Universiteit
Leiden
The Netherlands

Latency, energy, and schedulability of real-time embedded systems

Liu, D.; Liu D.

Citation
Liu, D. (2017, September 6). *Latency, energy, and schedulability of real-time embedded systems*. Retrieved from https://hdl.handle.net/1887/54951

Version:	Not Applicable (or Unknown)
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/54951

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

Cover Page

Universiteit Leiden

The handle http://hdl.handle.net/1887/54951 holds various files of this Leiden University dissertation

Author: Liu, D.
Title: Latency, energy, and schedulability of real-time embedded systems
Issue Date: 2017-09-06

Chapter 2

Background

> Predictability, not speed, is the foremost goal in real-time system design.
>
> John Stankovic [Sta88]

In this chapter, to better understand this dissertation, we introduce some common preliminaries which we use in the subsequent chapters, such as the cyclo-static dataflow (CSDF) model, the real-time theories, and the hard-real-time scheduling of CSDF.

2.1 Cyclo-Static Dataflow (CSDF) Model

In this dissertation, we use the cyclo-static dataflow (CSDF) model to model streaming applications. In this section, we introduce this model and its properties.

In [BELP96], Bilsen *et al.* proposed the cyclo-static dataflow (CSDF) model to model signal processing applications. CSDF generalizes the well-known synchronous dataflow (SDF) model [LM87]. A CSDF graph is defined as a directed graph $G = (A, \mathcal{E})$, where A is a set of actors and $\mathcal{E}$ is a set of edges. Actor $\tau_j \in A$ represents a piece of computation in an application and edge $e_i \in \mathcal{E}$ represents the communication between two actors, where an atomic data object that is transferred via an edge is called a **token**. In a CSDF graph, every actor $\tau_i \in A$ has an **execution sequence** $[F_i(1), F_i(2), \cdots, F_i(\mathcal{N}_i)]$ of length $\mathcal{N}_i$, meaning that the nth execution/firing of actor τ_i executes the code of function $F_i(((n-1) \mod \mathcal{N}_i) + 1)$. Similarly, each CSDF actor may produce/consume a variable but predefined number of data tokens in consecutive executions, called **production/consumption sequence**. The *production/consumption sequence* has the same length of $\mathcal{N}_i$ as the *execution sequence*. An edge $e_u \in \mathcal{E}$ is a *first-in, first-out* (FIFO) queue defined as pair $e_u = (\tau_i, \tau_j)$, denot-

ing that actor τ_i produces data tokens on edge e_u and actor τ_j consumes data tokens from edge e_u. Let $[x_i^u(1), x_i^u(2), \cdots, x_i^u(\mathcal{N}_i)]$ denote the **production sequence** of actor τ_i on edge e_u, meaning that at the nth execution actor τ_i produces $x_i^u(((n-1) \mod \mathcal{N}_i)+1)$ data tokens on edge e_u. $X_i^u(n) = \sum_{l=1}^{n} x_i^u(l)$ denotes the total amount of data tokens which actor τ_i produces on edge e_u after its first n executions. Let $[y_j^u(1), y_j^u(2), \cdots, y_j^u(\mathcal{N}_j)]$ denote the **consumption sequence** of actor τ_j on edge e_u. Similarly, at the nth execution actor τ_j consumes $y_j^u(((n-1) \mod \mathcal{N}_j) + 1)$ data tokens from edge e_u and $Y_j^u(n) = \sum_{l=1}^{n} y_j^u(l)$ denotes the total amount of data tokens which actor τ_j consumes from edge e_u after its first n executions.

A compelling and important property of CSDF is its *decidability*, i.e., a schedule for all actors in a CSDF graph G can be derived at design time. To derive a valid static schedule of a CSDF graph at design-time, the graph needs to be **consistent** and **live**.

Definition 2.1.1 ([BELP96])**.** A CSDF graph G is said to be **consistent** if a non-trivial solution exists for a *repetition vector* $\vec{q} = [q_1, q_2, \cdots, q_{|A|}]^T$.

The *repetition vector* $\vec{q}$ is defined as follows:

Definition 2.1.2 ([BELP96])**.** Given a connected CSDF graph G, a vector $\vec{q} = [q_1, q_2, \cdots, q_{|A|}]^T$ representing the number of invocations of the actors of G in a valid static schedule is called a *repetition vector* of G.

And the *repetition vector* $\vec{q}$ is computed by using the following theorem,

Theorem 2.1.1 ([BELP96])**.** *For a connected CSDF graph G, a* repetition vector $\vec{q} = [q_1, q_2, \cdots, q_{|A|}]^T$ *is given by*

$$\vec{q} = \mathbf{\Theta} \cdot \vec{r}, \qquad \textit{with} \qquad \Theta_{ik} = \begin{cases} \mathcal{N}_i & \textit{if } i = k \\ 0 & \textit{otherwise} \end{cases} \tag{2.1}$$

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

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

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

$$O_{uj} = \begin{cases} X_j^u(N_j) & \textit{if } \tau_j \textit{ produces on channel } e_u \\ -Y_j^u(N_j) & \textit{if } \tau_j \textit{ consumes from channel } e_u \\ 0 & \textit{Otherwise.} \end{cases} \tag{2.3}$$

Definition 2.1.3 ([BELP96])**.** A CSDF graph G is said to be **live** if a deadlock-free schedule can be found.

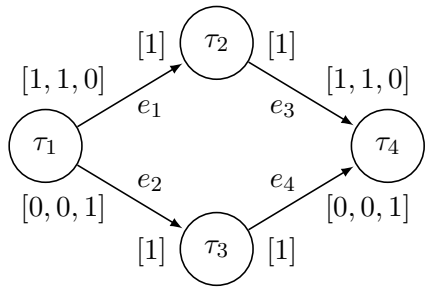

Figure 2.1: CSDF graph G

Definition 2.1.4. For a **consistent** and **live** CSDF graph, the graph completes one *iteration*, if every actor $\tau_i \in A$ executes for q_i times.

Below, we use an illustrative example to facilitate the understanding of the theories and definitions of CSDF presented above.

Example 2.1.1. Consider the CSDF depicted in Figure 2.1. There are four actors $\{\tau_1, \tau_2, \tau_3, \tau_4\}$ and four edges $\{e_1, e_2, e_3, e_4\}$. Each actor has different *production/ consumption sequences* on different edges. For example, actor τ_1 has a production sequence of $[1, 1, 0]$ on edge e_1 and actor τ_2 has a consumption sequence of $[1]$ on edge e_1 and a production sequence of $[1]$ on edge e_3. Then, according to Equation (2.1),(2.2), (2.3) in Theorem 2.1.1, we obtain

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

In this dissertation, we consider streaming applications which are modeled as *acyclic* CSDF graphs. From empirical studies in [TA10], Thies and Amarasinghe showed that 90% of streaming applications can be modeled as acyclic SDF graphs, where SDF graphs are a subset of CSDF graphs. For acyclic CSDF graphs, we have the following lemma,

Lemma 2.1.1 ([BS11])**.** *Any acyclic consistent CSDF graph is live.*

2.2 Real-Time Theories

This section introduces the real-time task models, real-time scheduling algorithms, the schedulability analysis techniques and the multiprocessor real-time scheduling

algorithms.

2.2.1 Real-Time Task Models

A real-time system is comprised of a collection of real-time applications. To ensure the system's timing correctness, each application in the application set is modeled as a real-time task τ_i and all tasks form a real-time task set Γ. A real-time task τ_i that might generate an infinite sequence of task instances, also called *jobs*, is usually specified by a parameter tuple $\{S_i, C_i, D_i, T_i\}$ and the interpretations of the parameters are as follows:

- S_i denotes the start time of τ_i;
- C_i denotes the estimated *Worst-Case Execution Time* (WCET) of τ_i;
- D_i denotes the relative deadline of τ_i; and
- T_i denotes the period of τ_i or the minimal arrival interval of two subsequent jobs generated by τ_i.

The different interpretations of T_i define two widely-used real-time task models: *periodic task model* and *sporadic task model*. In the *periodic task model*, T_i denotes the period of task τ_i, i.e., task τ_i releases its jobs strictly periodically at time instants $t = \{t|S_i + kT_i, k \in \mathbb{Z}^+\}$. In the *sporadic task model*, T_i denotes the minimal arrival interval of two subsequent jobs generated by τ_i, i.e., task τ_i can release its next job anytime once the minimal arrival interval T_i elapses.

The *task model* also features another characeristic related to the tasks' start times. If all tasks in a task set Γ have the same start time, i.e., $S_1 = S_2 = \cdots = S_n$, task set Γ is said to be a *synchronous task set*. Otherwise, it is called an *asynchronous task set*. Moreover, considering the relation between deadline D_i and period T_i, a *periodic/sporadic* task set can be defined as either *implicit deadline* task set or *constrained deadline* task set. The *implicit deadline task set* indicates that every task $\tau_i \in \Gamma$ has $D_i = T_i$, whereas the *constrained deadline task set* means that every task $\tau_i \in \Gamma$ has $D_i \leq T_i$. In real-time systems, the *utilization* is used to denote the ratio of WCET over a period. For a task set Γ, the *utilization* u_i of task $\tau_i \in \Gamma$ is $u_i = C_i/T_i$ and thus the *total utilization* U_Γ of task set Γ is the sum of the utilizations of all tasks, i.e., $U_\Gamma = \sum_{i=1}^{n} u_i$. In addition, we have another term, called *density*, which denotes the ratio of task's WCET over the lesser of a task's relative deadline and period , i.e., the *density* δ_i of task $\tau_i \in \Gamma$ is $\delta_i = C_i/\min(D_i, T_i)$ and the *total density* δ_Γ of task set Γ is computed by $\delta_\Gamma = \sum_{i=1}^{n} \delta_i$.

Table 2.1 lists the corresponding task models used in different chapters. Note that in this section we only introduce the basic task model. The exact task model used

	Task Model		
	synchronous/ asynchronous	constrained/ implicit	periodic/ sporadic
Chapter 3	asynchronous	constrained deadline	periodic
Chapter 4	asynchronous	implicit deadline	periodic
Chapter 5	synchronous	constrained deadline	periodic
Chapter 6	synchronous	implicit deadline	sporadic

Table 2.1: Task models considered in the dissertation' chapters

in each chapter might be slightly different from the basic task model, and we will elaborate the difference in each chapter.

2.2.2 Real-Time Scheduling

When a system and a set of real-time tasks are given, a real-time scheduling algorithm is required to schedule the task set on the specified system. A scheduling algorithm schedules tasks based on their *priority* and there are three ways to assign *priority* to each task [DB11].

- **Fixed task priority**: all jobs from a task have a single fixed priority. An example of this is the *rate-monotonic scheduling* [LL73];
- **Task-level priority**: the jobs of a task might have different priorities, but each job has a single static priority. An example of this is the *earliest deadline first (EDF) scheduling* [LL73];
- **Dynamic priority**: a job of a task might have different priorities at different times. An example of this is the *least laxity first (LLF) scheduling* [Leu89].

Preemption is another important feature pertaining to scheduling algorithms.

- If tasks can be preempted by a task with higher priority at any time, the scheduling algorithm is said to be a *preemptive scheduling algorithm*;
- If tasks start to execute and they cannot be suspended at runtime until their completions, then the scheduling algorithm is said to be a *non-preemptive scheduling algorithm*.

Since the optimal scheduling algorithms for uniprocessor or multiprocessor systems are both *preemptive* [LL73][BCPV93a], we in this dissertation focus our study on *preemptive scheduling algorithms*.

Schedulability Tests and Scheduling Algorithms

As we discussed in Chapter 1, the key problem of real-time systems is to decide whether real-time tasks are schedulable on a specific platform under a certain real-time scheduling algorithm, i.e., to guarantee the system's timing constraint. *Schedulability tests* are the formal and veryfing tools to help us check the schedulability. The formal definition of *Schedulability tests* can be found in Definition 1.2.1 on page 9. In this section, we introduce the schedulability tests used in this dissertation.

Generally, schedulability tests can be classified as follows [DB11]:

- **Sufficient test**: If a real-time task set which is schedulable according to a schedulability test is indeed schedulable, the schedulability test is said to be *sufficient*.
- **Necessary test**: If a real-time task set which is unschedulable according to a schedulability test is indeed unschedulable, the schedulability test is said to be *necessary*.
- **Exact test**: If a schedulability test is both *sufficient* and *necessary*, the schedulability test is said to be *exact*.

In some cases, it is highly difficult to derive an exact test, so we would like to derive a **sufficient test** to ensure the schedulability of a real-time task set.

The *preemptive* earliest deadline first (EDF) scheduling algorithm [LL73] is the most studied dynamic-priority scheduling algorithm. In this dissertation, we use EDF as the scheduling algorithm in **Chapter 5** and **6**. The seminal paper [LL73] proved the *exact* schedulability test for an *implicit deadline periodic* task set under EDF on a uniprocessor system.

Theorem 2.2.1 ([LL73])**.** *For an implicit deadline periodic task set Γ, it is schedulable by EDF if and only if*

$$U_\Gamma = C_1/T_1 + C_2/T_2 + \cdots + C_n/T_n = \sum_{i=1}^{n} C_i/T_i \leq 1. \tag{2.4}$$

On a uniprocessor system, EDF has been proven to be the optimal scheduling algorithm for *implicit deadline tasks* [Der74].

Theorem 2.2.2 ([Der74])**.** *If a job set $\mathcal{J}$ is schedulable by an algorithm A, then it is schedulable by EDF.*

Corollary 2.2.1. *EDF is an optimal scheduling algorithm on a uniprocessor system.*

However, for *constrained deadline* task set, Equation (2.4) only serves as a *necessary* test. Barauh *et al.* in [BMR90] proposed an *exact* schedulability test for *constrained deadline* task set under EDF on a uniprocessor system, where the *exact* schedulability test is derived based on the concept of *demand bound function* (dbf).

$$\mathsf{dbf}(\tau_i, t_0, t_f) = \max\left\{0, \left\lfloor \frac{(t_f - t_0) - D_i}{T_i} \right\rfloor + 1\right\} C_i \tag{2.5}$$

where $t_f - t_0$ denotes a time interval, t_0 and t_f are the start time and the end time of the interval, respectively. dbf computes the maximum cumulative execution time which a task demands within time interval $[t_0, t_f]$. If the total maximum execution time of task set Γ within the time interval does not exceed the time interval, the task set is schedulable. Otherwise, it is unschedulable for the task set.

Theorem 2.2.3 ([BMR90])**.** *A task set Γ is schedulable if and only if $U_\Gamma \leq 1$ and*

$$\forall t < L_a, \quad \mathit{dbf}(\Gamma, t_0, t_f) = \sum_{i=1}^{n} \max\left\{0, \left\lfloor \frac{(t_f - t_0) - D_i}{T_i} \right\rfloor + 1\right\} C_i \tag{2.6}$$

where L_a is defined as follows:

$$L_a = \max\left\{D_1, \cdots, D_n, \max_{1 \leq i \leq n}\{T_i - D_i\} \frac{U_\Gamma}{1 - U_\Gamma}\right\} \tag{2.7}$$

Theorem 2.2.3 is the exact test to check the schedulability of a task set under EDF scheduling. However, the exact test shown in Theorem 2.2.3 is computationally expensive because this exact test needs to test all absolute deadlines within the time interval and there can be a large number of absolute deadlines which need to be checked. To improve the efficacy of the EDF exact test, Zhang and Burns [ZB09] proposed a new exact test for the EDF scheduling, referred as Quick convergence Processor-demand Analysis (QPA).

Theorem 2.2.4 ([ZB09])**.** *A task set Γ is schedulable if and only if $U_\Gamma \leq 1$ and the result of the QPA iterative algorithm shown in Algorithm 1 is $\mathit{dbf}(\Gamma, t_o, t_f) \leq d_{\min}$, where $d_{\min} = \min\{D_i\}$.*

The extensive experimental results in [ZB09] demonstrate the efficiency of QPA in terms of reducing the time complexity of testing the schedulability. Therefore, in our work, we use QPA to test the schedulability of a task set when the utilization-based test is not applicable.

Algorithm 1: QPA

```
1 t ← max{d_i | d_i < L};
2 while (dbf(Γ, t) ≤ t ∧ dbf(Γ, t) > d_min)
3 {if (dbf(Γ, t) < t) t ← dbf(Γ, t);
4 else t ← max{d_i | d_i < L};
5 }
```

2.2.3 Multiprocessor Real-Time Scheduling

Nowadays, the increasing number of real-time systems is implemented on multiprocessor platforms. The prevalence of these real-time multiprocessor systems rises a new problem, namely the *assignment problem*, i.e., deciding which processor to execute which task. Multiprocessor scheduling algorithms can be classified as follows based on the tasks' assignment:

- *Partitioned scheduling*: tasks are only permitted to execute on their assigned processors and task migration is prohibited. On each processor, a uniprocessor scheduling algorithm is deployed to schedule the tasks assigned to the processor;

- *Global scheduling*: tasks are permitted to migrate to any processor at anytime and a global scheduling algorithm assigns tasks to proper processors at runtime;

- *Cluster Scheduling:* the system is comprised of several clusters where each cluster consists of a number of processors. Tasks are statically assigned to a fixed cluster and within a cluster tasks are permitted to migrate to the processors in the same cluster, i.e., a global scheduling algorithm is deployed within a cluster, but task migration between clusters is prohibited;

- *Semi-partitioned scheduling/Task-splitting*: the majority of tasks are statically assigned to processors and only a few tasks are permitted to migrate among processors. Usually, the assignment of migrative tasks is also known at design time, i.e., a migrative task executes partially on one processor and then migrates to another processor to complete its execution.

Assignment Algorithms

When *partition scheduling* is deployed to schedule tasks on a multiprocessor system, a key problem is to efficiently decide how to *assign* a task to a proper processor so

that a certain metric, e.g., schedulability, energy-efficiency, etc, is satisfied[1].

The *assignment problem* of real-time tasks on a multiprocessor system is inherently analogous to the well-known *bin-packing* problem [GJ79]. In the **bin-packing** problem, objects of different volumes are packed into a finite number of bins with fixed capacity such that the number of bins used is minimized. The *bin-packing* problem has been proven to be a **NP**-complete problem [GJ79], so an optimal solution cannot be obtained in a polynominal time unless **P=NP**. Therefore, many heuristic algorithms are developed to efficiently solve the *bin-packing* problem and to obtain a suboptimal result in a reasonable time. The close similarity between these two problems allows us to directly utilize the well-established heuristic algorithms for the *bin-packing* problem to assign real-time tasks on a multiprocessor system. Below, we introduce the most used heuristic algorithms [CGJ97, Joh74].

- **First-Fit (FF)** algorithm: the **FF** algorithm always tries to place an item I_i to the first bin B_j (i.e., lowest index). That is

  $$j = \min\{k : \text{size}(I_i) + \text{capacity}(B_k) \leq 1\}$$

  If no exsiting bin can accomodate the item, a new bin is opened and the item is placed in the new bin;

- **Worst-Fit (WF)** algorithm: the **WF** algorithm always tries to assign an item I_i to the bin B_j which has the most residual capacity after placing item I_i. That is

  $$j = \min\{k : \text{size}(I_i) + \text{capacity}(B_k) \text{ minimized}\}$$

  If no exsiting bin can accomodate the item, a new bin is opened and the item is placed in the new bin;

- **Best-Fit (BF)** algorithm: the **BF** algorithm always tries to assign an item I_i to the bin B_j which has the least residual capacity after placing item I_i. That is

  $$j = \min\{k : \text{size}(I_i)+\text{capacity}(B_k) \text{ maximized \& not exceed the bin volume.}\}$$

  If no exsiting bin can accomodate the item, a new bin is opened and the item is placed in the new bin.

Performance of these heuristic algorithms can be improved by sorting tasks in order of decreasing utilization or density. Then, we have:

[1]Throughout this dissertation, we may use the term "assign", "map", and "partition" interchangeably to denote the procedure that statically decides a processor for a task to complete its execution.

	Priority	Platform	Multiprocessor Scheduling
Chapter 3	task-level/dynamic	multiprocessor	global scheduling
Chapter 4	dynamic	multiprocessor	cluster scheduling
Chapter 5	task-level	multiprocessor	task-splitting
Chapter 6	dynamic [2]	uniprocessor	None

Table 2.2: Scheduling algorithms considered in each chapter

- **First-Fit-Decreasing (FFD)**: all tasks are sorted in decreasing order of their utilization or density (see Section 2.2.1) and then tasks are assigned using the **FF** algorithm;

- **Worst-Fit-Decreasing (WFD)**: all tasks are sorted in decreasing order of their utilization or density and then tasks are assigned using the **WF** algorithm;

Although **partitioned scheduling** has no migration cost and can directly apply a wealth of well-developed uniprocessor real-time theories, it suffers from low resource utilization due to the capacity loss during the assignment procedure [DB11]. The rest of the multiprocessor real-time scheduling algorithms can achieve better resource utilization and additionally the research on them is an increasingly hot topic in the real-time community. Therefore, in this dissertation, we consider the *global scheduling, cluster scheduling, and task-splitting approach*.

In the *cluster scheduling*, the inital step is to assign tasks to a cluster such that a global scheduling can be applied to the tasks assigned to the cluster. Similarly, in the *task-splitting approach*, a subset of tasks are statically assigned to processors and the rest of the tasks are splitted among the processors. The assignment procedures in both the *cluster scheduling* and the *task-splitting aproach* are similar to the assignment procedure in *partitioned scheduling*. Therefore, all heuristic assignment algorithms introduced aboven can be applied to the *cluster scheduling* and the *task-splitting approach*.

Table 2.2 summarizes the priority assignment schemes, platforms, and multiprocessor scheduling algorithms considered in each chapter.

[2] Although the original EDF is a task-level priority scheduling algorithm, EDF-VD may change deadlines of some tasks during runtime. As a result, the priority of these tasks will be changed upon their execution. Thus, we here consider EDF-VD as a dynamic priority scheduling.

2.3 Hard-Real-Time (HRT) Scheduling of CSDF graphs

Throughout this dissertation, instead of traditional dataflow scheduling techniques [MB07], we use a new scheduling framework [BS11, BS12, BS13] to schedule CSDF graphs. In this section, we give a brief introduction about this new scheduling framework.

Traditionally, CSDF graphs are scheduled by self-timed scheduling [MB07], where an actor starts to execute as soon as it receives enough tokens from its predecessors. However, since the self-timed scheduling normally provides best-effort services, it is difficult to provide a hard-real-time timing guarantee for every task in an application. Bamakhrama and Stefanov in [BS11, BS12, BS13] proposed a Hard-Real-Time (HRT) scheduling framework in which an acyclic CSDF graph is converted into an independent periodic task set. This conversion effectively bridges the gap between data-flow models and real-time theories and enables us to apply a plethora of well-developed real-time theories to CSDF graphs. The advantage of this framework is the direct application of real-time theories on dataflow models, such as schedulability tests and assignment algorithms, and the designers are able to accomplish *fast admission control* and *temporal isolation* and provide *hard-real-time guarantees*. A good example of the application of the *HRT* scheduling framework is demonstrated in [BZNS12], where the merit of the *HRT* scheduling framework helps to significantly reduce the complexity of the *design space exploration* when designing a real-time streaming multiprocessor system.

The *HRT* scheduling framework takes as an input a CSDF graph where the WCET of each actor in the CSDF graph is known in a priori, and finally it outputs a periodic task sets which can be scheduled by a real-time scheduler. **Note** that the WCETs considered in the *HRT* scheduling framework also account for the **worst-case communication overhead** because the *HRT* scheduling framework strives to ensure the feasibility of this approach regardless of the variance of different task assignments. The basic concept behind this framework is to derive the real-time parameters, i.e., period, start time, and deadline, for each actor according to actors' WCETs and the CSDF graph properties, e.g., the repetition vector explained in Section 2.1. The procedure goes through the following steps: **1)** it computes a period for each actor in the input CSDF graph according to its *repetition values* and WCETs; **2)** it computes the actors' *start times* such that the precedence constraints between actors are respected and thus the deadlock in execution is avoided; and **3)** it determines the *deadline* of each actor.

Computing Periods

To compute the period of an actor in a CSDF graph, we first define the *workload* of an actor and *the maximum actor workload* of a graph as follows:

Definition 2.3.1. The **workload** of an actor τ_i is $W_i = q_i C_i$ and the **maximum actor workload** of the graph is $\hat{W} = \max_{\tau_i \in G}\{W_i\}$

Then, we can use the *maximum actor workload* and the *repetition value* q_i of actor τ_i to compute the minimum period $\breve{T}_i$ of actor τ_i as follows [BS11]:

$$\breve{T}_i = \frac{\text{lcm}(\vec{q})}{q_i} \left\lceil \frac{\hat{W}}{\text{lcm}(\vec{q})} \right\rceil \tag{2.8}$$

where $\text{lcm}(\vec{q})$ is the *least common multiple* of the *repetition vector* $\vec{q}$ (explained in Section 2.1). The minimum periods deliver the maximal throughput for the CSDF graph under the *HRT* scheduling. Based on this minimum period, a *period scaling technique* can be applied to uniformly scale up the period of each actor under the *HRT* scheduling framework, thereby adjusting the throughput of the CSDF graph [ZBS13, SLS16] .

Computing Start Times

Once the periods of all actors are computed, we need to deal with the precedence constraints between actors. The execution semantics of a dataflow model requires an actor to receive sufficient data from its predecessors in order to trigger its execution, thus the existence of precedence constraints prohibits all actors to start their execution at the same time. To resolve the precedence constraints, the converting procedure in the *HRT* scheduling framework offsets the start times of actors such that by its start time every actor is capable of obtaining sufficient data on its input edges and its subsequently periodic executions are free from deadlock. The following lemma is used to compute the earliest start time for each actor in a CSDF graph

Lemma 2.3.1 (From [BS11])**.** *For an acyclic CSDF graph G, the earliest start time of an actor $\tau_j \in A$, denoted S_j, under HRT scheduling is given by*

$$S_j = \begin{cases} 0 & \text{if } \Omega(\tau_j) = \emptyset \\ \max_{\tau_i \in \Omega(\tau_j)} (S_{i \to j}) & \text{if } \Omega(\tau_j) \neq \emptyset \end{cases} \tag{2.9}$$

where $\Omega(\tau_j)$ is the set of predecessors of τ_j, and $S_{i \to j}$ is given by

$$S_{i \to j} = \min_{t \in [0, S_i + \alpha]} \{t : \underset{[S_i, \max(S_i, t) + k)}{\mathsf{prd}} (\tau_i) \geq \underset{[t, \max(S_i, t) + k]}{\mathsf{cns}} (\tau_j) \; \forall k \in [0, \alpha]\} \tag{2.10}$$

where $\alpha = q_i T_i = q_j T_j$, $\mathsf{prd}_{[t_s, t_e)}(\tau_i)$ is the number of tokens produced by τ_i during the time interval $[t_s, t_e)$, and $\mathsf{cns}_{[t_s, t_e]}(\tau_j)$ is the number of tokens consumed by τ_j during the time interval $[t_s, t_e]$.

Determing Deadlines

After the step of computing the start times, we have three parameters to specify an actor τ_i as a periodic task: the given WCET C_i, the period T_i, and the start time S_i computed by using Equation (2.8) and Equation (2.9), respectively. Recall that a periodic task is specified by a tuple of 4 parameters $\{S_i, C_i, D_i, T_i\}$, so we need to determine a deadline for each actor. In the *HRT* scheduling framework, the deadline D_i of actor τ_i can be selected within a well-defined range $[C_i, T_i]$, i.e., $D_i \in [C_i, T_i]$. The deadline selection is a highly relevant problem for the *HRT* scheduling framework because the deadline selection is able to influence both the performance (mainly the latency) of the CSDF graph and the number of processors required to schedule the CSDF graph. Later in Chapter 3, we propose a novel approach to optimally select deadlines in the *HRT* scheduling framework such that the latency requirement is ensured and the number of processors required is minimized.

Latency and Throughput

After the periodic tasks' parameters are computed, as explained above, the latency and throughput of the CSDF graph scheduled in the *HRT* scheduling framework can be computed. Equation (2.11) is used to compute the minimum latency of the CSDF graph,

$$L(G) = \max_{w \in \mathbf{W}} (S_{\text{out}} + (g^C_{\text{out}} + 1)T_{\text{out}} - (S_{\text{in}} + g^P_{\text{in}} T_{\text{in}})) \tag{2.11}$$

where w is one path of set $\mathbf{W}$ which consists of all paths from the input actor to the output actor. Here, S_{out} and T_{out} are the start time and period, respectively, of output actor τ_{out}, while S_{in} and T_{in} denote the start time and period, respectively, of input actor τ_{in}. g^C_{out} and g^P_{in} are two constants which denote the number of invocations the actor waits for the non-zero consumption/production of tokens on a path $w \in \mathbf{W}$. Note that when $D_i = C_i, \forall i$, the graph can reach the minimum latency achievable by the *HRT* scheduling framework. The throughput of the CSDF graph is computed as follows:

$$\mathcal{R} = 1/T_{\text{out}} \tag{2.12}$$

Note that when all actors have the minimum periods $\breve{T}_i$, the graph can reach the maximum throughput achievable by the *HRT* scheduling framework.