Chapter 2

Background

THIS chapter introduces the background necessary to understand the contribution of this thesis presented in the following chapters. First, we give in Table 2.1 a summary of the mathematical notations used throughout the thesis. Then, we present the dataflow models considered in this thesis in Section 2.1, while some results from the hard real-time scheduling theory relevant for this thesis are presented in Section 2.2.

Table 2.1: *Summary of mathematical notations*

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 (size) of a set x
$\hat{x}$	The maximum value of x
$\check{x}$	The minimum value of x
lcm	The least common multiple operator
mod	The integer modulo operator

2.1 Dataflow Models-of-Computations (MoCs)

As mentioned earlier in Section 1.1.2, dataflow MoCs have been used to efficiently express 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 PPN MoC is given in Section 2.1.2. The CSDF MoC is used to specify streaming applications within the hard real-time scheduling framework proposed in Chapter 3. The SDF MoC is used to specify the input streaming applications in the techniques which exploit parallelism in streaming applications to maximize the resource utilization and minimize the energy consumption, presented in Chapters 4 and 5, respectively. The PPN MoC is used to specify streaming applications within the solution for highly accurate modeling of energy consumption, presented in Chapter 6.

2.1.1 Cyclo-Static Dataflow (CSDF)

An application modeled as a CSDF [BELP96] is a directed graph $G = (\mathcal{V}, \mathcal{E})$ that consists of a set of actors $\mathcal{V}$ which communicate with each other through a set of communication channels $\mathcal{E}$. Actors represent a certain functionality of the application, while communication channels are first-in first-out (FIFO) buffers representing data dependencies and transferring *data tokens* between the actors. A data token is an atomic data object belonging to a stream of data transferred between the actors. We can associate each actor $v_i \in \mathcal{V}$ in a graph with two sets of actors, the *predecessors set*, denoted by prec(v_i), and the *successors set*, denoted by succ(v_i). These sets are given by:

$$\text{prec}(v_i) = \{v_j \in \mathcal{V} : \exists e_u = (v_j, v_i) \in \mathcal{E}\} \tag{2.1}$$

$$\text{succ}(v_i) = \{v_j \in \mathcal{V} : \exists e_u = (v_i, v_j) \in \mathcal{E}\} \tag{2.2}$$

In addition, we can define for each actor $v_i \in \mathcal{V}$ in a graph two sets of communication channels, the *input set*, denoted by inp(v_i), and the *output set*, denoted by out(v_i). The *input set* contains all the input channels to v_i, while the *output set* contains all the output channels from v_i. If an actor v_i receives an input data stream from the environment then v_i is called *input actor*, and v_i does not have input channels, that is, inp(v_i)$= \emptyset$. Similarly, if an actor v_i produces an output data stream for the environment then v_i is called *output actor*, and v_i does not have output channels, that is, out(v_i)$= \emptyset$. A *path* $w_{i \to j}$ between actors v_i and v_j is an ordered sequence of channels connecting v_i and v_j denoted as $w_{i \to j} = \{(v_i, v_k), (v_k, v_l), \cdots, (v_m, v_j)\}$.

Every actor $v_i \in \mathcal{V}$ has an *execution sequence* $[F_i(1), F_i(2), \cdots, F_i(\phi_i)]$ of length ϕ_i, that is, it has ϕ_i phases. The kth time that actor v_i is fired, it executes the function $F_i(((k-1) \bmod \phi_i) + 1)$. As a consequence, the execution time of actor v_i is also a sequence $[C_i^C(1), C_i^C(2), \cdots, C_i^C(\phi_i)]$ consisting of the worst-case computation time values for each phase. Similarly, every output channel e_u of an actor v_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 on every input channel e_u of an actor v_i is a predefined se-

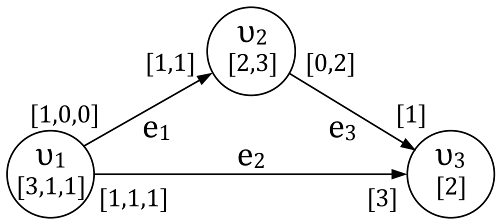


Figure 2.1: *A CSDF graph G.*

quence $[y_i^u(1), y_i^u(2), \cdots, y_i^u(\phi_i)]$, called *consumption sequence*. The total number of tokens on a channel e_u produced by v_i during its first n invocations and the total number of tokens consumed on the same channel by v_j during its first n invocations are $X_i^u(n) = \sum_{l=1}^{n} x_i^u(((l-1) \bmod \phi_i) + 1)$ and $Y_j^u(n) = \sum_{l=1}^{n} y_j^u(((l-1) \bmod \phi_j) + 1)$, respectively.

Figure 2.1 shows an example of a CSDF graph. For instance, actor v_1 has 3 phases, that is, $\phi_1 = 3$, its execution time sequence (in time units) is $[C_1^C(1), C_1^C(2), C_1^C(3)] = [3,1,1]$ and its token production sequence on channel e_1 is $[1,0,0]$.

An acyclic CSDF graph can be partitioned into a number of *levels*, denoted by L, in a way similar to topological sort. In that way, all input actors belong to level-1, the actors from level-2 have all immediate predecessors in level-1, the actors from level-3 have immediate predecessors in level-2 and can also have immediate predecessors in level-1, and so on.

An important property of the CSDF model is the ability to derive, at design time, a schedule for the actors. 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 [BELP96]). *In a CSDF graph G, a repetition vector $\vec{q} = [q_1, q_2, \cdots, q_N]^T$ is given by*

$$\vec{q} = \mathbf{\Phi} \cdot \vec{r}, \qquad \textit{with} \qquad \Phi_{jk} = \begin{cases} \phi_j & \textit{if } j = k \\ 0 & \textit{otherwise} \end{cases} \tag{2.3}$$

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

$$\mathbf{\Gamma} \cdot \vec{r} = \vec{0} \tag{2.4}$$

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

$$\Gamma_{uj} = \begin{cases} X_j^u(\phi_j) & \textit{if actor } v_j \textit{ produces on channel } e_u \\ -Y_j^u(\phi_j) & \textit{if actor } v_j \textit{ consumes from channel } e_u \\ 0 & \textit{otherwise.} \end{cases} \tag{2.5}$$

A CSDF graph G is said to be consistent if a positive integer solution $\vec{r} = [r_1, r_2, \cdots, r_N]^T$ exists for the balance equation, Equation (2.4). We call $\vec{r}$ *aggregated repetition vector*. The smallest non-trivial aggregated repetition vector $\vec{r}$ is called *basic aggregated repetition vector* $\vec{r}$. Its corresponding repetition vector $\vec{q}$ is called *basic repetition vector* $\vec{q}$. If a deadlock-free schedule can be found, G is said to be live.

Definition 2.1.1. For a consistent and live CSDF graph G, an **actor iteration** is the invocation of an actor $v_i \in \mathcal{V}$ for q_i times, a **phase iteration** is the invocation of one phase of an actor $v_i \in \mathcal{V}$ for r_i times, and a **graph iteration** is the invocation of *every actor* $v_i \in \mathcal{V}$ for q_i times, where $q_i \in \vec{q}$, and *every phase* of *every actor* $v_i \in \mathcal{V}$ for r_i times, where $r_i \in \vec{r}$.

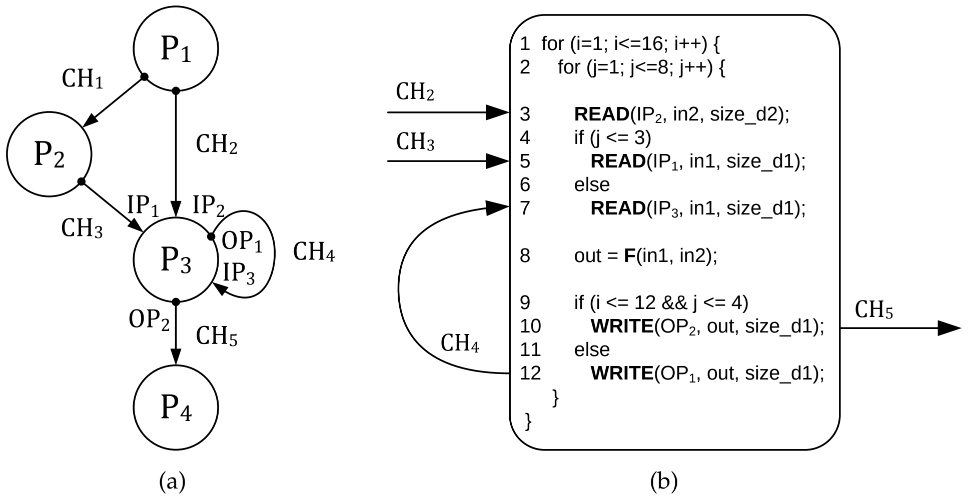
For the example CSDF graph G shown in Figure 2.1, we can compute the basic repetition vectors $\vec{r}$ and $\vec{q}$ by using the equations in Theorem 2.1.1, namely, Equations (2.3), (2.4) and (2.5), as follows:

$$\mathbf{\Gamma} = \begin{bmatrix} 1 & -2 & 0 \\ 3 & 0 & -3 \\ 0 & 2 & -1 \end{bmatrix}, \vec{r} = \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix}, \mathbf{\Phi} = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \text{and } \vec{q} = \begin{bmatrix} 6 \\ 2 \\ 2 \end{bmatrix}.$$

Two important subsets of the CSDF MoC are the Synchronous Data Flow (SDF) MoC [LM87] and the Homogeneous Synchronous Data Flow (HSDF) MoC [LM87]. All actors in an SDF graph $G = (\mathcal{V}, \mathcal{E})$ have only one phase, that is, for each $v_i \in \mathcal{V}, \phi_i = 1$. In an HSDF graph $G = (\mathcal{V}, \mathcal{E})$, in addition to $\forall v_i \in \mathcal{V}, \phi_i = 1$, all channels have production and consumption sequences equal to 1, that is, for each $e_u = (v_i, v_j) \in \mathcal{E}, x_i^u = [x_i^u(1)] = 1, y_j^u = [y_j^u(1)] = 1$.

2.1.2 Polyhedral Process Network (PPN)

An application modeled as a PPN [VNS07] is a directed graph $G = (\mathcal{P}, \mathcal{C})$ that consists of a set of processes $\mathcal{P}$, which communicate with each other via a set of communication channels $\mathcal{C}$. Processes in $\mathcal{P}$ represent tasks of an application. Channels in $\mathcal{C}$ are bounded FIFOs and represent one direction of data communication between two processes, that is, a channel $CH_l = (P_i, P_j)$ represents a data dependency between processes P_i and P_j, where P_i is the producer and P_j is the consumer process. An example of a PPN consisting of 4 processes which communicate with each other through 5 channels is given in Figure 2.2(a). Each PPN process has a set of *input ports* it reads from and a set of *output ports* it writes to. Process P_3 in the PPN example in Figure 2.2(a) has 3 input ports IP_1, IP_2, and IP_3, and 2 output ports OP_1 and OP_2. Channels of a



```
1  for (i=1; i<=16; i++) {
2    for (j=1; j<=8; j++) {

3      READ(IP2, in2, size_d2);
4      if (j <= 3)
5        READ(IP1, in1, size_d1);
6      else
7        READ(IP3, in1, size_d1);

8      out = F(in1, in2);

9      if (i <= 12 && j <= 4)
10       WRITE(OP2, out, size_d1);
11     else
12       WRITE(OP1, out, size_d1);
     }
   }
```

Figure 2.2: *Example of a PPN (a) and the structure of process P3 (b).*

process P_i connected to its input ports are *input channels* of P_i, while channels connected to the output ports of P_i are *output channels* of P_i.

The synchronization mechanism between the processes in the PPN MoC is *blocking read* from an empty FIFO and *blocking write* to a full FIFO. The execution of a PPN process is defined by using nested *for loops*, that is, the process execution is a set of iterations, called *process domain*. The process domain is represented using the polytope model [Fea96a]. Each PPN process has a precisely defined structure: the process reads data from a subset of its input ports depending on the values of loop iterators; then, it performs a computation on input data that generates output data; and finally, the process writes the output data through a subset of its output ports depending on the values of loop iterators.

Figure 2.2(b) shows the structure of process P_3 in the PPN example given in Figure 2.2(a). Process P_3 reads data from and writes data to channels through read and write primitives **READ**$(\cdots)$ and **WRITE**$(\cdots)$, respectively. The computation behavior of process P_3 is represented by a function $\mathbf{F}(\cdots)$ in Line 8 in Figure 2.2(b). The process domain of process P_3 is given as the polytope $D_{P_3} = \{(i,j) \in \mathbb{Z}^2 \mid 1 \leq i \leq 16 \wedge 1 \leq j \leq 8\}$. Accessing an input port of the PPN process is represented as a subset of the process domain, called *input port domain*. Similarly, accessing an output port of the PPN process is represented through *output port domain*. Process P_3 in Figure 2.2 reads data from input ports IP_1, IP_2 and IP_3. The input port domain of input port IP_2 is equal to process domain D_{P_3}, while the input port domain of port IP_1 is given as $D_{IP_1} = \{(i,j) \in \mathbb{Z}^2 \mid 1 \le i \le 16 \wedge 1 \le j \le 3\}$. Process P_3 writes data to output ports OP_1 and OP_2. Domain $D_{OP_2} = \{(i,j) \in \mathbb{Z}^2 \mid 1 \le i \le 12 \wedge 1 \le j \le 4\}$ is the output port domain of port OP_2.

2.2 Real-Time Scheduling Theory

In this section, we introduce the real-time periodic task model [DB11] and some important real-time scheduling concepts [DB11] instrumental to the approaches we present in Chapters 3, 4 and 5 of this thesis.

2.2.1 Task Model

The majority of the research on real-time scheduling considers a simple model to represent applications running on a hardware platform. In this simple model applications are modeled as a task set $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_n\}$ of n periodic tasks, which can be preempted at any time. A periodic task $\tau_i \in \mathcal{T}$ is defined by the 4-tuple $\tau_i = (S_i, C_i, D_i, T_i)$, where S_i is the start time of τ_i in absolute time units, C_i is the worst-case execution time (WCET), D_i is the deadline of τ_i in relative time units, and T_i is the task period in relative time units, where $C_i \le D_i \le T_i$. Each task τ_i executes periodically through a sequence of task invocations, that is, *job releases*, at $s_{i,k} = S_i + kT_i$, $k \in \mathbb{N}_0$. Once released, each job $\tau_{i,k}$, $k \in \mathbb{N}_0$, of a task τ_i must execute C_i time units before $s_{i,k} + D_i$, that is, the job must finish its execution before its deadline D_i. If $D_i = T_i$, then τ_i is said to have an *implicit-deadline*. Otherwise, if $D_i < T_i$, then τ_i is said to have a *constrained-deadline*. If all the tasks in a task set $\mathcal{T}$ are implicit-deadline periodic tasks, then task set $\mathcal{T}$ is an *implicit-deadline periodic (IDP) task set*. Otherwise, task set $\mathcal{T}$ is a *constrained-deadline periodic (CDP) task set*. Similarly, if all the tasks in a task set $\mathcal{T}$ have the same start time, then task set $\mathcal{T}$ is *synchronous*. Otherwise, task set $\mathcal{T}$ is *asynchronous*. In this thesis, we consider asynchronous task sets.

The **utilization** of task τ_i, denoted as u_i, where $u_i \in (0, 1]$, is defined as $u_i = C_i/T_i$. For a task set $\mathcal{T}$, $u_{\mathcal{T}}$ is the total utilization of $\mathcal{T}$ given by $u_{\mathcal{T}} = \sum_{\tau_i \in \mathcal{T}} u_i$. Similarly, the **density** of task τ_i is $\delta_i = C_i/D_i$ and the total density of $\mathcal{T}$ is $\delta_{\mathcal{T}} = \sum_{\tau_i \in \mathcal{T}} \delta_i$. The worst-case response time of task τ_i, denoted as R_i, is defined as the longest time interval from the arrival of a job of task τ_i to the completion of job's execution.

The *processor demand bound function* of a task set $\mathcal{T}$ over a time interval $[t_1, t_2]$ represents the maximum amount of task execution that can be released and completed in the time interval $[t_1, t_2]$, and is given by [BRH90]:

$$dbf(\mathcal{T}, t_1, t_2) = \sum_{\tau_i \in \mathcal{T}} \max\{0, \left\lfloor \frac{t_2 - S_i - D_i}{T_i} \right\rfloor - \max\{0, \left\lceil \frac{t_1 - S_i}{T_i} \right\rceil\} + 1\} \cdot C_i. \tag{2.6}$$

2.2.2 System Model

To present the important results from the real-time scheduling theory relevant for this thesis, we consider a system composed of a set $\Pi = \{\pi_1, \pi_2, \cdots, \pi_m\}$ of m identical processors. However, our contribution approaches, presented in this thesis, are applicable to both homogeneous and heterogeneous MPSoCs, because the processor heterogeneity is captured within the WCET of a task, which will be explained in more detail in Chapter 5. Thus, the results presented in the following section, Section 2.2.3, are applicable to heterogeneous MPSoCs as well.

2.2.3 Real-Time Scheduling Algorithms

In this section, we present some important scheduling concepts for scheduling applications modeled as real-time periodic tasks, introduced in Section 2.2.1, on a system modeled as described in Section 2.2.2.

Real-time scheduling algorithms for multiprocessors try to solve two problems [DB11]:

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

Depending on how they solve the *allocation problem*, scheduling algorithms are classified into:

- *No migration*. Each task is allocated to one processor and no migration is allowed.
- *Task-level migration*. The jobs of a task can execute on different processors. However, each job can execute only on one processor.
- *Job-level migration*. A job can migrate and execute on different processors. However, parallel execution of a job on processors is not allowed.

Scheduling algorithms that allow any job to migrate are called **global** algorithms. On the other hand, algorithms which do not allow migration are called **partitioned** algorithms. Finally, scheduling algorithms that allow migration of jobs released by a subset of tasks among a subset of processors are called **hybrid** algorithms.

Depending on how they solve the *priority problem*, scheduling algorithms are classified into:

- *Fixed task priority*. Each task has a single fixed priority shared by all its jobs. Examples of this class are the Rate Monotonic (RM) [LL73] and the Deadline Monotonic (DM) [LW82] scheduling algorithms.
- *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) scheduling algorithm [LL73].
- *Dynamic priority*. A single job may have different priorities during its execution. An example of this class is the Least Laxity First (LLF) scheduling algorithm [Leu89], [DK89].

A task set $\mathcal{T}$ is *feasible* on a system Π if there exist a scheduling algorithm that can construct a schedule of tasks such that all task deadlines are met. A task $\tau_i \in \mathcal{T}$ is *schedulable* on Π by using a scheduling algorithm $\mathcal{A}$ if its worst-case response time R_i under $\mathcal{A}$ is less than or equal to its deadline D_i. If all tasks in $\mathcal{T}$ are schedulable on Π under $\mathcal{A}$, then task set $\mathcal{T}$ is *schedulable* on Π under $\mathcal{A}$. Finally, a scheduling algorithm $\mathcal{A}$ is *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.

The real-time scheduling theory provides various *schedulability tests* to check a schedulability of a task set on a system under a given scheduling algorithm. A schedulability test is termed *sufficient* if all of the task sets that are deemed schedulable according to the test are in fact schedulable [DB11]. A schedulability test is termed *necessary* if all of the task sets that are deemed unschedulable according to the test are in fact unschedulable [DB11]. Finally, a schedulability test that is both sufficient and necessary is an *exact* schedulability test.

2.2.4 Uniprocessor Schedulability Analysis

In this section, we will present the most used scheduling algorithms and their schedulability tests for real-time periodic tasks on uniprocessors. These scheduling algorithms are the Earliest Deadline First (EDF), Rate Monotonic (RM) and Deadline Monotonic (DM) scheduling algorithm.

Earliest Deadline First (EDF)

The EDF algorithm is a scheduling algorithm that schedules tasks' jobs according to their deadlines. The earlier deadline a task's job has, the higher execution priority is given to it. The schedulability of an implicit-deadline periodic task set on a uniprocessor under EDF can be verified through the processor utilization. In particular, the following theorem gives a schedulability test for an implicit-deadline periodic task set on a uniprocessor under EDF [LL73]:

Theorem 2.2.1. *A set of periodic tasks $\mathcal{T}$ with implicit deadlines is schedulable under EDF if and only if*

$$\sum_{\tau_i \in \mathcal{T}} u_i \leq 1. \tag{2.7}$$

This schedulability test on uniprocessors under EDF is *exact*. In addition, the EDF scheduling algorithm is an *optimal* scheduling algorithm for periodic tasks on uniprocessors.

The *exact* schedulability test for constrained-deadline periodic tasks on uniprocessors under EDF is given by the following lemma [BRH90]:

Lemma 2.2.1. *A periodic task set $\mathcal{T}$ is feasible on one processor if and only if*

1. $\sum_{\tau_i \in \mathcal{T}} u_i \leq 1$, *and*
2. $dbf(\mathcal{T}, t_1, t_2) \leq (t_2 - t_1)$ *for all* $0 \leq t_1 < t_2 < \hat{S} + 2H$,

where $\hat{S} = \max\{S_1, \cdots, S_n\}$ *and* $H = \mathrm{lcm}\{T_1, \cdots, T_n\}$.

However, this schedulability test is known to be co-NP-hard in the strong sense [BRH90], hence performing the schedulability test is very time consuming. To improve, that is, reduce, the schedulability test time, researchers proposed several *sufficient* schedulability tests for (asynchronous) constrained-deadline periodic tasks on uniprocessors under EDF, such as [ZB09], [AS04] and [BF05]. These algorithms either check smaller number of time points to determine the schedulability of a task set [ZB09] or approximate the processor demand bound function to simplify the computation when checking the schedulability of a task set [AS04], [BF05].

Rate Monotonic (RM)

The RM algorithm is a scheduling algorithm that assigns priorities to tasks according to their rates, that is, periods. The higher rates a task has (that is the shorter period), the higher execution priority is given to the task. Given that the period of a periodic task is constant, RM is the fixed-priority algorithm. The *sufficient* schedulability test for an implicit-deadline periodic task set on uniprocessor under RM is given by the following theorem [LL73]:

Theorem 2.2.2. *A set of periodic tasks $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_n\}$ with implicit deadlines is schedulable under RM if*

$$\sum_{\tau_i \in \mathcal{T}} u_i \leq n(2^{1/n} - 1). \tag{2.8}$$

When the size of the task set is significantly big ($n \to \infty$), then $\sum_{\tau_i \in \mathcal{T}} u_i = ln(2) \approxeq 0.693$. That means that any implicit-deadline task set with total utilization less than 0.69 is schedulable using RM scheduling algorithm. It has been shown in [LL73] that RM is optimal among all fixed-priority assignments in the sense that no other fixed-priority algorithms can schedule an implicit-deadline task set that cannot be scheduled by RM. However, RM is in general not optimal on uniprocessors for real-time periodic task sets.

Deadline Monotonic (DM)

The DM algorithm [LW82] extends the RM algorithm by considering tasks with deadlines less than or equal to their period, that is,constrained deadlines. According to the DM algorithm, higher priorities are given to tasks with shorter relative deadlines. The schedulability of a task set with constrained deadlines can be checked by using the utilization based test given by Relation (2.8), where instead of putting the sum of task utilizations on the left-hand side, we put the sum of task densities. However, such a test would be quite pessimistic, because the workload on the processor would be overestimated. A less pessimistic schedulability test has been proposed in [ABRW91], [ABR$^+$93] based on *Response Time Analysis (RTA)*. That test is formulated in the following theorem:

Theorem 2.2.3. *A periodic taskset $\mathcal{T}$ is schedulable using DM priority scheduling if and only if*

$$\forall \tau_i \in \mathcal{T} : R_i \leq D_i \tag{2.9}$$

where the total response time R_i is given by solving the following fixed-point equation:

$$R_i = C_i + \sum_{\forall \tau_j \in \mathcal{T}_{hp}(\tau_i)} \left\lceil \frac{R_i}{\tau_j} \right\rceil C_j \tag{2.10}$$

and $\mathcal{T}_{hp}(\tau_i)$ represents the set of tasks with priorities higher than the priority of τ_i.

The test in Theorem 2.2.3 can be used as a *sufficient* test for asynchronous periodic tasks.

2.2.5 Multiprocessor Schedulability Analysis

Given a system consisting of m homogeneous processors, and a task set consisting of n periodic tasks, multiprocessor schedulability analysis should determine whether the tasks can be scheduled on the processors. In the following subsections we will present some scheduling algorithms on multiprocessors with regard to how they solve the allocation problem, as introduced earlier in Section 2.2.3.

Global Scheduling Algorithms

Global scheduling algorithms schedule tasks on processors while allowing task migration. Some of these algorithms are *optimal* for implicit-deadline periodic tasks, such as Pfair [BCPV96], LLREF [CRJ06], and SA [KS97]. In the case when these algorithms are used, an *exact* schedulability test for a set $\mathcal{T}$ of implicit-deadline periodic tasks on m processors is:

$$\sum_{\tau_i \in \mathcal{T}} u_i \leq m. \tag{2.11}$$

From Equation (2.11) the absolute minimum number of processors needed to schedule a set $\mathcal{T}$ of implicit-deadline periodic tasks can be computed as:

$$m_{\mathrm{OPT}} = \Big\lceil \sum_{\tau_i \in \mathcal{T}} u_i \Big\rceil. \tag{2.12}$$

In the case of constrained-deadline periodic tasks, there are no optimal online (nonclairvoyant) algorithms for the preemptive scheduling of these tasks on multiprocessors [Fis07].

Partitioned Scheduling Algorithms

Although global scheduling algorithms can be optimal for implicit-deadline periodic tasks, they introduce high migration and preemption overhead. To avoid these overheads, researchers proposed another class of scheduling algorithms, namely, *partitioned* algorithms. As soon as a set of tasks has been *partitioned* into subsets that will be executed on individual processors, the uniprocessor real-time scheduling and analysis techniques can be applied to each processor, which is the main advantage of using partitioning approaches to multiprocessor scheduling.

Given a system with m processors, and a task set of n periodic tasks, a partitioned scheduling algorithm should find a *schedulable x-partition* of the tasks, with $x \leq m$. The schedulable x-partition is a partition: 1) which subsets contain different tasks among each other, that is, a task is allocated to only one processor, 2) where all tasks are partitioned into subsets of tasks, and 3) that guarantees that each subset of the partition is schedulable on one processor under the considered uniprocessor scheduling algorithm.

The task allocation problem in a partitioned multiprocessor scheduling approach is analogous to the *bin packing* problem [DB11], where items corresponds to tasks and bins corresponds to processors. The bin packing problem is known to be NP-hard [GJ79]. Therefore, many heuristics have been proposed to approximately solve the bin packing problem [Joh74], [CGJ96]. Below, we present the most used bin packing heuristics.

The capacity of each bin, that is, processor in a system, is equal to the maximum possible processor utilization, that is, 1. The size of each item, that is, task τ_i, is equal to task utilization u_i. Let $\mathcal{T}_k$ denote the set of tasks currently assigned to processor π_k and $u_{\pi_k} = \sum_{\tau_i \in \mathcal{T}_k} u_i$ denote the total utilization currently assigned to processor π_k. At the beginning of the task partitioning no task is assigned to a processor, that is, $\mathcal{T}_k = \emptyset$ and $u_{\pi_k} = 0$ for all π_k. The heuristics assign each task $\tau_i \in \mathcal{T}$ to a certain processor $\pi_k \in \Pi$ following a certain, heuristic-specific, rule till the task is schedulable on a processor by a certain (selected) scheduling algorithm $\mathcal{A}$, considering one task at a time.

- **First-Fit (FF).** A task τ_i is assigned to the lowest-indexed processor π_k that can contain the task such that all the tasks assigned so far to π_k are schedulable. That is

$$k = \min_{j \in [1 \cdots m]} \{ j : u_i + u_{\pi_j} \leq 1 \wedge \mathcal{T}_j \text{ schedulable by } \mathcal{A} \text{ on } \pi_j \}. \tag{2.13}$$

  If the condition is not satisfied by any processor used so far, task τ_i is assigned to an unused processor in the platform. If no such processor exists, task set $\mathcal{T}$ is not schedulable on system Π.

- **Best-Fit (BF).** A task τ_i is assigned to a processor π_k such that π_k has the minimal remaining utilization after the task assignment and all the tasks assigned so far to π_k are schedulable. That is

$$k = \min_{j \in [1 \cdots m]} \{ j : u_i + u_{\pi_j} \text{ is closest to, without exceeding } 1 \wedge \mathcal{T}_j \text{ schedulable by } \mathcal{A} \text{ on } \pi_j \}. \tag{2.14}$$

  If the condition is not satisfied by any processor used so far, task τ_i is assigned to an unused processor in the platform. If no such processor exists, task set $\mathcal{T}$ is not schedulable on system Π.

- **Worst-Fit (WF).** A task τ_i is assigned to a processor π_k such that π_k has the maximal remaining utilization after the task assignment and all the tasks assigned so far to π_k are schedulable. That is

  $$k = \min_{j \in [1 \cdots m]} \{ j \; : \; u_i + u_{\pi_j} \text{ is minimal} \wedge \mathcal{T}_j \text{ schedulable by } \mathcal{A} \text{ on } \pi_j \}. \tag{2.15}$$

  If the condition is not satisfied by any processor used so far, task τ_i is assigned to an unused processor in the platform. If no such processor exists, task set $\mathcal{T}$ is not schedulable on system Π.

Often a preprocessing step is performed on tasks before performing a heuristic to improve the performance of the heuristic. The preprocessing step represents task sorting according to certain criteria, such us, increasing or decreasing task utilization, increasing or decreasing task density, and so on. Usually, the tasks are sorted in decreasing order of their utilization. When adding this preprocessing step to the previously presented partitioning heuristics, we obtain the **First-Fit Decreasing (FFD), Best-Fit Decreasing (BFD), and Worst-Fit Decreasing (WFD)** heuristics.

The partitioning heuristics can be compared among each other by using their *approximation ratio* metric. The approximation ratio of a heuristic says how much more processors are required to schedule a set of tasks when tasks are partitioned by using the corresponding partitioning heuristic in comparison to an optimal partitioning algorithm. For example, the approximation ratios for FF and BF are 17/10 [CGJ96], [GJ79], while approximation ratio for FFD is 11/9 [Yue91].

The minimum number of processors needed to schedule a task-set $\mathcal{T}$ by a partitioned scheduling algorithm is given by:

$$m_{\text{PAR}} = \min_{x \in \mathbb{N}} \{ x | \exists x\text{-partition of } \mathcal{T} \wedge \forall k \in [1, x] : \mathcal{T}_k \text{ is schedulable on } \pi_k \}. \tag{2.16}$$

Note that m_{OPT} is the lower bound on the number of processors m_{PAR} needed by partitioned scheduling algorithms.

Hybrid Scheduling Algorithms

Although partitioned scheduling approaches have low preemption overheads and do not introduce any migration overhead, they may not, in general, fully utilize available processing resources in a system, that is, they may introduce processing capacity loss. To utilize the benefits of global and partitioned scheduling approaches, researchers proposed hybrid approaches which combine elements of both. Hybrid scheduling approaches can be divided into semi-partitioned and clustering (hierarchical) approaches [DB11].

In the semi-partitioned scheduling approaches, a small number of tasks is allowed to migrate between certain processors to utilize better the processing resources. Examples of semi-partitioned scheduling algorithms which split some tasks into two components that execute at different times on different processors are EKG [AT06], Ehd2-SIP [KY07] and EDF-fm [ABD08].

In hierarchical scheduling approaches [LB03], [HA06], a set of tasks are grouped together and scheduled as a single entity, called server task or supertask. When the entity is scheduled, one of its tasks is selected to execute according to an internal scheduling policy. Hence, the supertasks/servers are scheduled globally, while the scheduling of the tasks within a supertask/server is done locally, that is, it is analogous to scheduling on a uniprocessor. In another case of hierarchical scheduling approaches, for example, [SEL08], tasks are allocated to (virtual) clusters of processors and scheduled according to a global scheduling algorithm on processors within their cluster. In this way, processing capacity loss is less than in fully partitioned approaches, while the small number of processors in each cluster may reduce migration overheads, depending on the particular hardware architecture.