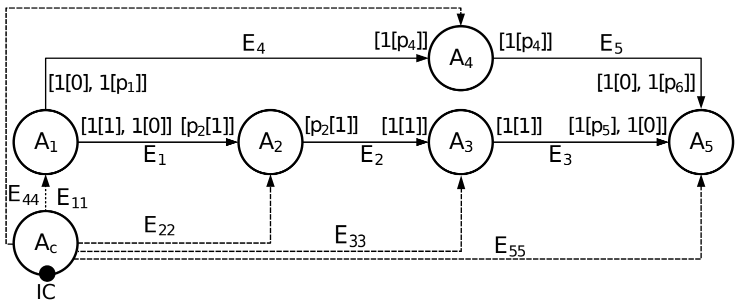
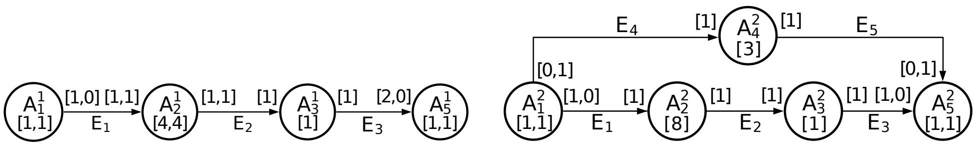


Figure 2.1: *Example of an MADF graph (G_1).*



(a) CSDF graph G_1^1 of mode SI^1.
(b) CSDF graph G_1^2 of mode SI^2.

Figure 2.2: *Two modes of the MADF graph in Figure 2.1.*

2.1.2 Mode-Aware Data Flow (MADF)

MADF [94] is an adaptive MoC which can capture multiple application modes associated with an adaptive streaming application, where each individual mode is represented as a CSDF graph [16]. Formally, an MADF is a multigraph defined by a tuple $(\mathcal{A}, A_c, \mathcal{E}, P)$, where $\mathcal{A}$ is a set of dataflow actors, A_c is the control actor to determine modes and their transitions, $\mathcal{E}$ is the set of edges for data/parameter transfer, and $P = \{\vec{p}_1, \vec{p}_2, \cdots, \vec{p}_{|\mathcal{A}|}\}$ is the set of parameter vectors, where each $\vec{p}_i \in P$ is associated with a dataflow actor $A_i \in \mathcal{A}$. The detailed formal definitions of all components of the MADF MoC can be found in [94].

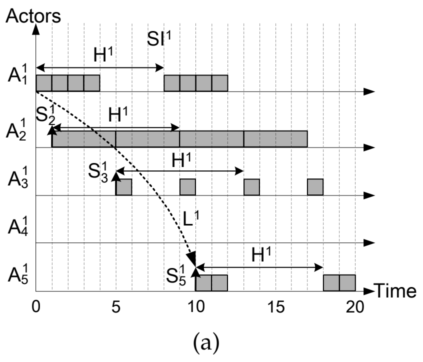
Here, we explain the MADF intuitively by an example. The MADF graph G_1 of an adaptive streaming application with two different modes is shown in Figure 2.1. This graph consists of a set of five actors A_1 to A_5 that communicate data over FIFO channels, i.e., the edges E_1 to E_5. Also, there is an extra actor A_c which controls the switching between modes through control FIFO channels, i.e., the edges E_{11}, E_{22}, E_{33} E_{44}, and E_{55}, at run-time. Each data FIFO channel contains a production and a consumption pattern, and some of these production and consumption patterns are parameterized. Having different values of parameters and WCET of the actors determine different modes. For example, to specify the consumption pattern with variable length on a data FIFO channel in graph G_1, the parameterized notation $[a[b]]$ is used to represent a sequence of a elements with integer value b, e.g., $[2[1]] = [1, 1]$ and $[1[2]] = [2]$. For the MADF example in Figure 2.1, $P = \{\vec{p}_1 = [p_1], \vec{p}_2 = [p_2], \vec{p}_3 = [], \vec{p}_4 = [p_4], \vec{p}_5 = [p_5, p_6]\}$. Now let assume that the parameter vector $[p_1, p_2, p_4, p_5, p_6]$ can take only two values $[0, 2, 0, 2, 0]$ and $[1, 1, 1, 1, 1]$. Then, A_c can switch the application between two corresponding modes SI^1 and SI^2 by setting the parameter vector to the first value and the second value, respectively, at run-time. Figure 2.2(a) and Figure 2.2(b) show the corresponding CSDF graphs of modes SI^1 and SI^2.

While the operational semantics of an MADF graph [94] in steady-state, i.e., when the graph is executed in each individual mode, are the same as that of a CSDF graph [16], the transition of MADF graph from one mode to another is the crucial part that makes MADF fundamentally different from CSDF. The protocol for mode transitions has a strong impact on the design-time analyzability and implementation efficiency, discussed in Section 1.2.2. In the existing adaptive MoCs like FSM-SADF [32], a protocol, referred as self-timed transition protocol, has been adopted which specifies that tasks are scheduled as soon as possible during mode transitions. This protocol, however, introduces timing interference of one mode execution with another one that can significantly affect and fluctuate the latency of an adaptive streaming application across a long sequence of mode transitions. To avoid such undesirable behavior caused by the self-timed transition protocol, MADF employs a simple, yet effective transition protocol, namely the maximum-overlap offset (MOO) transition protocol [94] when switching an application's mode by receiving a mode change request (MCR) from the external environment via the IC port of actor A_c (see the black dot in Figure 2.1). The MOO protocol can resolve the timing interference between modes upon mode transitions by properly offsetting the starting time of the new mode by $x^{o \to n}$ computed as follows:

$$x^{o \to n} = \begin{cases} \max_{A_i \in \mathcal{A}^o \cap \mathcal{A}^n} (S_i^o - S_i^n) & \text{if } \max_{A_i \in \mathcal{A}^o \cap \mathcal{A}^n} (S_i^o - S_i^n) > 0 \\ 0 & \text{otherwise,} \end{cases} \tag{2.4}$$

where S_i^o and S_i^n are the start times of actor A_i in mode SI^o and SI^n, i.e., the current and the new mode, respectively.

For instance, consider the valid schedules of modes SI^1 and SI^2 shown in Figure 2.3(a) and (b), respectively. In these figures, H is the *iteration period*, also called *hyper period*, that represents the duration needed by the graph to complete one iteration and L is the *iteration latency* that represents the time



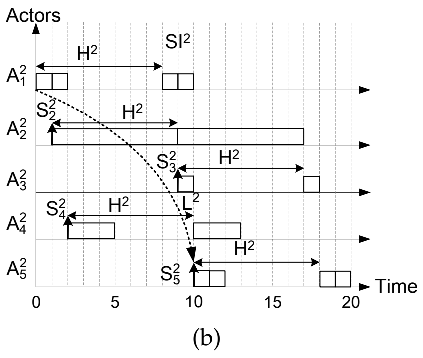


Figure 2.3: *Execution of two iterations of both modes SI^1 and SI^2. (a) Mode SI^1 in Figure 2.2(a). (b) Mode SI^2 in Figure 2.2(b).*

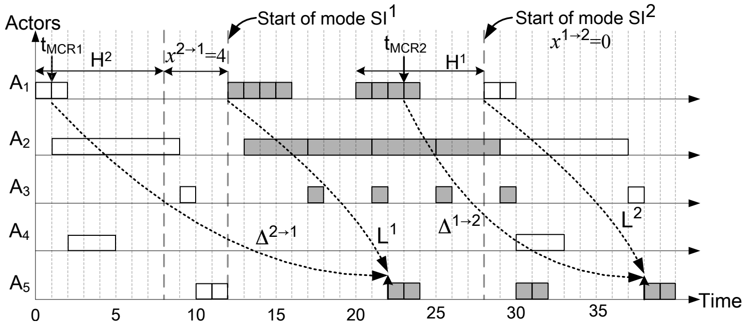


Figure 2.4: *Execution of graph G_1 with two mode transitions under the MOO protocol.*

distance between the starting times of the input actor and the output actor. Then, the offset $x^{1\to2}$ for the mode transition from SI^1 to SI^2 is computed by the following equations: $S_1^1 - S_1^2 = 0 - 0 = 0, S_2^1 - S_2^2 = 1 - 1 = 0, S_3^1 - S_3^2 = 5 - 9 = -4, S_5^1 - S_5^2 = 10 - 10 = 0$, and is $\max(0, 0, -4, 0) = 0$. Similarly, the offset $x^{2\to1}$ for the mode transition from SI^2 to SI^1, using the equations $S_1^2 - S_1^1 = 0, S_2^2 - S_2^1 = 0, S_3^2 - S_3^1 = 4, S_5^2 - S_5^1 = 0$, is $\max(0, 0, 4, 0) = 4$. An execution of G_1 with the two mode transitions and the computed offsets is illustrated in Figure 2.4, in which, the iteration latency L of the schedule of the modes, in Figure 2.3(a) and (b), are preserved during mode transitions.

To quantify the responsiveness of a transition protocol, a metric, called *transition delay* and denoted by $\Delta^{o\to n}$, is also introduced in [94] and calculated as

$$\Delta^{o\to n} = \sigma_{out}^{o\to n} - t_{\mathrm{MCR}} \tag{2.5}$$

where $\sigma_{out}^{o\to n}$ is the earliest start time of the output actor in the new mode

SI^n and t_{MCR} is the time when the mode change request MCR occurred. In Figure 2.4, we can compute the transition delay for MCR1 occurred at time $t_{\mathrm{MCR1}} = 1$ as $\Delta^{2\rightarrow 1} = 22 - 1 = 21$ time units.

2.2 Real-Time Scheduling Theory

In this section, we introduce the real-time periodic task model [29] and some important real-time scheduling concepts and algorithms [29] which are instrumental to the contributions we present in this thesis.

2.2.1 System Model

To present the important results from the real-time scheduling theory relevant to this thesis, we consider a *homogeneous* multiprocessor system composed of a set $\Pi = \{\pi_1, \pi_2, \cdots, \pi_m\}$ of m identical processors. However, the results of our research contributions, presented in this thesis, are applicable to heterogeneous multiprocessor systems as well. This is because the processor heterogeneity can be captured within the WCET of real-time periodic tasks, which will be explained in Chapter 4.

2.2.2 Real-Time Periodic Task Model

Under the real-time periodic task model, applications running on a system are modeled as a set $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$ of n periodic tasks, that can be preempted at any time. Every periodic task $\tau_i \in \Gamma$ is represented by a tuple $\tau_i = (C_i, T_i, S_i, D_i)$, where C_i is the WCET of the task, T_i is the period of the task in relative time units, S_i is the start time of the task in absolute time units, and D_i is the deadline of the task in relative time units. The task τ_i is said to be a *constrained-deadline periodic* (CDP) task if $D_i \leq T_i$. When $D_i = T_i$, the task τ_i is said to be an *implicit-deadline periodic* (IDP) task. Each task τ_i executes periodically in a sequence of task invocations. Each task invocation releases a *job*. The $k-$th job of task τ_i, denoted as $\tau_{i,k}$, is released at time instant $s_{i,k} = S_i + kT_i, \forall k \in \mathbb{N}_0$ and executed for at most C_i time units before reaching its deadline at time instant $d_{i,k} = S_i + kT_i + D_i$.

The **utilization** of task τ_i, denoted as u_i, is defined as $u_i = C_i/T_i$, where $u_i \in (0, 1]$. For a task set Γ, u_Γ is the total utilization of Γ given by $u_\Gamma = \sum_{\tau_i \in \Gamma} u_i$. Similarly, the **density** of task τ_i is $\delta_i = C_i/D_i$ and the total density of Γ is $\delta_\Gamma = \sum_{\tau_i \in \Gamma} \delta_i$.

2.2.3 Real-Time Scheduling Algorithms

When a multiprocessor system Π and a set of real-time period tasks Γ are given, a real-time scheduling algorithm is needed to execute the tasks on the system such that all task deadlines are always met. According to [29], real-time scheduling algorithms for multiprocessor systems try to solve the following two problems:

- The *allocation problem*, that is, on which processor(s) jobs of tasks should execute.
- The *priority assignment problem*, that is, when and in what order each job of a task with respect to jobs of other tasks should execute.

Depending on how the scheduling algorithms solve the allocation problem, they can be classified as follows [29]:

- *No migration*: each task is statically allocated on a processor and no migration is allowed.
- *Task-level migration*: jobs of a task can execute on different processors. However, each job can only execute on one processor.
- *Job-level migration*: jobs of a task can migrate and execute on different processors. However, each job cannot execute on more than one processor at the same time.

A scheduling algorithm that allows migration, either at task-level or job-level, among all processors is called a **global** scheduling algorithm, while an algorithm that does not allow migration at all is called a **partitioned** scheduling algorithm. Finally, an algorithm that allows migration, either at task-level or job-level, only for a subset of tasks among a subset of processors is called a **hybrid** scheduling algorithm.

Depending on how the scheduling algorithms solve the priority assignment problem, they can be classified as follows [29]:

- *Fixed task priority*: each task has a single fixed priority that is used for all its jobs.
- *Fixed job priority*: jobs of a task may have different priorities. However, each job has only a single fixed priority.
- *Dynamic priority*: a single job of a task may have different priorities at different times during its execution.

The scheduling algorithms can be further classified into [29]:

- *Preemptive*: tasks can be preempted by a higher priority task at any time.
- *Non-preemptive*: once a task starts executing, it will not be preempted and it will execute until completion.

A task set Γ is said to be **feasible** with respect to a given system Π if there exists a scheduling algorithm that can construct a schedule in which all task deadlines are always met. A scheduling algorithm is said to be **optimal** with respect to a task model and a system, if it can schedule all task sets that comply with the task model and are feasible on the system. A task set is said to be **schedulable** on a system under a given scheduling algorithm, if all tasks can execute under the scheduling algorithm on the system without violating any deadline. To check whether a task set is schedulable on a system under a given scheduling algorithm, the real-time scheduling theory provides various analytical **schedulability tests**. Generally, schedulability tests can be classified as follows [29]:

- *Sufficient*: if all task sets that are deemed schedulable by a schedulability test are in fact schedulable.
- *Necessary*: if all task sets that are deemed unschedulable by a schedulability test are in fact unschedulable.
- *Exact*: if a schedulability test is both sufficient and necessary.

Uniprocessor Schedulability Analysis

In this thesis, we use the *preemptive* earliest deadline first (EDF) scheduling algorithm [54], which is the most studied and popular *dynamic-priority* scheduling algorithm on uniprocessor systems, as the basis scheduling algorithm. The EDF algorithm schedules jobs of tasks according to their absolute deadlines. More specifically, jobs of tasks with earlier deadlines will be executed at higher priorities [21]. The EDF algorithm has been proven to be the optimal scheduling algorithm for periodic tasks on uniprocessor systems [21, 54]. An *exact* schedulability test for an implicit-deadline periodic task set on a uniprocessor system under EDF is given in the following theorem.

Theorem 2.2.1 (From [54]). *Under EDF, an implicit-deadline periodic task set Γ is schedulable on a uniprocessor system if and only if:*

$$u_\Gamma = \sum_{\tau_i \in \Gamma} u_{\tau_i} \leq 1. \tag{2.6}$$

For a *constrained-deadline* periodic task set, however, Equation (2.6) serves as a *necessary* test. An *exact* schedulability test for a constrained-deadline periodic task set on a uniprocessor under EDF is given in the following lemma.

Lemma 2.2.1 (From [13]). *Under EDF, a periodic task set Γ is schedulable on a uniprocessor system if and only if $u_\Gamma \leq 1$ and $dbf(\Gamma, t_1, t_2) \leq (t_2 - t_1)$ for all*

$0 \le t_1 < t_2 < \hat{S} + 2H$, where $dbf(\Gamma, t_1, t_2)$, termed as ***processor demand bound function****, denotes the total execution time that all tasks of Γ demand within time interval $[t_1, t_2]$ and is given by*

$$dbf(\Gamma, t_1, t_2) = \sum_{\tau_i \in \Gamma} \max\{0, \left\lfloor \frac{t_2 - S_i - D_i}{T_i} \right\rfloor - \max\{0, \left\lceil \frac{t_1 - S_i}{T_i} \right\rceil\} + 1\} \cdot C_i,$$

$\hat{S} = \max\{S_1, S_2, \cdots, S_{|\Gamma|}\}$, and $H = \mathrm{lcm}\{T_1, T_2, \cdots, T_{|\Gamma|}\}$.

However, this schedulability test is computationally expensive because it needs to check all absolute deadlines, which can be a large number, within the time interval. To improve the efficiency of the EDF exact test, a new exact test for the EDF scheduling is proposed in [95] which checks a smaller number of time points within the time interval.

Multiprocessor Schedulability Analysis

On multiprocessor systems, there are several *optimal* global scheduling algorithms for implicit-deadline periodic tasks, such as Pfair [12] and LLREF [27], which exploit job-level migrations and dynamic priority. Under these scheduling algorithms, an exact schedulability test for an implicit-deadline periodic task set Γ on m processors is:

$$u_\Gamma = \sum_{\tau_i \in \Gamma} u_{\tau_i} \le m. \tag{2.7}$$

Based on the above equation, the absolute minimum number of processors, denoted as $\check{m}_{\mathrm{OPT}}$, needed by an optimal scheduling algorithm to schedule an implicit-deadline periodic task set Γ is:

$$\check{m}_{\mathrm{OPT}} = \lceil u_\Gamma \rceil. \tag{2.8}$$

In the case of constrained-deadline periodic tasks, however, no optimal algorithm for global scheduling exists [29]. Under global dynamic priority schedulings, a *sufficient* schedulability test for a constrained-deadline periodic task set Γ on m processors is [6,31]:

$$\delta_\Gamma = \sum_{\tau_i \in \Gamma} \delta_{\tau_i} \le m. \tag{2.9}$$

According to this test, the minimum number of processors needed by a global dynamic priority scheduling to schedule a constrained-deadline periodic task set Γ is:

$$\check{m} = \lceil \delta_\Gamma \rceil. \tag{2.10}$$

The other class of multiprocessor scheduling algorithms for periodic task sets are partitioned scheduling algorithms [29] that do not allow task migration. Under partitioned scheduling algorithms, a task set is first partitioned into subsets (according to Definition 2.2.1) that will be executed statically on individual processors. Then, the tasks on each processor are scheduled using a given uniprocessor scheduling algorithm.

Definition 2.2.1. *(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

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

In this regard, the minimum number of processors needed to schedule a task set Γ by a partitioned scheduling algorithm is:

$$ \check{m}_{\text{PAR}} = \min\{ x \in \mathbb{N} \mid \exists x\text{-partition of } \Gamma \wedge \forall i \in [1, x] : {}^{x}\Gamma_i \text{ is schedulable on } \pi_i \}. \tag{2.11} $$

The derived x-partition of a task set, using Equation (2.11), is optimal because of requiring the least amount of processors to allocate all tasks while guaranteeing schedulability on all processors. Deriving such optimal partitioning is inherently equivalent to the well-known *bin packing* problem [45]. In the bin packing problem, items of different sizes must be packed into bins with fixed capacity such that the number of needed bins is minimized. However, finding an optimal solution for the bin packing problem is known to be NP-hard [46]. Therefore, several heuristic algorithms have been developed to solve the bin packing problem and obtain approximate solutions in a reasonable time interval. Below, we introduce the most commonly used heuristics [28,46].

- **First-Fit (FF)** algorithm: places an item to the first (i.e., lowest index) bin that can accommodate the item. If no such bin exists, a new bin is opened and the item is placed on it.
- **Best-Fit (BF)** algorithm: places an item to a bin that can accommodate the item and has the minimal remaining capacity after placing the item. If no such bin exists, a new bin is opened and the item is placed on it.
- **Worst-Fit (WF)** algorithm: places an item to a bin that can accommodate the item and has the maximal remaining capacity after placing the item. If no such bin exists, a new bin is opened and the item is placed on it.

The performance of these heuristic algorithms can be improved by sorting the items according to a certain criteria, such as their size. Then, we obtain the **First-Fit Decreasing (FFD)**, **Best-Fit Decreasing (BFD)**, and **Worst-Fit Decreasing (WFD)** heuristics.

2.3 HRT Scheduling of Acyclic CSDF Graphs

As mentioned in Section 1.3, recently, a scheduling framework, namely, the Strictly Periodic Scheduling (SPS) framework, has been proposed in [8] which enables the utilization of many scheduling algorithms from the classical hard real-time scheduling theory (briefly introduced in Section 2.2) to applications modeled as acyclic CSDF graphs. The main advantages of these scheduling algorithms are that they provide: 1) temporal isolation and 2) fast, yet accurate calculation of the minimum number of processors that guarantee the required performance of an application and mapping of the application's tasks on processors. The basic idea behind the SPS framework is to convert a set $\mathcal{A} = \{A_1, A_2, \cdots, A_n\}$ of n actors of a given CSDF graph to a set $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$ of n real-time implicit-deadline periodic tasks[1]. In particular, for each actor $A_j \in \mathcal{A}$ of the CSDF graph, the SPS framework derives the parameters, i.e., the period (T_j) and start time (S_j), of the corresponding real-time periodic task $\tau_j = (C_j, T_j, S_j, D_j = T_j) \in \Gamma$. The period T_i of task τ_j corresponding to actor A_j under the SPS framework can be computed as:

$$T_j = \frac{\mathrm{lcm}(\vec{q})}{q_j} \cdot s, \tag{2.12}$$

$$s \geq \check{s} = \left\lceil \frac{\hat{W}}{\mathrm{lcm}(\vec{q})} \right\rceil \in \mathbb{N}, \tag{2.13}$$

where $\mathrm{lcm}(\vec{q})$ is the least common multiple of all repetition entries in $\vec{q}$ (explained in Section 2.1.1), $\hat{W} = \max_{A_j \in \mathcal{A}}\{C_j \cdot q_j\}$ is the maximum actor workload of the CSDF graph, and $C_j = \max_{1 \leq \phi \leq \phi_j}\{C_j(\phi)\}$, where $C_j(\phi)$ includes both the worst-case computation time and worst-case data communication time required by a phase ϕ of actor A_j. *Note that $C_j(\phi)$ includes the worst-case data communication time in order to ensure the feasibility of the derived schedule regardless of the variance of different task allocations.* In general, the derived period vector $\vec{T}$ satisfies the condition:

$$q_1 T_1 = q_2 T_2 = \cdots = q_n T_n = H \tag{2.14}$$

[1]Throughout this thesis, we may use the terms *task* and *actor* interchangeably.

where H is the iteration period. Once the period of each task has been computed, the throughput $\mathcal{R}$ of the graph can be computed as:

$$\mathcal{R} = \frac{1}{T_{out}} \tag{2.15}$$

where T_{out} is the period of the task corresponding to output actor A_{out}. *Note that when the scaling factor $s = \check{s} = \lceil \hat{W} / \operatorname{lcm}(\vec{q}) \rceil$, the minimum period ($\check{T}_j$) is derived using Equation (2.12) which determines the maximum throughput achievable by the SPS framework.*

Then, to sustain the strictly periodic execution of the tasks corresponding to actors of the CSDF graph with the periods derived by Equation (2.12), the earliest start time S_j of each task τ_j corresponding to actor A_j, such that τ_j is never blocked on reading data tokens from any input FIFO channel connected to it during its periodic execution, is calculated using the following expression:

$$S_j = \begin{cases} 0 & if \;\; \operatorname{prec}(A_j) = \emptyset \\ \max_{A_i \in \operatorname{prec}(A_j)}(S_{i\to j}) & otherwise, \end{cases} \tag{2.16}$$

where $\operatorname{prec}(A_j)$ represents the set of predecessor actors of A_j and $S_{i\to j}$ is given by:

$$\begin{aligned} S_{i\to j} = \min_{t \in [0, S_i + H]} \Big\{ t : & \operatorname*{Prd}_{[S_i, \max\{S_i, t\}+k)}(A_i, E_u) \\ & \geq \operatorname*{Cns}_{[t, \max\{S_i, t\}+k]}(A_j, E_u),\ \forall k \in [0, H], k \in \mathbb{N} \Big\} \end{aligned} \tag{2.17}$$

where $\operatorname{Prd}_{[t_s, t_e)}(A_i, E_u)$ is the total number of tokens produced by a predecessor actor A_i to channel E_u during the time interval $[t_s, t_e)$ with the assumption that token production happens as **late** as possible at the deadline of each invocation of actor A_i, $\operatorname{Cns}_{[t_s, t_e]}(A_j, E_u)$ is the total number of tokens consumed by actor A_j from channel E_u during the time interval $[t_s, t_e]$ with the assumption that token consumption happens as **early** as possible at the release time of each invocation of actor A_j, and S_i is the earliest start time of actor A_i.

The authors in [8] also provide a method to calculate the minimum buffer size needed for each FIFO communication channel and the latency of the CSDF graph scheduled in a strictly periodic fashion. In this framework, once the start time of each task has been calculated, the minimum buffer size of each FIFO communication channel $E_u = (A_i, A_j) \in \mathcal{E}$, denoted with b_u, is calculated as follows:

$$b_u = \max_{k \in [0, H]} \Big\{ \operatorname*{Prd}_{[S_i, \max(S_i, S_j)+k)}(A_i, E_u) - \operatorname*{Cns}_{[S_j, \max(S_i, S_j)+k)}(A_j, E_u) \Big\} \tag{2.18}$$

with the assumption that token production happens as **early** as possible at the release time of each invocation of actor A_i and token consumption happens as **late** as possible at the deadline of each invocation of actor A_j. Indeed, b_u is the maximum number of unconsumed data tokens in channel E_u during the execution of A_i and A_j in one graph iteration period. Finally, the latency $\mathcal{L}$ of the graph can be calculated as follows:

$$\mathcal{L} = \max_{w \in W}(S_{out} + g^{C}_{out}T_{out} + D_{out} - (S_{in} + g^{P}_{in}T_{in})) \tag{2.19}$$

where w is one path of set W which includes all paths in the CSDF graph from the input actor to the output actor, S_{in} and S_{out} are the earliest start times of the tasks corresponding to the input and output actors, respectively, T_{in} and T_{out} are the periods of the tasks corresponding to the input and output actors, respectively, D_{out} is the deadline of the task corresponding to the output actor, and g^{C}_{out} and g^{P}_{in} are two constants which denote the number of invocations the actor waits for the non-zero production/consumption on/from a path $w \in W$.

2.4 HRT Scheduling of MADF Graphs

Based on the proposed MOO protocol for mode transitions, briefly described in Section 2.1.2, a hard real-time analysis and scheduling framework for the MADF MoC is proposed in [94] which is an extension of the SPS framework, briefly described in Section 2.3, developed for CSDF graphs. As explained in Section 2.3, the key concept of the SPS framework is to derive a periodic task set representation for a CSDF graph. Since an MADF graph in steady-state can be considered as a CSDF graph, it is thus straightforward to represent the steady-state of an MADF graph as a periodic task set (see Section 2.3) and schedule the resulting task set using any well-known hard real-time scheduling algorithm.

Using the SPS framework, we can derive the two main parameters for each task τ_i^o corresponding to an MADF actor A_i in mode SI^o, namely the period (T_i^o using Equation (2.12)) and the earliest start time (S_i^o using Equation (2.16)). Then, the offset $x^{o \to n}$ for mode transition of the MADF graph from mode SI^o to mode SI^n can be simply computed using Equation (2.4). For instance, by applying the SPS framework for graphs G_1^1 and G_1^2, shown in Figure 2.2(a) and 2.2(b), corresponding to modes SI^1 and SI^2 of graph G_1 shown in Figure 2.1, the task set $\Gamma_1^1 = \{\tau_1^1 = (C_1^1 = 1, T_1^1 = 2, S_1^1 = 0, D_1^1 = T_1^1 = 2), \tau_2^1 = (4,4,2,4), \tau_3^1 = (1,4,6,4), \tau_5^1 = (1,4,14,4)\}$ of four IDP tasks and the task set $\Gamma_1^2 = \{\tau_1^2 = (C_1^2 = 1, T_1^2 = 4, S_1^2 = 0, D_1^2 = T_1^2 = 4), \tau_2^2 = (8,8,4,8), \tau_3^2 =$

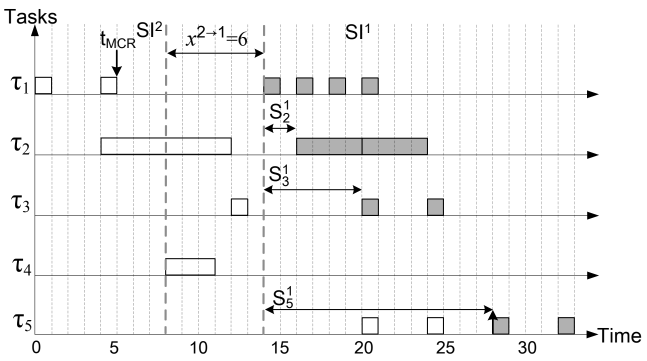


Figure 2.5: *Execution of graph G_1 with a mode transition from mode SI^2 to mode SI^1 under the MOO protocol and the SPS framework.*

$(1,8,12,8), \tau_4^2 = (3,8,8,8), \tau_5^2 = (1,4,20,4)\}$ of five IDP tasks can be derived, respectively. An execution of graph G_1 with a mode transition from mode SI^2 to mode SI^1, using the derived task sets Γ_1^1 and Γ_1^2, is shown in Figure 2.5, where the offset $x^{2\to1}$ is computed by the following equations (see Equation (2.4)): $S_1^2 - S_1^1 = 0 - 0 = 0$, $S_2^2 - S_2^1 = 4 - 2 = 2$, $S_3^2 - S_3^1 = 12 - 6 = 6$, $S_5^2 - S_5^1 = 20 - 14 = 6$, and is $\max(0,2,6,6) = 6$. However, this offset is only the lower bound because the task allocation on processors is not yet taken into account. This means, the execution of tasks using the schedule, shown in Figure 2.5, is valid when each task is allocated on a separate processor.

In a system where multiple tasks are allocated on the same processor, the processor may be potentially overloaded during mode transitions due to the presence of executing tasks in both modes. To avoid overloading of processors, a larger offset may be needed to delay the start time of tasks in the new mode. In [94], this offset, referred as $\delta^{o\to n}$, is calculated as follows:

$$\delta^{o\to n} = \min_{t\in[x^{o\to n}, S_{\text{out}}^o]} \{t : u_{\pi_j}(k) \leq UB,\ \forall k \in [t, S_{\text{out}}^o] \wedge \forall \pi_j \in \Pi\}. \tag{2.20}$$

This equation simply tests all time instants when tasks in both modes SI^o and SI^n are present in the system and checks whether the processors are consequently overloaded or not. If yes, the starting time of the new mode SI^n, which already was delayed by $x^{o\to n}$, is further delayed to $\delta^{o\to n}$. Thus, $\delta^{o\to n}$ of interest for the mode transition from mode SI^o to mode SI^n is the minimum time t in the bounded interval $[x^{o\to n}, S_{\text{out}}^o]$ such that the total utilization does not exceed the utilization bound (UB), e.g., 1 for EDF, for all remaining time instants in the interval. To compute the total utilization of all tasks allocated on processor π_j in any time instant k, the following equation is used in [94].

$$u_{\pi_j}(k) = \underbrace{\sum_{\tau_i^o \in {}^x\Gamma_j} \left(u_i^o - h(k - S_i^o) \cdot u_i^o\right)}_{u_{\pi_j}^o(k)} + \underbrace{\sum_{\tau_i^n \in {}^x\Gamma_j} \left(h(k - S_i^n - t) \cdot u_i^n\right)}_{u_{\pi_j}^n(k)} \tag{2.21}$$

In this equation, the terms denoted by $u_{\pi_j}^o(k)$ and $u_{\pi_j}^n(k)$ refers to the total utilization of tasks that are allocated on processor π_j and are executing in the current mode SI^o and the new mode SI^n, respectively, at time instant k. h(t) is the Heaviside step function.

For instance, consider the execution of the tasks in the schedule, shown in Figure 2.5, on platform $\Pi = \{\pi_1, \pi_2\}$ with two processors and the tasks allocation ${}^2\Gamma = \{{}^2\Gamma_1 = \{\tau_1, \tau_3, \tau_4, \tau_5\}, {}^2\Gamma_2 = \{\tau_2\}\}$. In this schedule, the earliest start time of the new mode SI^1 is at time instant 14 corresponding to $\delta^{2\to 1} = x^{2\to 1} = 6$. Then, the total utilization of processor π_1 demanded by the tasks in the old mode SI^2 at time instant 14, i.e., $u_{\pi_1}^2(6)$, can be computed as follows using Equation (2.21):

$$\begin{aligned}
u_{\pi_1}^2(6) &= \sum_{\tau_i^2 \in {}^2\Gamma_1} u_i^2 - h(6 - S_i^2) \cdot u_i^2,\ i \in \{1,3,4,5\} \\
&= u_1^2 - h(6) \cdot u_1^2 + u_3^2 - h(-6) \cdot u_3^2 + u_4^2 - h(-2) \cdot u_4^2 + u_5^2 - h(-14) \cdot u_5^2 \\
&= 0 + u_3^2 + u_4^2 + u_5^2 = \frac{1}{8} + \frac{3}{8} + \frac{1}{4} = \frac{3}{4}.
\end{aligned}$$

Now, releasing task τ_1^1 in the new mode SI^1 at time 14 would yield

$$u_{\pi_1}(6) = u_{\pi_1}^2(6) + u_1^1 = \frac{3}{4} + \frac{1}{2} > UB = 1,$$

thereby leading to being unschedulable on processor π_1. In this case, the earliest start times of the new mode SI^1 must be delayed by $\delta^{2\to 1} = 8$ time units to time instant 16 as shown in Figure 2.6. At time instant 16, the total utilization of processor π_1 demanded by the tasks in the old mode SI^2 is

$$\begin{aligned}
u_{\pi_1}^2(8) &= \sum_{\tau_i^2 \in {}^2\Gamma_1} u_i^2 - h(8 - S_i^2) \cdot u_i^2,\ i \in \{1,3,4,5\} \\
&= u_1^2 - h(8) \cdot u_1^2 + u_3^2 - h(-4) \cdot u_3^2 + u_4^2 - h(0) \cdot u_4^2 + u_5^2 - h(-12) \cdot u_5^2 \\
&= 0 + u_3^2 + 0 + u_5^2 = \frac{1}{8} + \frac{1}{4} = \frac{3}{8}.
\end{aligned}$$

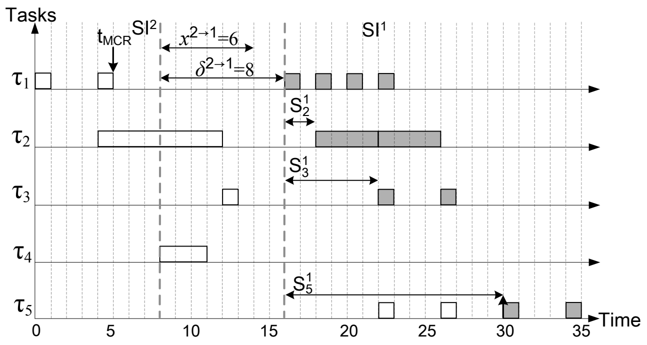


Figure 2.6: *Execution of graph G_1 with a mode transition from mode SI^2 to mode SI^1 under the MOO protocol and the SPS framework with task allocation on two processors.*

Now, releasing task τ_1^1 in the new mode SI^1 at time instant 16 results in the total utilization of processor π_1 as

$$u_{\pi_1}(8) = u_{\pi_1}^2(8) + u_1^1 = \frac{3}{8} + \frac{1}{2} < 1.$$

Next, assuming that the new mode SI^1 starts at time instant 16, the above procedure should be repeated for the remaining tasks in the new mode SI^1, namely τ_3^1 and τ_5^1, to ensure that they can start execution with S_3^1 and S_5^1, respectively, without overloading processor π_1. Then, if processor π_1 is overloaded again, a larger offset $\delta^{2\to1}$ is needed that can be calculated using Equation (2.20).