A Simulator for Control Applications on TTNoC-based Multicore Systems

Andrei Cibu

Kongens Lyngby 2014

Abstract

Embedded systems are everywhere. Increasingly they are used in areas such as industrial control, where there are strict requirements on the quality of control (QoC), cost, dependability and security.

There is a trend towards multicore systems and more integration of mixed-criticality functions. In this project we address systems composed of multicore processors, where the cores are interconnected using a time-triggered network-on-chip (TTNoC).

The objective of the project is to develop a simulator for control applications on TTNoC-based multicore systems. The simulator will be used to evaluate the QoC of different system implementations in terms of the scheduling policy used for the tasks, the routing and the scheduling policy used for the messages and the allocation of resources.

Preface

This thesis was prepared at the Department of Informatics and Mathematical Modelling of the Technical University of Denmark in fulfillment of the requirements for acquiring a M.Sc. in Computer Science and Engineering. The work was carried out between September 23, 2013 and April 07, 2014 and it amounts to the equivalent of 32.5 ECTS credits.

The work was supervised by Associate Professor Paul Pop.

Kongens Lyngby, April 2014

Andrei Cibu, s111445

Notations and Symbols

BCET	Best Case Execution Time
C	Computation time
EDF	Earliest Deadline First
ET	Event-Triggered
D	Task relative deadline
d_i	Absolute deadline of the ith task instance
f_i	Finishing time of the ith task instance
Φ	Phase
$G(s)$	Continuous-time model of the plant
h	Nominal sampling period of a control application
$H_A(z)$	Discrete-time model of a actuator system
$H_C(z)$	Discrete-time model of a controller system
$H_S(z)$	Discrete-time model of a sampler system
J_{io}	Input-output jitter
J_S	Sampling jitter
L_{io}	Input-output latency
L_s	Sampling latency
LQG	Linear Quadratic Gaussian
ω_0	Natural frequency of a process
P_τ	Probability density function
Q	Positive semi-definite cost matrix
QoC	Quality of Control
RM	Rate monotonic

r_i	Realease time of the ith task instance
s_i	Start time of the ith task instance
δ	Time resolution of the JITTERBUG timing model
t	Continuous time variable
t_k	Discrete time variable
T	Task period
TT	Time-Triggered
TTNoC	Time-Triggered Network-on-Chip
τ	Task
τ_A	Actuator task
τ_C	Controller task
τ_S	Sampler task
τ_i	Generic task
u	Vector of system inputs
v	Input noise that affects the plant
WCET	Worst Case Execution Time
y	System output vector
x	System state vector

Contents

1. Introduction

"Real-time systems are computing systems that must react within precise time constraints to events in the environment. As a consequence, the correct behavior of these systems depends not only on the value of the computation but also on the time at which the results are produced. A reaction that occurs too late could be useless or even dangerous." [1]

The spectrum of applications that involve real-time systems is very large, ranging from banking transactions or small embedded systems, like ordinary CD-players, to nuclear plant or space mission control systems.

The basic constraint to deal with in a real-time system is *time.* [2] Depending on the consequences of not meeting the time constraints, i.e. missing deadlines, real-time systems can be distinguished in three classes: *hard, firm* and *soft real-time systems.* In case of a hard real-time system, missing a deadline can result in a catastrophic consequence. Soft real-time systems tolerate deadline misses, but they result in degraded performance. Similarly, firm systems tolerate infrequent deadline misses, but the results produced after their deadlines are discarded.

Many real-time systems are control systems. They are used to command, manage or regulate other systems. There are two classes of control systems: *open* and *closed loop systems*, also called *feedback control systems*. In this thesis we will only consider closed loop control systems. A generic model for a closed loop control system is shown in Figure 1.1. The *process*, which, in the control theory literature, is also referred to as the *plant,* represents the system that needs to be controlled. In a closed loop system

the process is constantly monitored in order to ensure that its behavior complies with the desired behavior. Observations about the process are being collected by the *sampler,* also known as sensor or measurement device. Based on these observations, the *controller* computes a control signal, which is meant to compensate for the deviations of the plant from the desired behavior. This control signal is applied to the plant via the *actuator.*

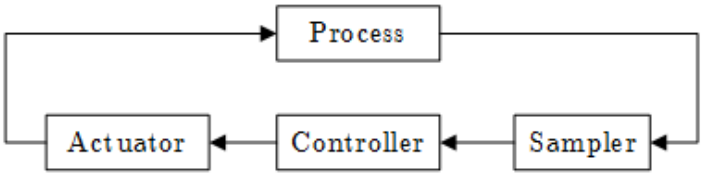


Figure 1.1 Feedback control system

An example of a closed loop control system could be a vehicle fitted with a cruise controller. The purpose of a cruise controller is to maintain a constant speed set by the driver. The process in this case is the vehicle, the cruise control system is the controller, the sampler is represented by the wheel speed sensors and the actuator could be an electrically controlled vacuum actuator that controls the throttle. The cruise controller constantly reads the speed from the wheel speed sensors and compares it with the desired speed. Deviations from the desired speed will be compensated by accordingly adjusting the throttle via the vacuum actuator.

1.1 Thesis Objectives

The goal of this thesis is to develop a simulator, further referred to as SIMULATOR, for hard real-time feedback controllers which run on TTNoC-based multicore systems. The SIMULATOR should be able to evaluate the control performance of these controllers under various system configurations.

1.2 Thesis Overview

This thesis is structured as follows:

Chapter 2 describes the application and architecture models used in this thesis. This chapter also describes the JITTERBUG models that are used to evaluate the control performance of the simulated control applications.

Chapter 3 focuses on the implementation details of the SIMULATOR.

Chapter 4 analyzes the impact that different design decisions concerning the hardware configuration, scheduling policies, task mapping, frame routing or frame scheduling, have on the control performance of a control application.

Chapter 5 concludes the thesis.

2. System Model

This chapter describes the application and the architecture models used in this thesis.

2.1 Application Model

The *applications* executed by the SIMULATOR are modelled as directed, acyclic graphs of *tasks*, further called *task precedence graphs.* A *task* is a sequence of instructions that, in the absence of other activities, is continuously executed by a processor until completion. The task precedence graph is a *rooted* graph, meaning that it has exactly one node, called the *root,* which has no predecessors. A directed edge in the graph denotes a dependency between the connected tasks. More specifically, the task that the edge points to, i.e. *successor task*, depends on the task that the edge leaves from, i.e. *predecessor task,* in the sense that the predecessor task must finish execution and, eventually, produce some data before the successor task can start executing.

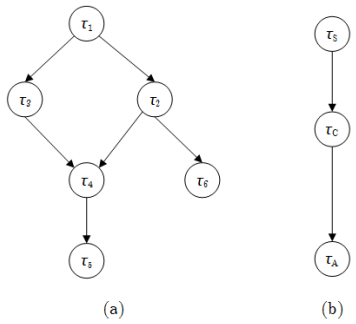


Figure 2.1 Examples of application models: (a) task graph for a generic application and (b) task graph for a control application.

There are two types of applications that the SIMULATOR must handle: *generic applications* and *control applications*.

A control application is a real-time application that implements the functionality of a controller in a control loop. The operations performed by a controller resume to reading the input data from the sampler, executing the control algorithm and sending the control command to the actuator. These three types of operations are modelled with *control tasks,* i.e.: *sampler tasks* (τ_S), *controller tasks* (τ_C) and *actuator tasks* (τ_A). There is a natural precedence between these tasks: the sampler tasks must read the sampled data first and make it available to the controller tasks, which further compute the control signals and send them through the actuator tasks to the actuator. Examples of both generic and control application task precedence graphs are given in Figure 2.1.

Generic applications are those real-time applications that are not involved in control loops. The tasks of a generic application will further be called *generic tasks* ($\tau_i, i \in \mathbb{N}$). We include also the generic applications in the application model as in particular cases they can share the computational resources with the control applications, and therefore they can delay the execution of the control tasks.

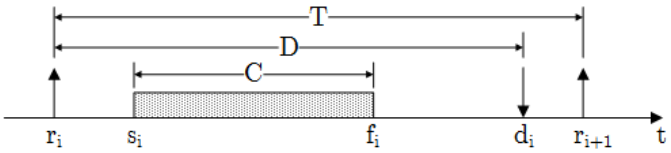


Figure 2.2 Periodic task timing parameters

As mentioned previously, real-time applications are composed of tasks which are the basic executable entities in the system. All the tasks in the system are *periodic* tasks. A periodic task consists of an infinite sequence of task instances or *jobs* which are released with a constant period. A periodic task is characterized by the following timing parameters:

- **Computation time** (C) – the time necessary for a task instance to execute when the processor is fully allocated to it. In most of the cases it is nearly impossible to know the execution times of each task instance, and therefore a more realistic way to express the computation times is through their lower and upper bounds. The lower bound of the execution times is called **best-case execution time** (BCET), while the upper bound is called **worst-case execution time**

(WCET). The computation times of the tasks ran by the SIMULATOR can either randomly vary between their BCET and WCET or they can be fixed and equal to WCET.

- **Period** (T) – it is the time period between successive task activations.
- **Relative deadline** (D) – it is the maximum time allowed for a task instance to finish execution.
- **Release time** (r_i) – it is the time at which the ith instances of a task is created. The release time of the first instance of a task is called *phase* (Φ). Given that a task instance is created regularly, we can agree that $r_i = \Phi + (i-1)T$.
- **Start time** (s_i) – it is the time at which the ith job effectively starts executing.
- **Finishing time** (f_i) – it is the time at which ith job finishes its execution.
- **Absolute deadline** (d_i) – it is the time before which the ith instance of the task must complete its execution ($d_i = r_i + D$).

Control applications are designed to run periodically with a period (h), also called *nominal sampling period*, imposed by the dynamics of the controlled processes. This implies that all the tasks of a control application have their periods equal to the sampling period of the application. In most cases this sampling interval is selected such that $0.2 < \omega_c h < 0.6$, where ω_c is the bandwidth of the closed-loop system [3].

2.2 Architecture Model

The TTNoC-based multicore system considered in this thesis consists of *processing elements* (PE) interconnected by a *time-triggered network on chip* (TTNoC). An example of a TTNoC-based multicore system is depicted in Figure 2.3.

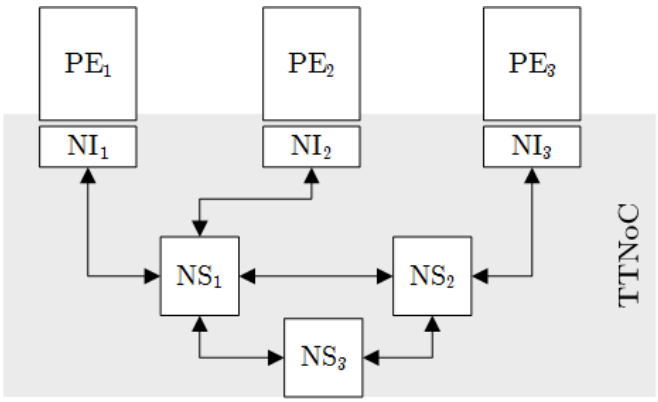


Figure 2.3 Example of a TTNoC-based multicore system with three processing elements connected through a network of three switches

2.2.1 Processing Elements

A processing element represents the hardware component on which tasks are executed. It comprises a processing unit and a memory unit and it has direct access to external signals through input-output ports. The multicore system implements a heterogeneous architecture in which the processing elements can be specialized for different functions. They also have different clock domains.

Each PE is configured to execute a set of periodic tasks. The order in which the instances of the tasks are executed is established according to a predefined criterion, called a *scheduling policy*. The *scheduling algorithm*, which implements a given scheduling policy, differentiates between four states that a task instance can have (see Figure 2.4) and designates the job that should be executed only from those jobs that are in the *READY* or *RUNNING* states.

When a job is created it is assigned the *RELEASED* state. The job remains in this state until it receives a notification that its predecessor job has finished execution. Along with this notification it also receives the data that its predecessor job might have sent to it. Once such a notification is received, the job changes its state to *READY.* Based on the implemented scheduling policy, the scheduler can dispatch the

job for execution, situation in which its state is set to *RUNNING.* If the scheduling policy supports preemption, the running job can be preempted in favor of another job, situation in which its state is set back to *READY.* When a job has run long enough to have completed execution, its state is set to *TERMINATED.*

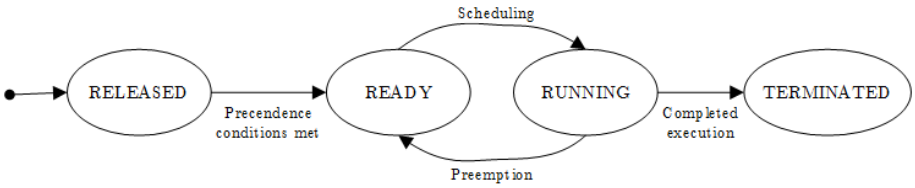


Figure 2.4 Job states

There are three scheduling policies that can be configured on each PE: *static-cyclic scheduling, fixed-priority* (FP) and *earliest deadline first* (EDF) *scheduling.*

The **static-cyclic scheduling** is an offline scheduling policy which executes the tasks in a predefined order and at fixed moments of time. It is configured through a scheduling table in which it is stated what task should be running at any given time. The entries in the table correspond to a period of time, called *hyper-period*, which covers a repeating execution pattern of the tasks. The scheduling table is reiterated every time the hyper-period expires. In most cases the hyper-period is equal to the least common multiple of the periods of the tasks that are to be scheduled. The static-cyclic scheduling policy does not support preemption.

In the **fixed-priority scheduling** the tasks are scheduled based on a priority attribute that each task is assigned at design time. A particular case of fixed-priority scheduling is the *rate-monotonic* (RM) policy. This involves setting the priority of the tasks proportional to their rates, i.e. the higher the rates (or the smaller the periods), the higher the priorities. The RM scheduling is particularly import as it is an optimal scheduling policy among all FP scheduling policies, "in the sense that no other fixed-priority algorithms can schedule a task set that cannot be scheduled by RM" [1]. By a task set that can be scheduled, or a schedulable task set, we understand that all tasks in the set finish execution within their timing constraints. Fixed-priority scheduling is a preemptive scheduling policy: a task can be preempted by any newly released task that has a higher priority.

The **earliest deadline first scheduling** is a dynamic preemptive scheduling policy that selects tasks for execution based on their absolute deadlines. The task with the earliest absolute deadline gets the processor. EDF is an optimal scheduling policy among all scheduling policies, in the sense that if a task set is schedulable, EDF can find a schedule for it [1].

2.2.2 Time Triggered Network on Chip

The TTNoC is modelled as an undirected graph of *network components,* i.e. *network interfaces* (NI) and *network switches* (NS). A network interface is a network component that connects a PE to the network, whereas a network switch is a network component that links several other network components.

We call the link between two network components a *dataflow link.* This is a full-duplex communication channel. An ordered sequence of dataflow links that connect one NI to another is called a *dataflow path.*

Communication Policy

The data transmitted over the network is packed in *frames*. Each frame is configured to follow a fixed dataflow path from source to destination. Therefore, a frame can have a single source and only a single destination. The frames are designed to transport data that originates from a single task.

The TTNoC supports two traffic classes: time-triggered (TT) and event-triggered (ET) frames. TT frames are transmitted over the network based on transmission schedule tables that must be defined on each network component. ET frames, on the other hand, are transmitted over a dataflow link only if all the following conditions are met:

- There are no TT frames scheduled for transmission over the dataflow link.
- The time to the next transmission of a TT frame over the dataflow link is greater than the amount of time required for the ET frame to complete its transmission.
- There is no ongoing transmission of another ET frame over the dataflow link.
- The ET frame has the highest priority among the ET frames that are to be transmitted on the same channel.

The ET and TT traffic classes resemble the time-triggered and the rate-constraint traffic classes defined in the TTEthernet protocol. Also the integration of the TT traffic with the ET traffic is similar to the timely block policy in the TTEthernet protocol. [4]

2.3 Control Application Modelling in JITTERBUG

The control performance of a control application is evaluated using JITTERBUG and it is based on the timing parameters determined for each control task during the simulation of the system. JITTERBUG "is a MATLAB-based toolbox that allows the computation of a quadratic performance criterion for a linear control system under various timing conditions". [5] It offers a convenient way to quickly evaluate how sensitive a control system is to sampling delay, latencies and jitter. Based on linear quadratic Gaussian (LQG) theory and jump linear systems, JITTERBUG can evaluate the performance of a system if it has knowledge of the sampling periods and latency distributions in the control loop.

For a control system to be analyzed in JITTERBUG, it has to be described by two models: a *signal model* and a *timing model*.

2.3.1 Signal Model

The signal model is a representation of the linear continuous- and discrete-time systems in the control loop with their connections. The signal model of the closed loop control systems handled by the SIMULATOR is depicted in Figure 2.5. The plant is described by a continuous-time system $G(s)$. The sampler, the controller and the actuator are described by the discrete-time systems $H_S(z)$, $H_C(z)$ and $H_A(z)$ respectively. The connections between the systems in the control loop are indicated with arrows. The plant can be affected by input noise. The input noise is represented by the signal v.

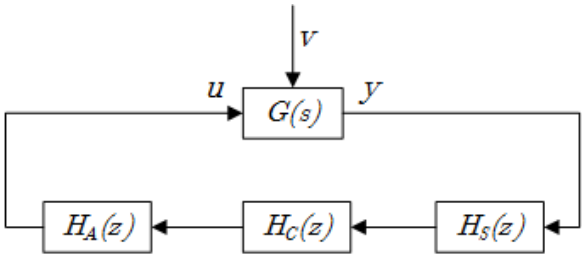


Figure 2.5 JITTERBUG signal model for a closed loop control system

In JITTERBUG, continuous-time and discrete-time systems can be specified in either state-space or transfer-function forms [5].

The state-space form for a continuous-time system is described by

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) + v_c(t), \\ y^0(t) &= Cx(t), && (continuous\ output) \\ y(t_k) &= y^0(t_k) + e_d(t_k), && (measured\ discrete\ output) \end{aligned}$$

where A, B and C are constant matrices and v_c is a continuous-time white-noise process (input noise) with zero mean and covariance R_{1c} and e_d is a discrete-time white-noise process (measurement noise) with zero mean and covariance R_2.

In transfer-function form, the continuous-time system must be specified as

$$\begin{aligned} y^0(t) &= G(p)\big(u(t) + v_c(t)\big), && (continuous\ output) \\ y(t_k) &= y^0(t_k) + e_d(t_k), && (measured\ discrete\ output) \end{aligned}$$

where $G(p)$ is a strictly proper transfer function, i.e. the degree of the numerator is less than the degree of the denominator, v_c is a continuous-time white-noise process with zero mean and covariance R_{1c} and e_d is a discrete-time white-noise process with zero mean and covariance R_2.

The state-space form for the discrete-time systems is

$$\begin{aligned} x(t_{k+1}) &= \Phi x(t_k) + \Gamma u(t_k) + v_d(t_k), \\ y^0(t_k) &= Cx(t_k) + Du(t_k), && (discrete\ output) \end{aligned}$$

$$y(t_k) = y^0(t_k) + e_d(t_k), \quad (measured\ discrete\ output)$$

where v_d and e_d are discrete-time white-noise processes with zero mean and covariance

$$R_d = \mathrm{E}\begin{pmatrix} v_d(t_k) \\ e_d(t_k) \end{pmatrix}\begin{pmatrix} v_d(t_k) \\ e_d(t_k) \end{pmatrix}^T$$

In transfer function form, discrete-time systems are given as

$$\begin{aligned} y^0(t_k) &= H(q)\big(u(t_k) + v_d(t_k)\big), && (discrete\ output) \\ y(t_k) &= y^0(t_k) + e_d(t_k) && (measured\ discrete\ output) \end{aligned}$$

where $H(q)$ is a proper transfer function and v_d and e_d are discrete-time white-noise processes with zero mean and covariance R_d defined in (3.4).

2.3.2 Timing Model

The timing model is a representation of the delays encountered in the control loop. It consists of a list of connected timing nodes. Each timing node is associated with a time delay. This association indicates that the system waits for that time until it advances to the next node. A timing node can also be associated with one or more discrete-time systems. Such an association indicates that the discrete-time system(s) is/are updated when the node is entered. Delays are described by discrete-time probability density functions $P_\tau = [P_\tau(0) \;\; P_\tau(1) \;\; \ldots]$, where $P_\tau(k)$ represents the probability of a delay of $k\delta$ seconds, δ being the time-grain of the model, i.e. the minimum representable time in the system.

The timing model can represent both periodic and aperiodic systems. We consider only period systems in our analysis. The periodicity of a system is modelled by considering the first timing node in the model to be periodic. This node will be entered at times which are multiple of the time period. A consequence of this is that the cumulated time delays in the timing model cannot exceed the period, as the system will restart when the period expires. This behavior models hard deadlines in real time systems and it matches the application model described previously.

The JITTERBUG timing model used by the SIMULATOR is depicted in Figure 2.6. The timing nodes are represented by circles and the discrete-time systems by rectangles.

The first node, illustrated with a double circle, is a periodic node, having the period equal to the period of the simulated control application.

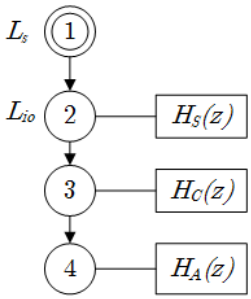


Figure 2.6 JITTERBUG timing model of the supported systems

The only delays that we expect on the signal path occur in the controller block. We differentiate between two types of delays: sampling latencies (L_s) and input-output latencies (L_{io}). The sampling latency is the time from the start of a controller cycle, i.e. multiple of the sampling period, when the sampler task is instantiated until it starts to execute. Therefore the process is considered to be actually sampled only when the sampler task becomes active. This delay can be dealt with only by revising the used scheduling policy. The input-output latency is the delay from when the process is sampled, i.e. sampler task starts, until the actuator task finishes execution. Sources of input-output delays are the scheduling policy, the non-zero execution times of the tasks and the inter-task data communication delays.

The signal model in Figure 2.6 can therefore be interpreted as: at the beginning of each period a random sampling delay (L_s) is waited until the output of the process is sampled, i.e the sampler block is executed (H_S). Then it takes another random input-output delay (L_{io}) until the controller block (H_C) is executed. As soon as the controller block computes the control signal, the actuator block (H_A) is activated.

2.4 Design Metrics

2.4.1 Timing Parameters

In the ideal case the sampling period would be constant, i.e. the output of the process would be sampled at times which are exact multiples of the sampling period. Moreover, the control signal would be computed instantly once the plant is sampled. In reality, things are different. A control application, i.e. the controller, consists of several tasks responsible with reading the data from the sensors, running the control

algorithm and sending the control signal to the actuator. All these tasks are subject to scheduling and their execution times are non-zero. This basically means that there are latencies associated with the controller.

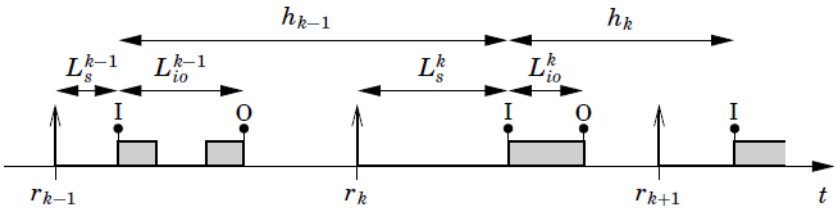


Figure 2.7 Controller timing. From [6]

We define next the timing parameters (see also Figure 2.7) that characterize the delays in the controller:

- **Sampling latency** (L_s) – it is the time passed from the release of the sampling task until it actually starts executing. We consider that the process is sampled only when the sampling task starts to execute.
- **Sampling jitter** (J_S) – it characterizes the variation of the sampling latency. It is the difference between the maximum and the minimum sampling latencies.

$$J_S = L_S^{max} - L_S^{min}$$

- **Input-ouput latency** (L_{io}) – it is the time from when the sampler task starts executing until the actuator task finishes executing.
- **Input-output jitter** (J_{io}) – it is a measure of the variation in the input-output latency and it is defined as the difference between the maximum and the minimum input-output latencies: $J_{io} = L_{io}^{max} - L_{io}^{min}$.

Sampling jitter

A constant sampling latency has no impact on the performance of the controller as the sampling interval would remain equal to the nominal one. A variation of the sampling latency instead, i.e. the sampling jitter, translates into a variation of sampling interval, causing the control performance to degrade. Sampling jitter compensation methods, e.g. constant adjustment of the parameters of the controller based on the length of the last sampling interval [6], exist, but they are not in the concern of this thesis.

Input-output latency

In the control theory it is well known that "a constant input-output latency decreases the phase margin of the control system, and that it introduces a fundamental limitation on the achievable closed-loop performance" [6]. There are ways to account for a constant input-output delay when designing the control system.

The main sources for input-output latencies are:

- the non-zero execution times of the tasks in the controller;
- scheduler-induced latencies, i.e. the waiting periods before getting the processor or the preemption times;
- communication-induced latencies, i.e. time that it takes for a frame to be transmitted over the network.

In order to reduce the input-output latencies we can address their sources. The communication-induced latencies can be reduced by mapping the tasks that exchange the most data preferably to the same PE. If the system configuration does not allow this, then one could try to increase the priorities of the frames, if they are event-triggered, or pick different dataflow paths that would result in reduced transmission times.

The scheduler-induced latencies can be reduced by assigning higher priorities to the tasks in case of FP scheduling, or setting shorter deadlines in case of EDF scheduling. Also a non-preemptive scheduler could be used to avoid preemption. Any of these approaches have an impact on some other applications running in the system.

The latencies caused by the non-zero computation times of the tasks could be reduced by separating the calculation of the control signal part of the control algorithm from the part that updates its internal states. In this way, the control signal can be computed and sent to the actuator before updating the internal states, therefore the delay from sampling to actuation is reduced [6].

Input-output jitter

The input-output jitter represents the variation of the input-output latency. If present, it causes degraded control performance [7]. There are several jitter compensation schemes [6] [7] that could be implemented to reduce the effects of the jitter on the control performance, but they are not in the scope of this thesis.

2.4.2 Cost Computation

The performance of a control system is evaluated using JITTERBUG based on the timing parameters determined during the simulation of the system. By knowing the distributions of the latencies in the system, JITTERBUG computes a quadratic performance criterion, further referred to as *cost function.*

The cost function does not capture all the aspects of a control loop, therefore it does not fully characterize the performance of a system, but it provides an easy way to quickly evaluate different controller configurations. The goal is to minimize the cost function. Higher values suggest that a system is less stable, i.e. more oscillatory, and infinite values mean that the system is unstable. [8]

The cost function that evaluates the performance of a controller in a closed loop is given by

$$J = \lim_{T\to\infty} \frac{1}{T} \int_0^T \begin{pmatrix} x(t) \\ u(t) \end{pmatrix}^T Q \begin{pmatrix} x(t) \\ u(t) \end{pmatrix} dt$$

where Q, referred to as *cost matrix,* is a positive semi-definite matrix and x and u are the state and the input vectors of the plant.

3. Design and Implementation

The SIMULATOR consists of two main parts. There is a part that simulates the execution of a TTNoC-based multicore system and a part that evaluates a quadratic performance criterion for the control applications that are executed on the system. These two parts are implemented using different technologies. The system simulation part is implemented in .NET, using the C# programing language, while the computation of the quadratic performance criteria is performed in MATLAB using the JITTERBUG toolbox [5].

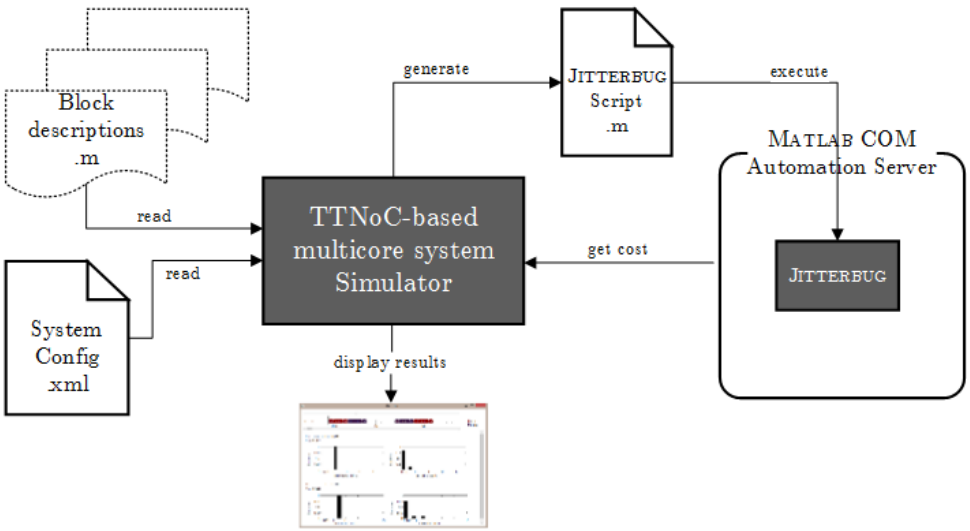


Figure 3.1 SIMULATOR - information flow diagram

By following the information flow diagram, depicted in Figure 3.1, we can briefly summarize the way SIMULATOR works. It starts by reading an .xml file, further called the *system configuration file*, which contains the setup of the multicore system. Once

the system is configured, the simulation can start. The outcome of the simulation is a set of timing parameters corresponding to each control application. By using these timing parameters and the description functions of the systems involved in the control loops, the tool generates JITTERBUG models and stores them in MATLAB script files (*.m files). The scripts are evaluated in MATLAB, through a MATLAB COM Server. The results of the evaluations are imported back into the application and displayed in an output window.

3.1 Requirements

In this project we intend to build a tool for simulating control applications that run on TTNoC-based multicore systems. The simulator must therefore implement the following requirements:

- It must support any application that has the characteristics described in the application model. The number of applications that are simulated at a time must not be restricted.
- It must handle all hardware configurations that comply with the architecture model, including heterogeneous architectures.
- It must implement the fixed-priority, earliest deadline first and static cyclic task scheduling policies.
- It must support both event-triggered and time-triggered frame classes.
- It must be easily configurable through a configuration file.
- It must impose hard deadline constraints on all the tasks, i.e. the tasks that do not finish execution before their deadlines are to be discarded.
- Application instances which do not finish before their deadlines will be considered as having input-output latencies equal to their period plus an extra clock cycle. This is in agreement with the JITTERBUG signal model, which interprets all latencies greater than the period of the application as deadline misses and the cost is penalized accordingly..
- It must display task execution and network communication timing diagrams based on the logs collected during the simulation of the system. The task execution diagrams will only include tasks of control applications.
- It must display the timing parameters determined from simulation for each control application, i.e. sampling latency and input-output latency probability distributions.
- It must display the calculated QoC for each control application.

3.2 TTNoC-Based Multicore System Simulator

The SIMULATOR was developed in the C# programming language and it was designed as a stand-alone application.

The **MulticoreSystem** class is the class that orchestrates the simulation of a multicore system. It is basically an in-memory representation of a multicore system. It contains references to all the processing elements, to the TT network that connects them, to the applications that are configured to run in the system and to the frames that are to be transmitted over the network. At initialization, it reads the system configuration file and it configures the system accordingly. For the multicore system to pass the configuration stage it must be ensured that the configuration file complies with the schema given in Appendix A.

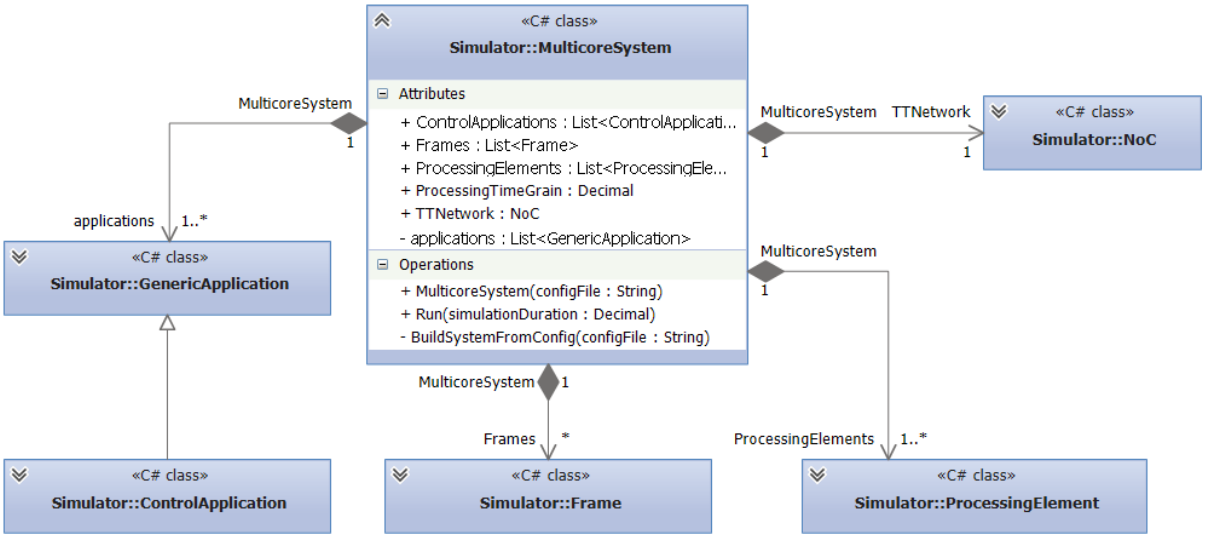


Figure 3.2 Multicore system - UML Class Diagram

The *Run()* method in the **MulticoreSystem** class launches the simulation of the multicore system. It triggers the activation of the PEs and the TTNoC at times that are multiple of their internal clock periods. The clock periods of the internal timers in the PEs and the TTNoC are expected to be evenly divisible by the smallest of these periods.

Before describing the behavior of a processing element on activation, we introduce the classes that implement the application model. As stated section 2.1, there are two types of applications that the simulator supports: generic and control applications.

They are implemented by the **GenericApplication** and the **ControlApplication** classes respectively (see Figure 3.4), the latter extending the former. Each application represents a directed graph of **Task** objects that corresponds to a task precedence graph. The graph is implemented by the generic class **Graph**, using adjacency lists. Both application classes extend the **Graph** class depicted in Figure 3.3.

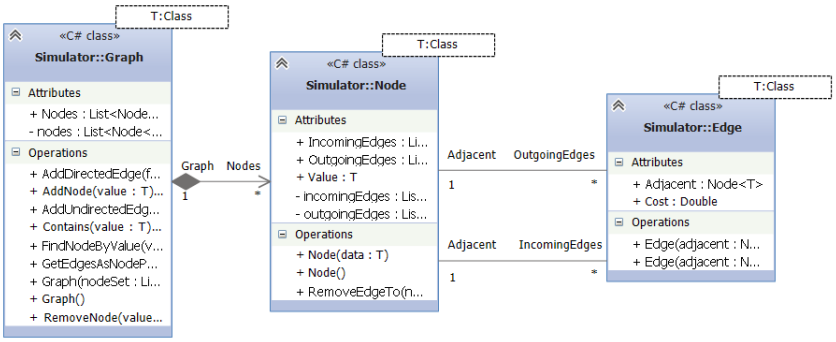


Figure 3.3 Graph – UML Class Diagram.

A task is implemented by the **Task** class. This class holds information about the configurable timing parameters of a task, i.e. period, BCET, WCET and its relative deadline, and it logs the execution of the tasks during the simulation of the system. The task execution logs are used to derive the timing parameters of the control applications. The calculation of the timing parameters, based on the logged data, is performed in the *CalculateTimingParameters()* method implemented by the **ControlApplication** class.

The actual schedulable entities in the simulator, i.e. the jobs, are instances of the **Job** class. They mimic the successive instances of a task. A job inherits the properties of a task, including its dependencies.

The execution of a task, materialized through its jobs, is tracked in a task execution log, which is implemented as a list of **TaskExecutionLog** objects. There is such an object associated with every job of a task. The **Task** class provides the *CreateJob()* method which creates a new job and adds an entry for it in the execution log. An execution log entry stores information such as the release time, the response time or the active times of a job. It also tracks whether the execution of a job was aborted due to missed deadline. A job updates its execution log by calling the *LogRunningJob(), LogJobTerminated()* or *LogJobAborted()* methods provided by its parent task.

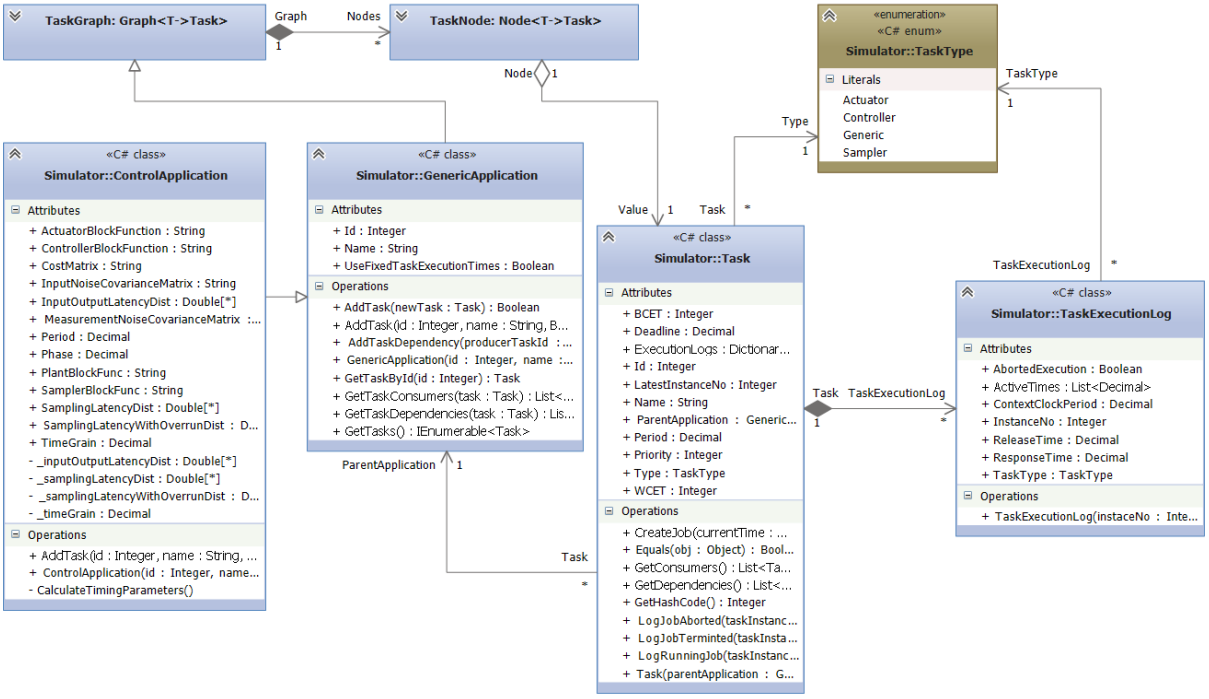


Figure 3.4 Applications – UML Class Diagram.

The jobs are executed by the processing elements in the system, which are implemented by the **ProcessingElement** class. The implementation is in accordance with architecture model. Each processing element is designated to execute jobs for a given set of tasks. The jobs are allocated the computing resources according to a schedule determined by the employed scheduler. A scheduler must implement the **ITaskScheduler** interface. This interface defines only one method, i.e. *GetRunningJob()*, which returns a reference to the job that should run at a given time. The current implementation of the SIMULATOR includes three scheduling policies: fixed priority, earliest deadline first and static cyclic scheduling, implemented in the **TaskPriorityBasedScheduler, TaskEarliestDeadlineFirstScheduler** and **TaskStaticCyclicScheduler** classes respectively.

All scheduler classes use job lists to store the active jobs. A job is considered to be active if it did not complete its execution or it did not miss its deadline. All jobs that miss their deadlines are aborted, this being in compliance with the hard deadline constraints policy. These commonalities are implemented in the **TaskScheduler** class which is extended by the three scheduler classes.

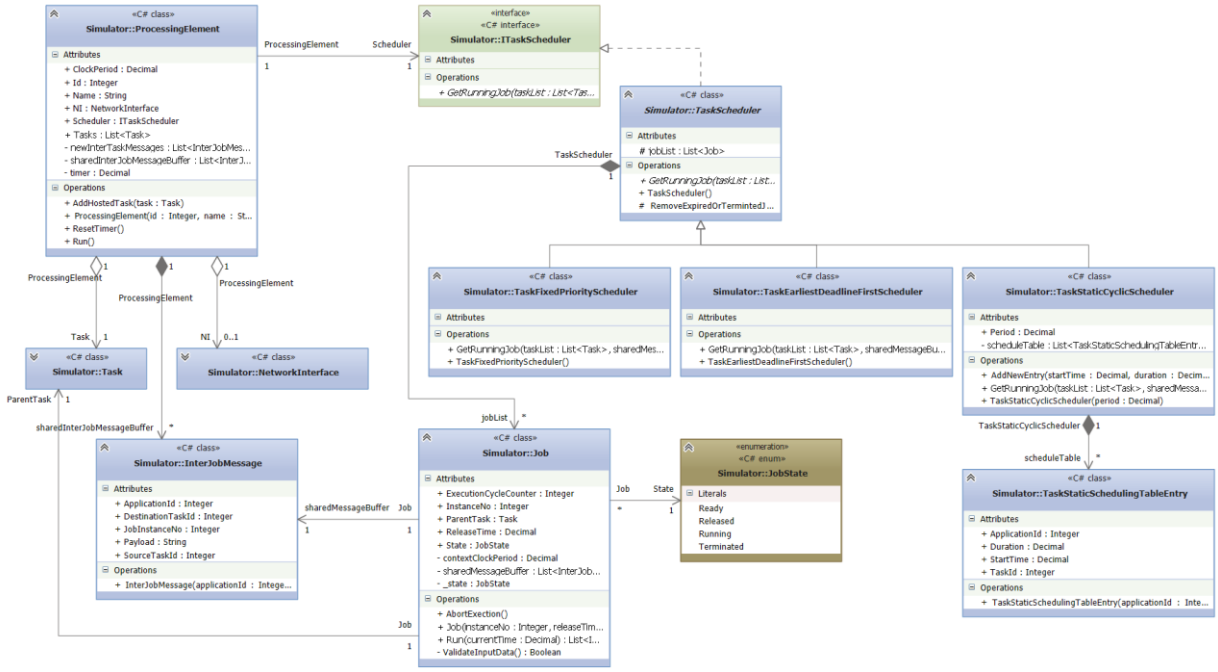


Figure 3.5 Processing elements – UML Class Diagram.

The selection of a job for execution depends partly on the scheduling policy implemented by the scheduler and partly on the state of the job. The possible states of a job are: RELEASED, READY, RUNNING and TERMINATED. Right after creation, a job enters the RELEASED state. It remains in this state until the data from the jobs that it depends on becomes available. The data exchanged by the jobs is packed in a message-like form that contains information about the job that produced it and the job that should consume it. We call such a message an inter-job message. The inter-job messages are stored in the local memory of a processing element until they are consumed by the jobs they are addressed to. When all the inter-job messages that a job expects become available, the state of the job switches to READY. A job that is in the READY state can be selected for execution by the scheduler. The processing element simulates the execution of a job by calling the job's *Run()* method. When a job is executing, its state changes to RUNNING, the execution is logged and the remaining execution time of the job is decremented. If the remaining execution time reaches zero, the state of the job is set to TERMINATED. The total execution time of a job is equal to the WCET of the parent task, or to a random time between BCET and WCET, depending on whether the simulator is configured to use fixed or random execution times. When the job completes execution, it produces inter-job messages addressed to all of its consumer jobs. These messages are inserted into the local memory of the PE.

Besides the selection of jobs for execution, a scheduler is also responsible with the creation of new jobs. The FP and EDF schedulers create new jobs at the beginning of each task period, whereas the static cyclic scheduler creates a new job when the entry in the static schedule table indicates the start of a new job. A static schedule table is implemented as a list of **TaskStaticSchedulingTableEntry** objects, which store the start time and the duration of the execution of a given task.

Job scheduling and job execution are triggered within the *Run()* method of the processing element. This method is called from the **MulticoreSystem** class at discrete times that are multiples of the internal clock period of the PE.

Those inter-job messages produced by the jobs that finished execution that are addressed to jobs located on other nodes, are packed by the processing element in their corresponding frames. The frames are passed to the network interface to be transmitted over the network.

The time-triggered network in the multicore system is implemented by the **NoC** class. It fully complies with the architecture model defined in section 2.2.2. As shown in Figure 3.6, this class represents a graph of **NetworkComponent** objects. The edges in the graph are the dataflow links that connect the network components.

The definitions of the frames that the network must handle are stored in objects of class **Frame**. These objects also store the frame transmission logs. A transmission log contains the times when a frame instance was transmitted over a dataflow link. The information from the logs is used to determine the frame transmission latencies. The actual entities that are transferred over the network correspond to frame instances and they are of type **FrameInstance.** A frame instance inherits all the properties of a frame.

The transmission of the frames instances over the network is simulated by repeatedly calling the *Run()* method of the **NoC** class. Every call to *Run()* simulates the transmission of the amount of data that corresponds to one network clock cycle. This method triggers the data transmission in all of the network components, by calling their respective *Run()* methods.

The **NetworkComponent** class implements the network communication policy. It serves as the base class for the **NetworkInterface** and the **NetworkSwitch** classes, which correspond to the network interfaces and the network switches in the TTNoC.

The only difference between these two is the fact that only a network interface can introduce a new frame instance in the network.

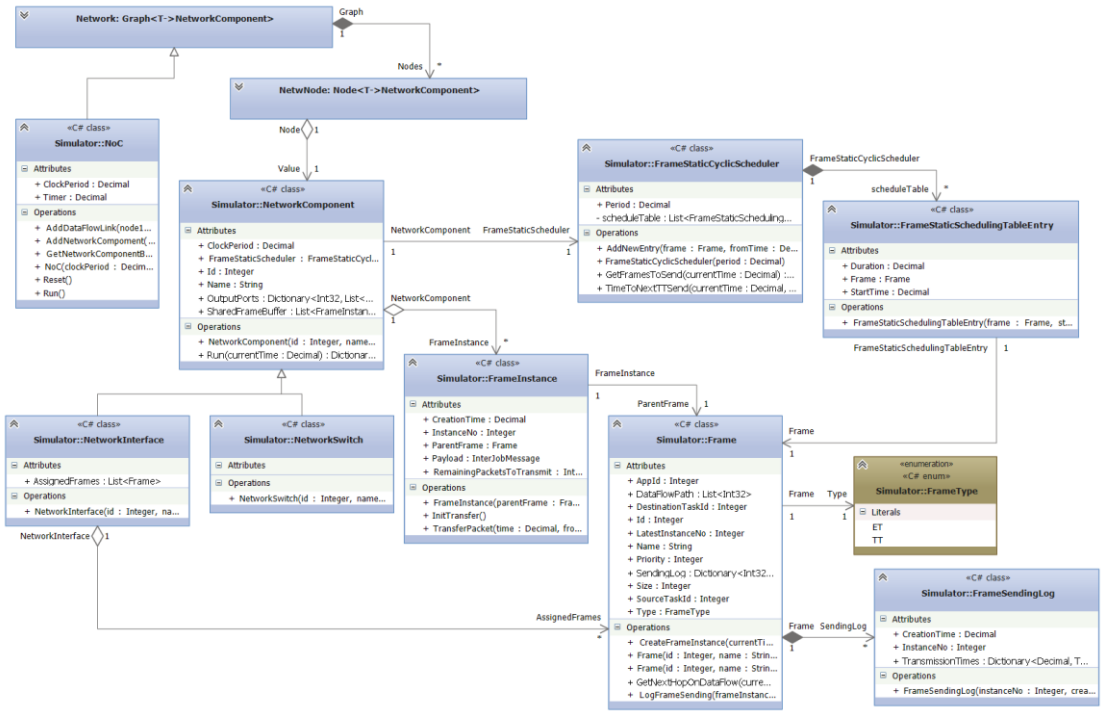


Figure 3.6 TTNoC – UML Class Diagram.

Each network component has an input buffer, i.e. *SharedFrameBuffer*, where it stores all the received frame instances, and several output ports, one for each dataflow link.

The network communication policy is implemented in the *Run()* method of the **NetworkComponent** class. The transmission of the TT frames is scheduled according to a static frame scheduler deployed on the network component and implemented by the **FrameStaticCyclicScheduler** class. The scheduler returns, through the *GetFramesToSend()* method, the TT frames that are to be transmitted at a given time. These frames, if present in the input buffer, are immediately moved to the appropriate output port.

The transmission of an ET frame, on the other hand, on an output channel is only allowed if all the following conditions are met:

- There are no TT frames scheduled for transmission on the output channel;

- The time to the next transmission of a TT frame on the output channel is greater than the amount of time required for the event-triggered frame to complete its transmission. The time until the next send of a TT frame is returned by the *TimeToNextTTSend()* method provided by the static scheduler.
- There is no ongoing transmission of another event-triggered frame on the output channel
- The event-triggered frame has the highest priority among the event-triggered frames that are to be transmitted on the same output channel.

A frame instance is kept in an output port until it is completely transmitted. The transmission of a frame instance is simulated by calling its *TransferPacket()* method. This method decrements the remaining number of packets and it logs the transmission by calling the *LogFrameSending()* method of its parent frame. When the number of packets reaches 0, the frame instance is removed from the output port and it is inserted into the input buffer of its destination network component.

At the end of the simulation, when all the timing information is collected, the SIMULATOR generates JITTERBUG models for all of the control applications, by calling the *GenerateJitterbugScript()* method in the **MatlabBridge** utility class. These models are evaluated then by a MATLAB COM server object which is invoked in the *EvaluateJitterbugScript()* method provided by the same utility class.

The simulation results are displayed in an output window which is built using the WPF framework.

3.3 Input/Output Files

3.3.1 SIMULATOR Configuration File

The settings of the SIMULATOR are stored in the App.config file, which is located in the same folder as the application. These settings include:

`JitterbugToolboxPath`	Local path to the JITTERBUG toolbox.
`JitterbugModelsOutputPath`	Folder where the JITTERBUG scripts are to be saved.
`SimulationTime`	Length of the simulation, expressed in seconds.
`SystemConfigurationFile`	Path to the system configuration file.

3.3.2 System Configuration File

The system configuration file contains information about the architecture of the multicore system and the applications and the frames that are handled by the system. Figure 3.1 shows an overview of the structure of this file as generated from its schema (see Appendix A), using <oXygen/> XML Developer[1].

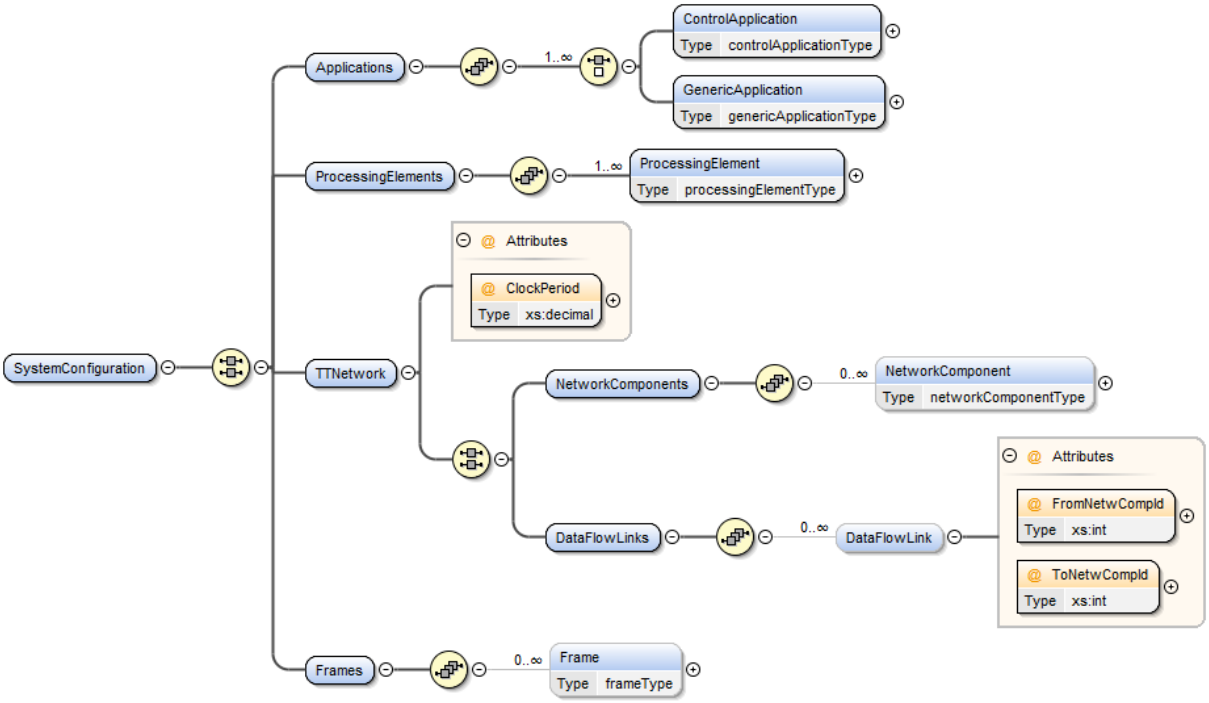


Figure 3.7: Structure of the system configuration file

The .xml file is basically divided into four main sections where the real-time applications, processing elements, time-triggered network and the frames are to be configured.

Applications

The list of applications that are to be handled by the system must be inserted as child elements of the **Application** element. There are two types of supported applications: generic and control applications, identified in the .xml file by the **GenericApplication** and **ControlApplication** elements respectively.

[1] <oXygen/> XML Developer is a tool for XML development. It can be downloaded at http://www.oxygenxml.com/xml_developer.html

A **GenericApplication** element has the following attributes:

`Id`	Number that uniquely identifies an application.
`Name`	String that represents the name of the application.
`UseFixedTaskExecutionTimes`	Flag that indicates whether the application tasks have fixed execution times, i.e. equal to their WCETs, or the execution times randomly vary between BCET and WCET.

Besides the attributes of a generic application, a control application has a few extra attributes, most of them containing information about the control loop components. The attributes that are characteristic to a control application are the following:

`Period`	Sampling period of the control application expressed in seconds. This value will be passed as period to all the tasks that belong to this application.
`Phase`	The amount of time to delay the start of the first instance of the application. This is an optional parameter. By default the phase is considered equal to 0.
`G`	String representing the path to the MATLAB function file that describes the controlled process. The function must return a strictly proper continuous-time LTI system in state-space, transfer function, or pole-zero-gain form.
`H_S`	String representing the path to the MATLAB function that describes the sampling block. The function must return a discrete-time LTI system in state-space or transfer function form.
`H_C`	String representing the path to the MATLAB function that describes the controller block. The function must return a discrete-time LTI system in state-space or transfer function form.
`H_A`	String representing the path to the MATLAB function that describes the actuator block. The function must return a discrete-time LTI system in state-space or transfer function form.
`CostMatrix`	String representing a valid positive semi-definite matrix expressed in MATLAB code. This matrix corresponds to the Q matrix used in the definition of the cost function evaluated in JITTERBUG.

`InputNoiseCovarianceMatrix`	String representing a valid matrix expressed in MATLAB code. It corresponds to the covariance matrix of the input noise (R_{1c}) that disturbs the controlled process.
`MeasurementNoiseCovarianceMatrix`	String representing a valid matrix expressed in MATLAB code. It corresponds to the covariance matrix of the measurement noise (R_2).

As described in Chapter 2, applications are made of tasks, which are organized in precedence graphs. As a consequence of this, a real-time application, either generic or control application, contains a list of tasks and task dependencies organized under the **Tasks** and the **TaskDependencies** elements. An application must contain at least one task in order to be considered valid.

There are some slight differences between the definitions of a control application task and a generic application task. We will cover first their common attributes and specify then the specific ones. Any task should be defined within a **Task** element which includes the following attributes:

`Id`	Number that uniquely identifies a task within an application.
`Name`	String that represents the name of the task.
`BCET`	Best-case execution time of the task, expressed in number of clock cycles, hence BCET is dependent on the clock rate of the processing element on which the task is executed.
`WCET`	Worst-case execution time of the task, expressed in number of clock cycles, hence WCET is dependent on the clock rate of the processing element on which the task is executed.
`Priority`	Number representing the priority level of a task. The higher the value, the higher the priority. This is an optional attribute and it is only used in case a priority-based scheduling policy is employed, i.e. FP.

Unlike the tasks of a control application which all inherit the period of the application, the tasks of a generic application must be specified a period. This is done in the following attribute, which is part of the **Task** element:

`Period`	Period of a task given in seconds. It is an optional attribute which is used with non-static scheduling policies, i.e. FP and EDF.

The tasks of a control application are specialized tasks, therefore they must be specified their types. This is done in the following attribute:

`Type`	String representing the type of the task. The possible values are: “Sampler”, “Controller” and “Actuator”.

The dependencies/precedencies between tasks are established through the **TaskDependency** elements. Each **TaskDependency** element defines a dependency between two tasks, through the following attributes:

`ProducerTaskId`	ID of the task which produces some data that another task waits for, or which must finish execution before another task can start, i.e. producer task.
`ConsumerTaskId`	ID of the task which depends on the producer task.

Processing elements

The configurations for the processing elements are grouped under the **ProcessingElements** XML node. A multicore system must contain at least one processing element in order to be validated. All the configuration data for a PE is contained within a **ProcessingElement** node. The attributes of a **ProcessingElement** are the following:

`Id`	Number that uniquely identifies a PE.
`Name`	String representing the name of a PE.
`ClockPeriod`	Number representing the internal clock period of a PE, expressed in seconds.
`SchedulingPolicy`	String representing the type of the scheduling policy configured on the PE. The possible values are: “FP”, “SC” and “EDF” and they correspond to fixed-priority, static-cyclic and earliest deadline first scheduling policies respectively.

If a static cyclic scheduler is configured to run on a PE, then the **ProcessingElement** node must include the definition of the static table used by the task scheduler. A static table can be defined within a **SchedulingTable** node. It has one attribute, i.e. `Period`, which represents the hyper-period of the static-cyclic scheduler expressed in seconds. The actual entries of the table are defined as a list of **Entry** elements. An **Entry** element has the following attributes:

`Time`	Number representing the time, relative to the period of the scheduler, when a task should start running. The time is expressed in seconds.
`Duration`	Number representing the amount of time, expressed in seconds, for which a task should be running.
`TaskId`	ID of the task that the entry corresponds to.
`ApplicationId`	ID of the application that owns the task.

The mapping of tasks to PEs is done within the **ProcessingElement** node in the **HostedTasks** section. We can map any number of tasks to a PE. Each mapping is registered through a **HostedTask** element which has the following attributes:

`Id`	ID of the task that is to be executed on the PE.
`ApplicationId`	ID of the application that owns the task.

Time-triggered network

The configuration of the TT network is contained in the **TTNetwork** section. By design the internal clocks of all network elements are synchronized and have the same clock rates. The clock period of the network is set through the `ClockPeriod` attribute and it is to be expressed in seconds.

As described in the architecture model, the TTNoC is made of linked network components, which can be either network interfaces or network switches. In the configuration file there is no difference between the two. They are both configured as network components. SIMULATOR categorizes them based on their IDs. If the ID of a network component matches the ID of a PE, then the parser creates a network interface and attaches it to the PE, or it creates a network switch otherwise. The network components are defined in the **NetworkComponents** section, within the **NetworkCompenent** elements. A network component has the following attributes:

`Id`	Number that uniquely identifies a network component. If it matches the ID of a PE, the SIMULATOR instantiates a network interface, or a network switch otherwise.
`Name`	String representing the name of the network component.

Each network component must include a timetable based on which the TT frames routed through it are being transmitted. The timetable is defined in the **SchedulingTable** element. It has one attribute, i.e. `Period`, which represents the period of the TT transmission expressed in seconds. The actual entries in the table are defined as a list of **Entry** elements, each of these elements having the following attributes:

`Time`	Number representing the time, expressed in seconds, relative to the period of the TT scheduler, when the transmission of a TT frame should start.
`Duration`	Number representing the amount of time, expressed in seconds, allocated to the transmission of a TT frame.
`FrameId`	ID of the frame that the entry corresponds to.

The full-duplex connections between the network components, i.e. the data flow links, are defined in the **DataFlowLinks** section. This section contains a list of **DataFlowLink** elements, each having the following attributes:

`FromNetwCompId`	ID of a network component that defines the data link.
`ToNetwCompId`	ID of the other network component that defines the data link.

Frames

The definitions of the frames handled by the TT network are contained within the **Frames** section in the **Frame** elements. The attributes that configure a frame are the following:

`Id`	Number that uniquely identifies a frame.
`Name`	String representing the name of the frame.
`AssociatedAppId`	ID of the application that the frame is associated with. By definition, a frame carries data packets between two tasks of the same application.
`SourceTaskId`	ID of the sender task.
`DestinationTaskId`	ID of the destination task.
`Size`	Size of the frames given in number of network clock cycles needed for a frame to be transmitted.
`Type`	String representing the type of the frame. It can be either “TT” (i.e. time-triggered) or “ET” (i.e. event-triggered).

`Priority`	Number representing the priority level of a frame. The higher the value, the higher the priority. This is an optional attribute and it is only used if the frame is of type "ET".

Each frame is assigned a route that it can follow through the network, route which is configured in the **DataFlowPath** node. The path is basically expressed as a list of network components IDs, contained within **NetwComp** elements. Note that the order of the elements in this list is very important. The first and the last network components in the list must be network interfaces corresponding to the source and the destination PEs.

3.3.3 Block Description File

The dynamics of the systems involved in a feedback control loop, i.e. the plant, the sampler, the controller and the actuator must be described in separate MATLAB function files. These system description MATLAB functions take no parameters and return a single value, which can be either a transfer-function model, a state-space model or a pole-zero gain model. As required by MATLAB, the names of the files that contain functions must match the names of the functions they contain.

An example of such a block description function is given in Listing 3-1. It returns a discrete-time state-space model of an LQG controller computed using the `lqgdesign` function from the JITTERBUG toolbox.

Listing 3-1 Example of a block description function

```
function C = Controller( )

s = tf('s');
o = 6.7;
P = o^2/(s^2-o^2);
R1c = 1/o;                          % Continuous-time input noise
R2 = 0.0001;                        % Discrete-time measurement noise
Qc = diag([1 1]);                   % Cost function

h = 0.030;                          % Sampling period
tau = 0.006;                        % Assumed input-output delay

C = lqgdesign(P,Qc,R1c,R2,h,tau);  % LQG controller
end
```

3.3.4 JITTERBUG Script

The SIMULATOR uses the timing information, i.e. sampling and input-output latencies, collected during the simulation of the control applications, to build JITTERBUG models which are evaluated in MATLAB. These models are saved as .m files, one for each control application. The name of an .m file includes the name of the control application that it corresponds to, followed by a timestamp, i.e. the time when the file was generated ([APPPLICATION_NAME]_[yyyyMMdd]_[hhmmss].m). We further refer to files that contain JITTERBUG models as JITTERBUG scripts.

A JITTERBUG script fully describes a feedback control system in terms of both timing and signal models (see section 2.3. for the description of these models), based on which it computes a quadratic performance criterion. An example of a generated JITTERBUG script is given in Listing 3-2. The logic implemented by a JITTERBUG script resumes to:

1. Configure the MATLAB search path to include the locations of the JITTERBUG toolbox and of the block description functions.
2. Initialize a new JITTERBUG system by calling `initjitterbug`. This function must be provided with the period of the system, i.e. the sampling period of the control application, and the granularity of the time-discretization. We consider the time granularity as being the smallest clock period of a PE that executes tasks from the studied application.
3. Construct the timing model. A timing node is added by calling the `addtimingnode` function. The first timing node is assigned the probability distribution of the sampling latency, whereas the second timing node is assigned the probability distribution of the input-output latency.
4. Construct the signal model. The process is expected to be a continuous system, therefore it will be added using the `addcontsys` function. This function will be passed as input also the expression of the performance criterion that is to be evaluated. The sampler, controller and actuator are assumed to be discrete systems and they are added using `adddiscsys` function.
5. Evaluate the performance criterion. This is achieved by calling the `calcdynamics` and `calccost` functions.

Listing 3-2 Example of a JITTERBUG script

```
addpath('C:\Jitterbug');                    % Path to Jitterbug toolbox
addpath('C:\BlockDefinitions\Plant');       % Path to the plant description function
addpath('C:\BlockDefinitions\Sampler');     % Path to the sampler description function
addpath('C:\BlockDefinitions\Controller'); % Path to the controller description function
addpath('C:\BlockDefinitions\Actuator');    % Path to the actuator description function

h = 0.010;          % Sampling period
```

```
dt = 0.001;         % Time-grain

P = Process();      % Plant
S = Sampler();      % Sampler
C = Controller();   % Controller
A = Actuator();     % Actuator

sampL = [1 0 0 0 0 0 0 0 0 0 0];          % Sampling latency probability distribution
ioL = [0 0 0 0 0 0 0 0 0.01 0 0.33 0.66]; % Input-output latency probability distribution

N = initjitterbug(dt, h);           % Initialize Jitterbug

% timing model definition

N = addtimingnode(N, 1, sampL, 2);  % Add node 1 (the periodic node)
N = addtimingnode(N, 2, ioL, 3);    % Add node 2
N = addtimingnode(N, 3, [1], 4);    % Add node 3 - no delay
N = addtimingnode(N, 4);            % Add node 4

% signal model definition

N = addcontsys(N, 1, P, 4, diag([1 1]), 1/20, 0.0001); % Add the plant
N = adddiscsys(N, 2, S, 1, 2);      % Add the sampler (S) to timing node 2
N = adddiscsys(N, 3, C, 2, 3);      % Add the controller (C) to timing node 3
N = adddiscsys(N, 4, A, 3, 4);      % Add the actuator (A) to timing node 4

%cost computation

N = calcdynamics(N);                % Calculate the internal dynamics
J = calccost(N);                    % Calculate the cost
```

3.4 User Interface Design and Description

The SIMULATOR displays the results of a simulation in an output window, as depicted in Figure 3.8.

The output window is basically divided into two regions. The upper region displays a Gantt chart that captures the allocation of the computational resources among tasks and the transmission of frames between the network components during the entire simulation time. The lower region displays the simulation summaries and the quadratic costs calculated for every control application.

The Gantt chart, in the upper region, includes timelines for every PE and for every dataflow link in the TTNoC. Each timeline is labelled with the name of a PE, or with the names of the end nodes of a dataflow link, separated by an arrow. The rectangles displayed on the timelines represent the time intervals when frame instances are being transferred or when task instances execute. All rectangles are color coded. There are different colors assigned to different frames and all the instances of a frame share the same color. Similarly, there are different colors assigned to different applications and

all the task instances of a control application share the same color. The association between colors and control applications, or frames, is given in the legend from the right-hand side of the chart. Every rectangle in the chart is also assigned an identifier, displayed inside the rectangle. This identifier can be a number representing the instance number of a frame, or it can be a string representing the name and the instance number of a task.

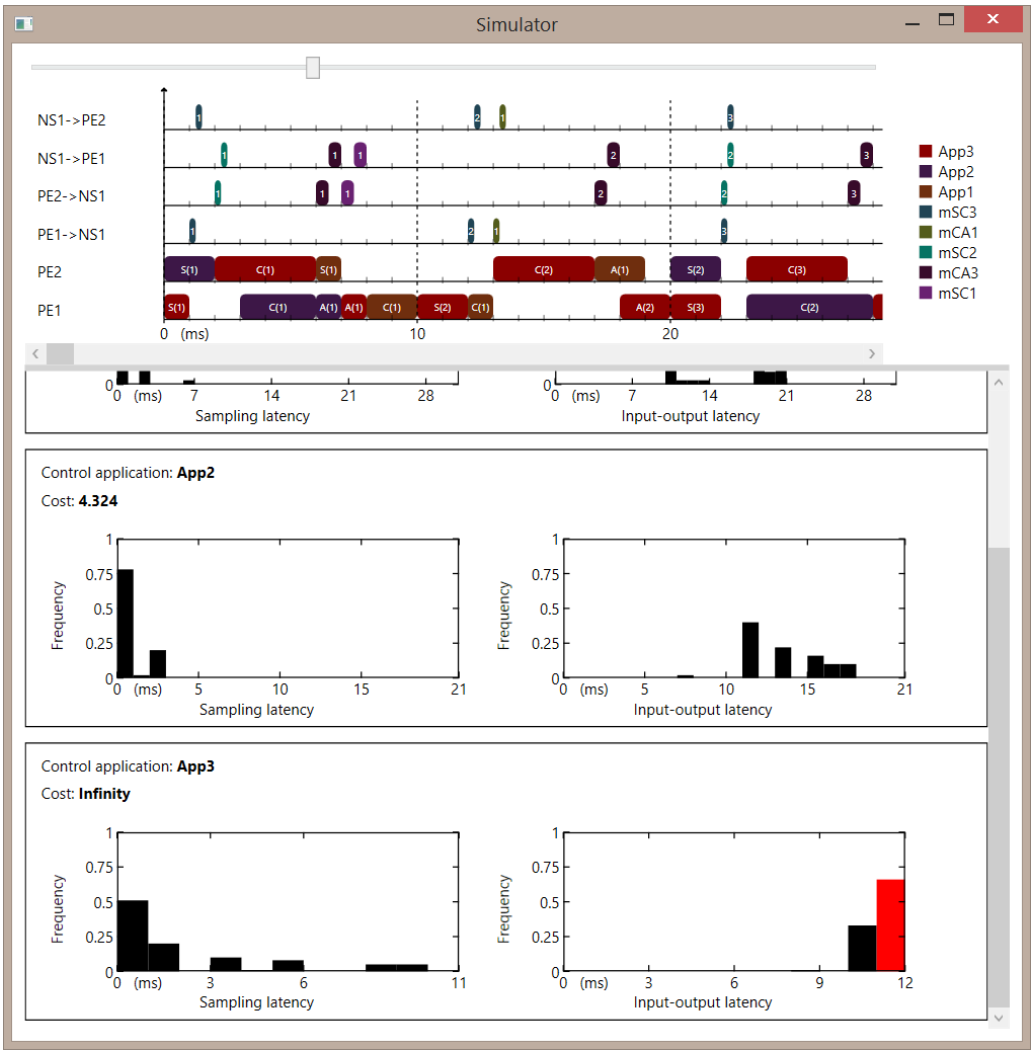


Figure 3.8 Output window

The lower part of the output window accommodates a list of simulation summaries for each control application. A simulation summary displays the control performance of a control application along with two normalized histograms depicting the input-output

and the sampling latency distributions, as they result from the simulation of the system. The width of the bins in each histogram is equal to the smallest clock period among the clock periods of the PEs that execute tasks owned by the application that the histogram corresponds to. As the control applications handled by the SIMULATOR are considered to have hard deadlines, the latencies in their execution must not exceed their periods. This means that the last bin in a histogram should correspond to the period of the application. This is valid in all the cases where no deadline misses are encountered during simulation. If there are deadline misses though, we add an extra bin, marked in red, which corresponds to a delay that is one clock cycle longer than the application period.

The output window depicted in Figure 3.8 displays the results of the simulation of three control applications, i.e. “App1”, “App2” and “App3”, on a multicore system comprising two processing elements, i.e. “PE1” and “PE2”, connected through a network switch, i.e. “NS1”. There are two dataflow links in the network: one between the network interface of “PE1” and the network switch “NS1”, and another one between the network interface of “PE2” and the network switch “NS1”. The TT network is configured to handle five frames, i.e. “mSC3”, “mCA1”, “mSC2”, “mCA3” and “mSC1”. Considering this system configuration, the Gantt chart includes four timelines for the two dataflow links, one timeline for each direction, and two timelines for the PEs. We give now a few examples of how the results should be interpreted. The bottom timeline in the Gantt chart indicates that job 1 of task “S” of application “App3” starts to execute on the processing element “PE1” at moment 0 and finishes its execution after 1ms. A situation of preemption is shown on the same timeline at time t = 10ms when the instance 1 of task “C” of application “App1” is preempted for 2ms by the second instance of task “S” of application “App3”. The transmission of the first instance of frame “mSC3” from “PE1” to “NS1” starts at t=1ms and it ends at t=1.25ms. The simulation summary for application “App3” shows that the performance criterion for this application evaluates to infinity, meaning that the plant controlled by application “App3” is unstable in the current setup. We can see that in about 66% of the cases the controller misses to deliver the control signal before deadline, this being the reason for the instability.

Even though the SIMULATOR handles also generic applications that share the computational resources with the control applications, only the data collected from the control applications is displayed in the output window. This was a design decision meant to avoid cluttering the view.

4. Design Optimizations

In this chapter we analyse the impact that different design decisions concerning the hardware configuration, scheduling policies, task mapping, frame routing or frame scheduling, have on the performance of the control applications.

4.1 Case Study: Three Inverted Pendulums

We chose to analyze the performance of three control applications running on a multicore system. The control applications are responsible for simultaneously stabilizing three inverted pendulums. This case study is inspired from [6].

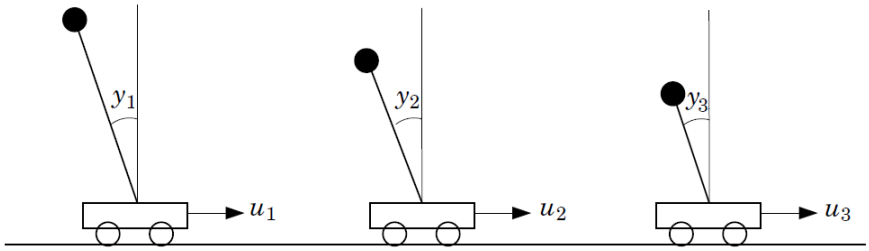


Figure 4.1 Three inverted pendulums that are to be simultaneously stabilized using a single multicore system. From [6].

The three inverted pendulums have different lengths (l) of 0.22m, 0.1m and 0.025m, and natural frequencies (ω_0) of approximately 6.7rad/s, 10rad/s and 20rad/s respectively ($\omega_0 = \sqrt{g/l}$). Each of the three pendulums can be described by the linear transfer function

$$G(s) = \frac{\omega_0^2}{s^2 - \omega_0^2} .$$

The pendulums are considered to be disturbed by continuous-time input noise with the variance $R_1 = 1/\omega_0$ and discrete-time measurement noise with the variance $R_2 = 10^{-4}$.

The control applications implement discrete-time LQG controllers which are designed to minimize the continuous-time cost function

$$J = \lim_{T\to\infty} \int_0^T \left(y^2(t) + u^2(t)\right)dt$$

We use this cost function as the control performance evaluation criterion. The cost matrix in this case is $Q = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.

The nominal sampling periods (h) of the controllers are chosen such that $0.2 < \omega_o h < 0.6$ [3]. Table 4-1 presents the configuration of the control applications, including their decomposition in tasks. The tasks are imposed the following precedence constraint: the controller tasks depend on the sampler tasks and the actuator tasks depend on the controller tasks.

Table 4-1 Configuration of the control applications

Application	Sampling period [ms]	Task	Task type	BCET [clock cycles]	WCET [clock cycles]
A_1	30	τ_S	Sampler	1	1
		τ_C	Controller	2	3
		τ_A	Actuator	1	2
A_2	20	τ_S	Sampler	1	1
		τ_C	Controller	2	4
		τ_A	Actuator	1	2
A_3	10	τ_S	Sampler	1	1
		τ_C	Controller	2	3
		τ_A	Actuator	1	1

In the ideal case where the control applications execute at their nominal sampling periods and they encounter no sampling jitter, no input-output latency or no input-output jitter, the cost function evaluates to 3.206, 3.229 and 3.271, for A_1, A_2 and A_3 respectively. In more realistic scenarios the latencies cannot be avoided, therefore the costs are expected to increase. We analyse next a few of such scenarios.

In the cases where RM scheduling policy is employed we use the priorities defined in in Table 4-2.

Table 4-2 RM priority assignment

Application	A_1			A_2			A_3		
Task	τ_S	τ_C	τ_A	τ_S	τ_C	τ_A	τ_S	τ_C	τ_A
Priority	1	1	1	2	2	2	3	3	3

4.1.1 Single Processing Element Configuration

The simplest hardware configuration includes only one processing element, with all tasks mapped to it.

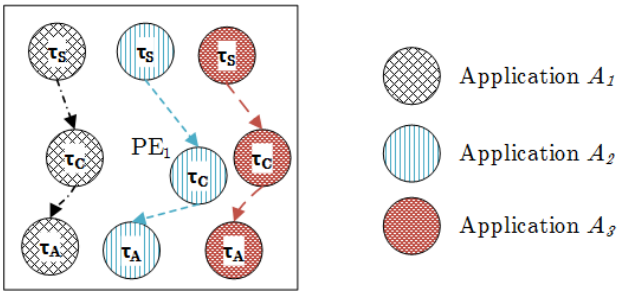


Figure 4.2 Single PE configuration

The processing element is set to run with a clock period of 1ms. Considering the WCET for the tasks given in Table 4-1, it can be determined that the processor is overloaded, i.e. the processor utilization is 105%, and therefore we expect that not all tasks finish execution before their deadlines (the processor utilization represents the fraction of processor time spent in the execution of the task set). This configuration is interesting for analyzing how the control applications handle the deadline miss situations.

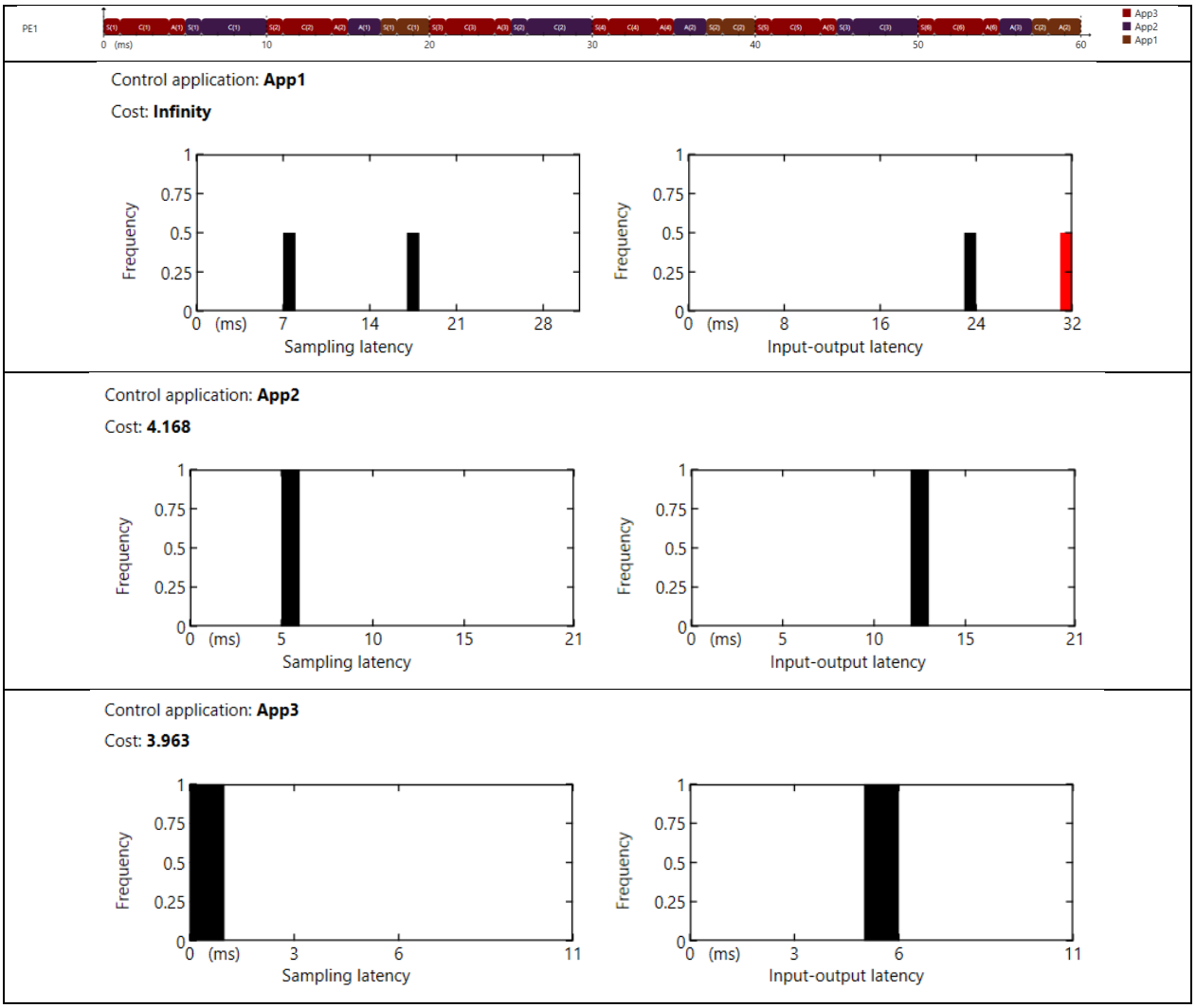
Case 1-1: RM Scheduling with Fixed Task Execution Times

In this case we employ a rate-monotonic scheduling policy and we consider the computation time of each task to be equal to its WCET. The simulation is set to run for 60ms, which corresponds to the hyper-period of the schedule.

According to the results of the simulation, depicted below, the cost function for control application A_1 evaluates to infinity. This indicates that application A_1 fails to stabilize its inverted pendulum. The reason for the instability is that in half of the cases, application A_1 does not finish before deadline. This results in an actual sampling period of 60ms (i.e. twice as big as the nominal sampling period), which the controller is not designed to handle. RM scheduling implies that the tasks of application A_1 have

the lowest priority, causing them to be preempted by the higher priority tasks of A_2 and A_3.

Table 4-3 Simulation results for Case 1-1



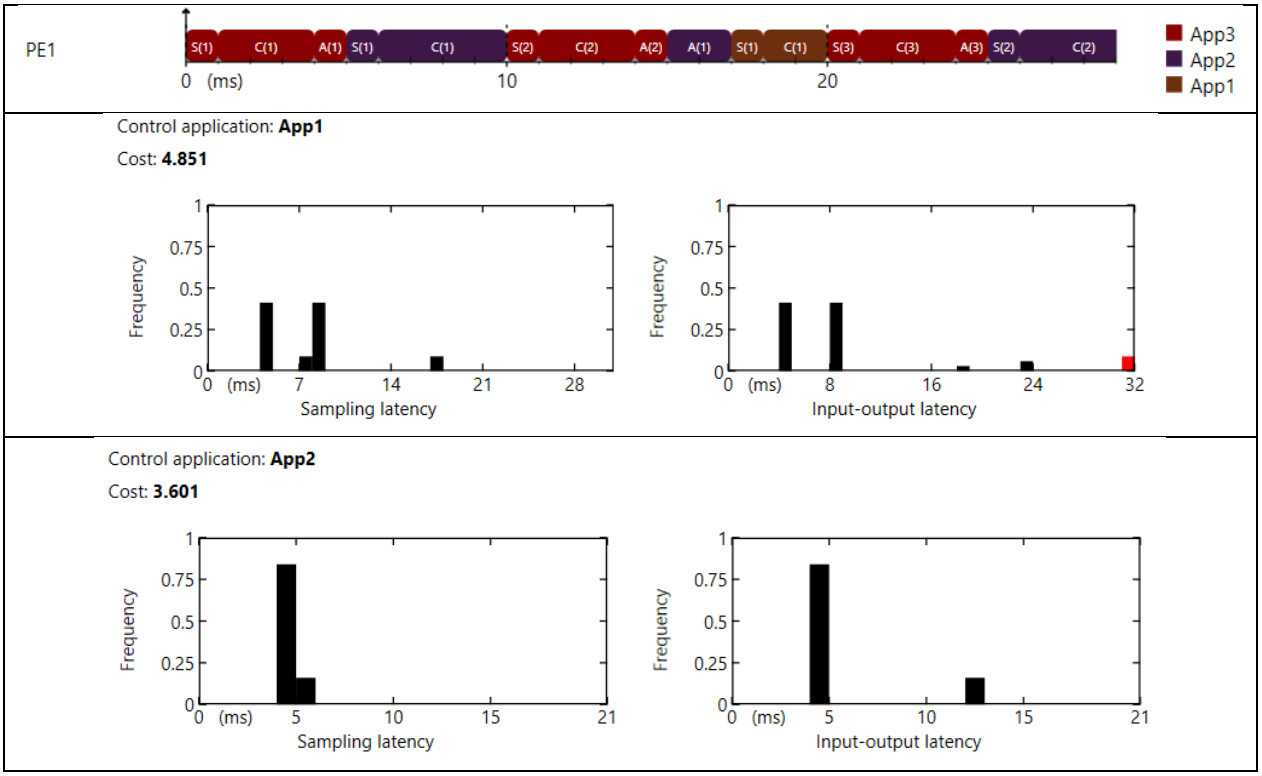
We have to note that, given the fact that application A_3 has the highest priority, it is not affected by any scheduling delays, and therefore it achieves a good control performance. Application A_2, on the other hand, encounters a high input-output latency of 12ms, compared to its total computation time of 7ms, which degrades the QoC. The constant sampling latency does not have any influence on the control performance of application A_2 and it could be removed completely by configuring the phase of the application to 5ms.

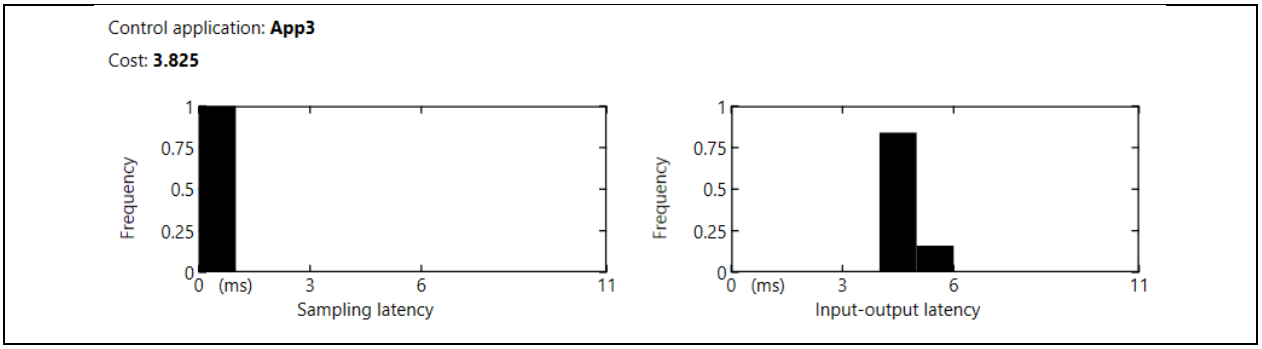
Case 1-2: RM Scheduling with Random Task Execution Times

In this case we employ a rate-monotonic scheduling policy and the tasks are configured to have random execution times. The simulation time was increased to 1s.

Using random execution times, between BCET and WCET, for all tasks, leads to decreased processor utilization. This means that it is likely to encounter less deadline misses for application A_1. The simulation results confirm this expectation. Application A_1 encounters less deadline misses than before and it manages to stabilize the pendulum, even though the control performance is poor (i.e. the cost, 4.851, is relatively high). The costs for applications A_2 and A_3 are better than in the previous case. This is because of their reduced input-output latency determined by the reduced computation times.

Table 4-4 Simulation results for Case 1-2



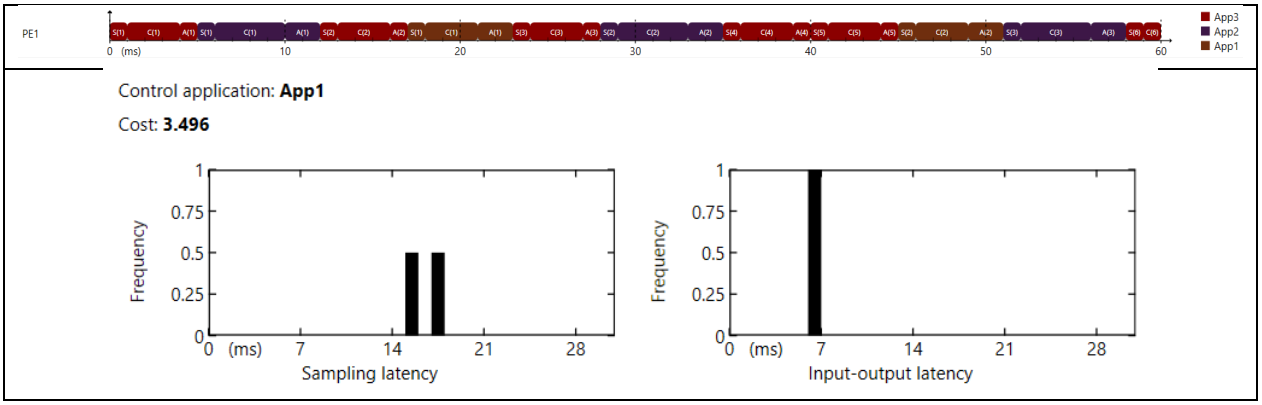


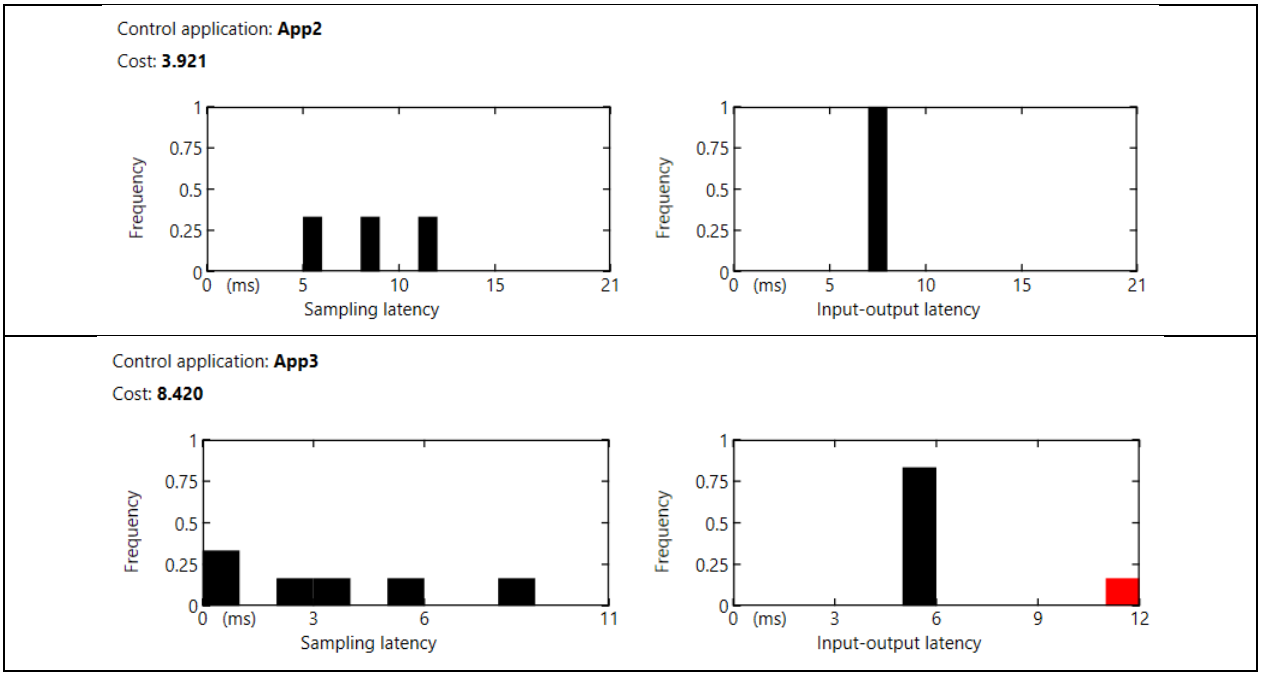
Case 1-3: EDF Scheduling with Fixed Task Execution Times

In this case we analyse the behaviour of the three control applications under EDF scheduling. The tasks are considered to have constant computation times equal to their WCETs. The simulation was run for a period of 60ms.

Compared to Case 1-1, where we used RM scheduling, the results obtained in this case are consistently better, in the sense that all three controllers manage to stabilize the plants they are associated with. In this case application A_3 misses its deadline in one sixth of the cases and this causes a significant degradation of its QoC. By analysing the task execution timing diagram in Table 4-5, we observe that the 6th instance of application A_3 does not finish in time. This is caused by the fact that at time t=50ms when the jobs of application A_3 are released they are assigned the same priority as of all the other jobs already in the queue, all having the same absolute deadline d=60ms, and therefore the jobs of application A_3 have to wait for the other jobs to finish.

Table 4-5 Simulation results for Case 1-3



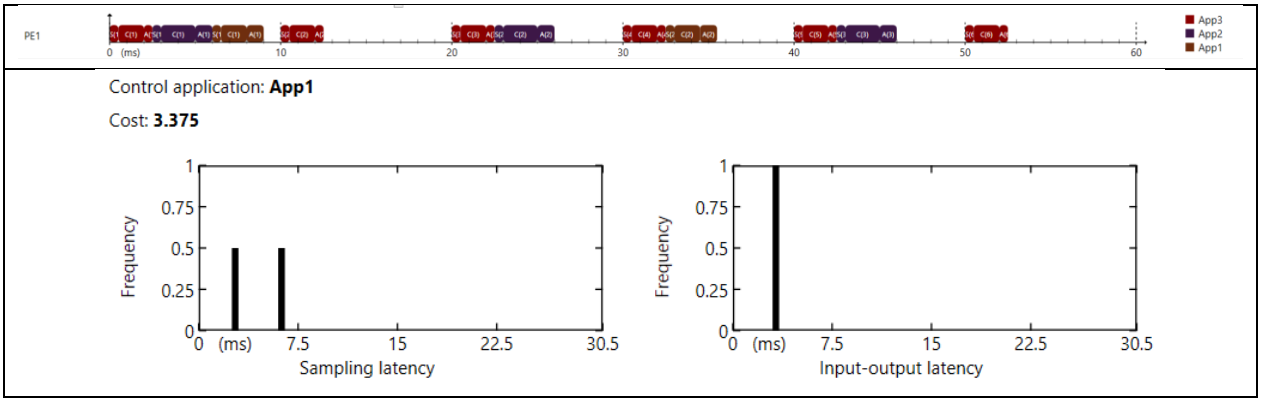


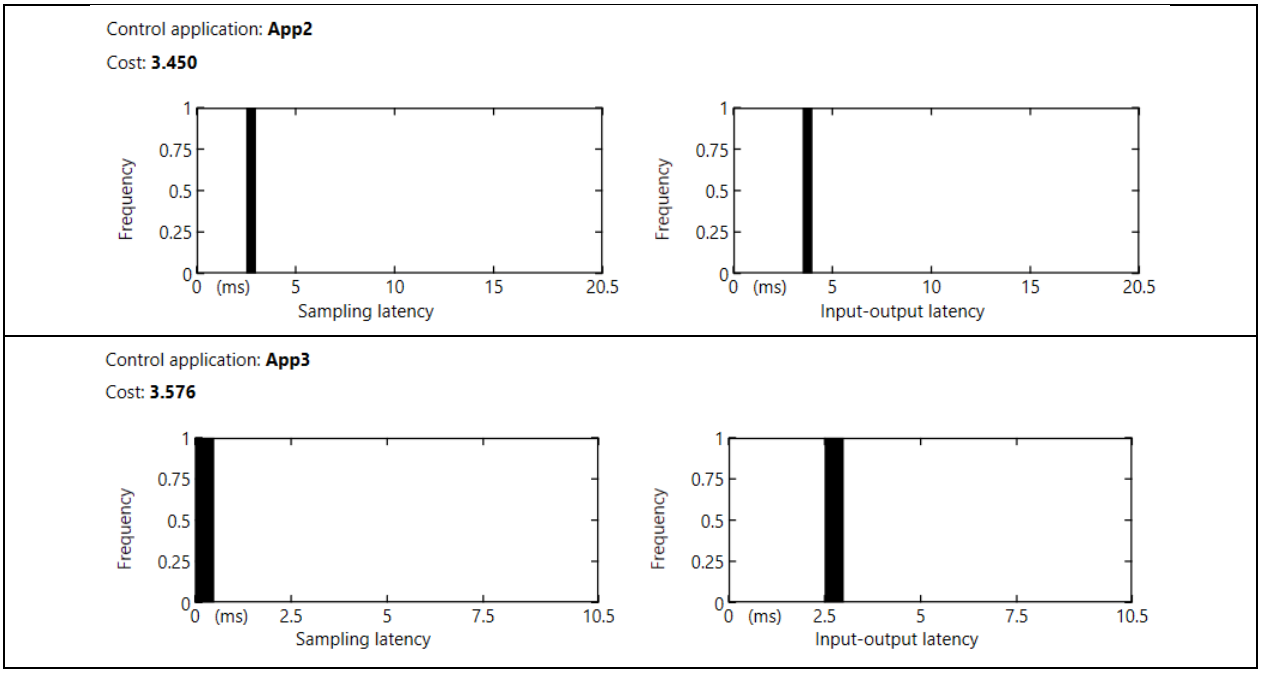
Case 1-4: EDF Scheduling with Fixed Task Execution Times and 2x Clock Rate

In this case we employ EDF scheduling with fixed task execution times and we double the clock rate of the processing element, i.e. the clock period is 500ns. This results in halved WCETs for all the tasks. The simulation was run for a period of 60ms and the results are displayed below.

All control applications finish in time and their control performance is improved.

Table 4-6 Simulation results for Case 1-4





4.1.2 Two Processing Elements Configuration

Using a faster processor to reach a desired control performance, i.e. to minimize the total cost, is not always an option. The alternative is to use a multicore system instead.

In this section we consider a hardware configuration that includes two processing elements connected through a single dataflow link. Each processing element runs with a clock period of 1ms. The TTNoC is configured to operate with a transmission clock period of 250ns.

In our analysis we consider two possible ways of mapping the tasks to the processing elements. The first task-to-PE mapping is depicted in Figure 4.3. It involves mapping the sampler tasks to one processing element, while the remaining tasks are mapped to the second processing element. Such a mapping may be appropriate in those cases where a specialized processing element is used to process the measurement data.

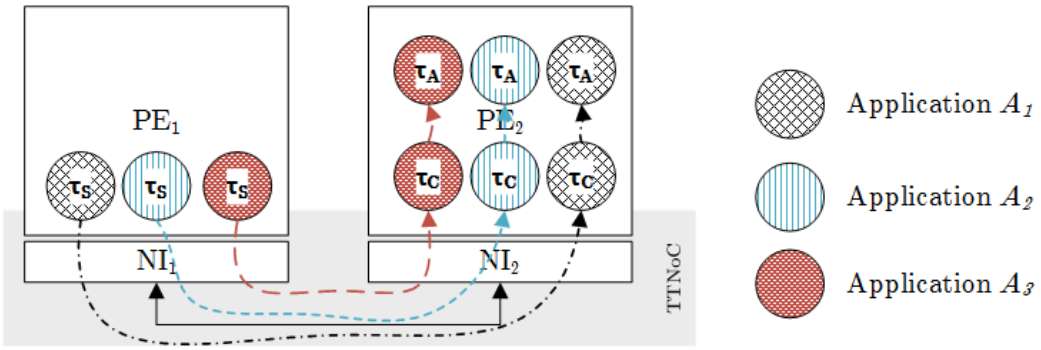


Figure 4.3 Task mapping 1: map the sampler tasks to PE1 and the rest of the tasks to PE2

The definitions of the frames that are used to transfer data between the tasks are given in Table 4-7.

Table 4-7 Frames corresponding to task mapping 1

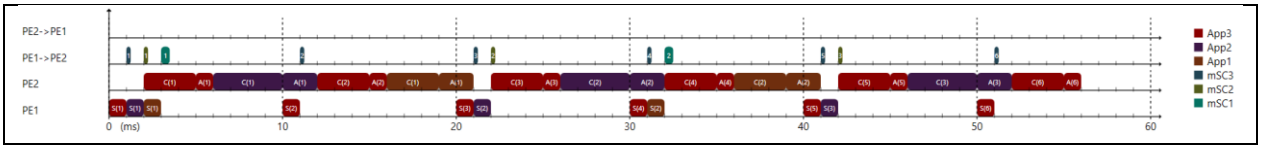
Frame	Associated application	Source task	Destination task	Transmission time [ms]	Priority	Route
mSC_1	A_1	τ_S	τ_C	0.5	1	N_1-N_2
mSC_2	A_2	τ_S	τ_C	0.25	2	N_1-N_2
mSC_3	A_3	τ_S	τ_C	0.25	3	N_1-N_2

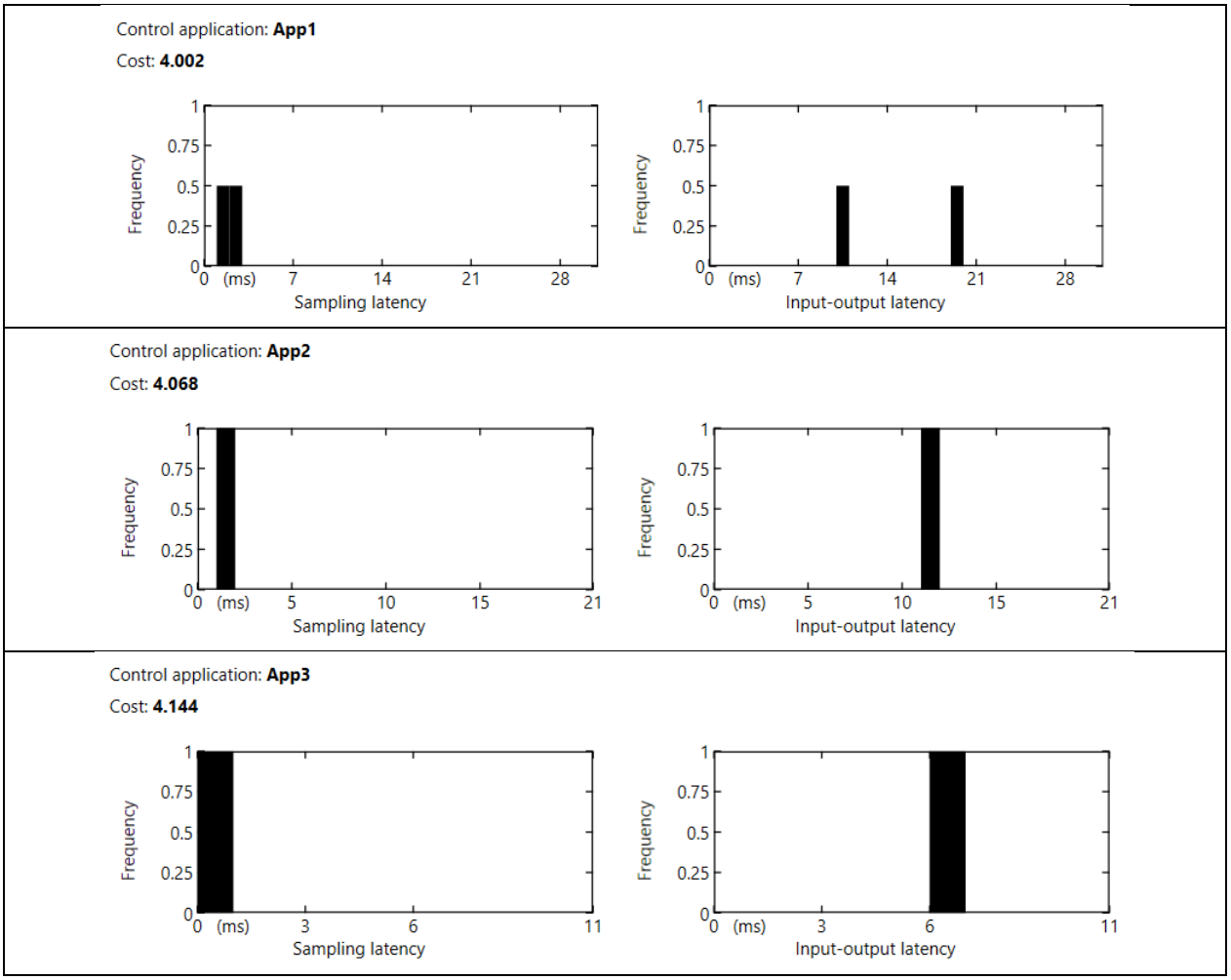
Case 2-1: Task Mapping 1 - RM Scheduling and ET Frames

In this case we simulate the system configuration given in Figure 4.3. Both processing elements employ rate-monotonic scheduling policies. All the frames in the network are event-triggered. The simulation is set to run for 60ms.

The results of the simulation show that all applications finish in time and they perform reasonably well. Application A_3, which is the most sensitive to delays, does not encounter any scheduling induced latencies. The only source for its increased input-output latency is the transmission time of frame mSC_3. Applications A_1 and A_2 experience increased input-output latencies because of preemption.

Table 4-8 Simulation results for Case 2-1

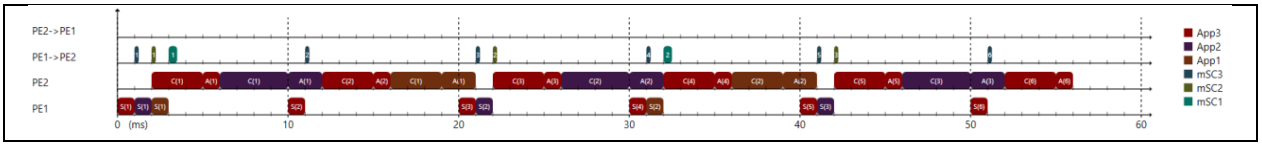


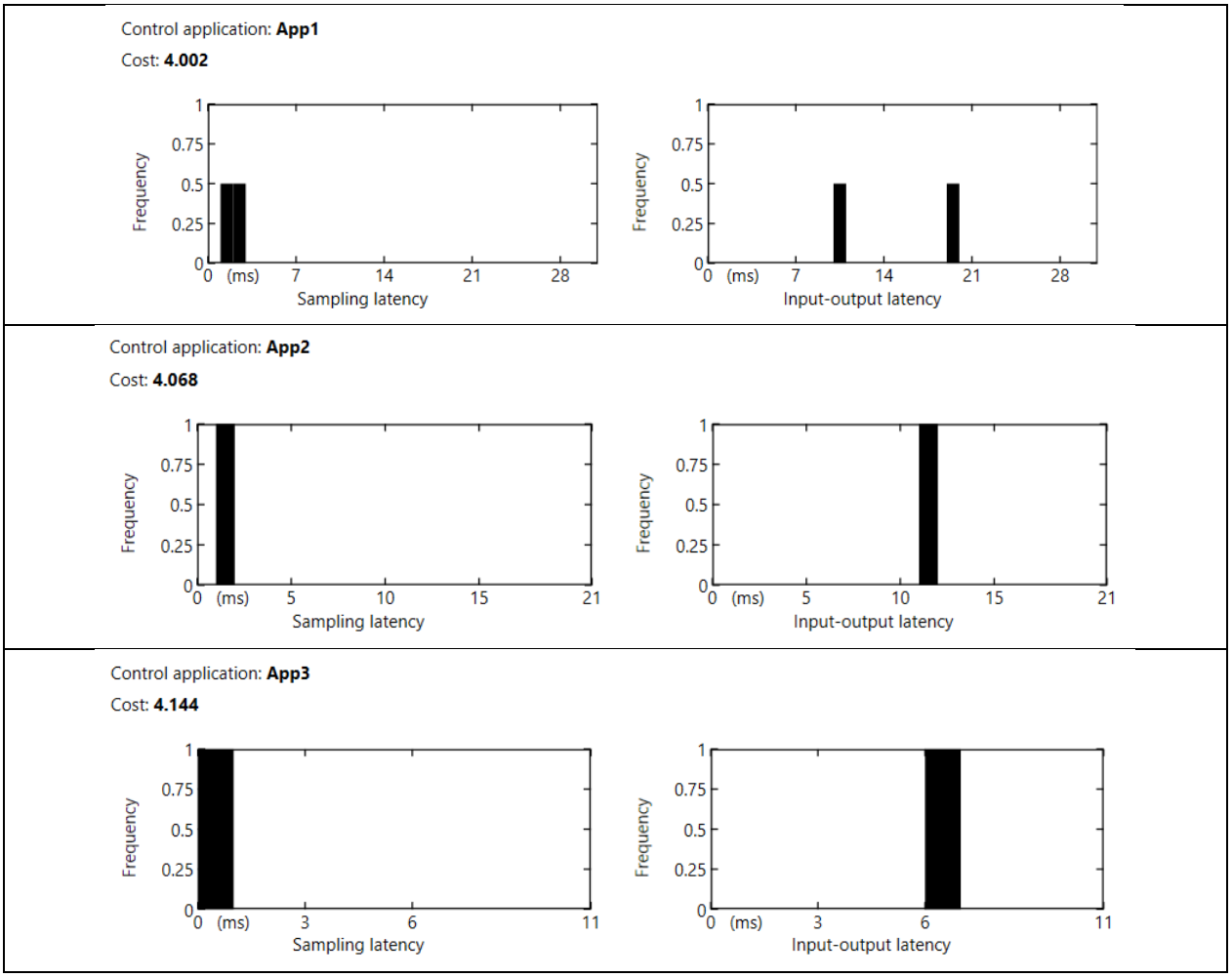


Case 2-2: Task Mapping 1 - EDF Scheduling and ET Frames

Under this specific system configuration, changing the scheduling policies from RM to EDF does not bring any improvement in the control performance. The results that we obtain are the same as in the previous case.

Table 4-9 Simulation results for Case 2-2

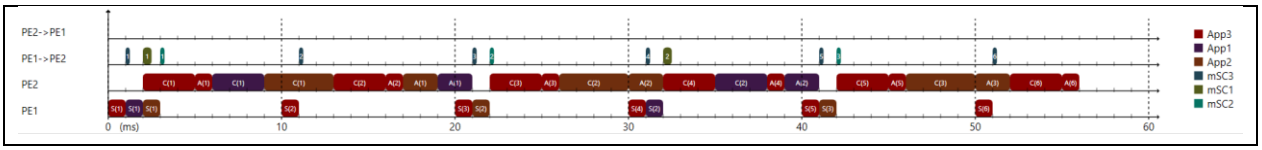


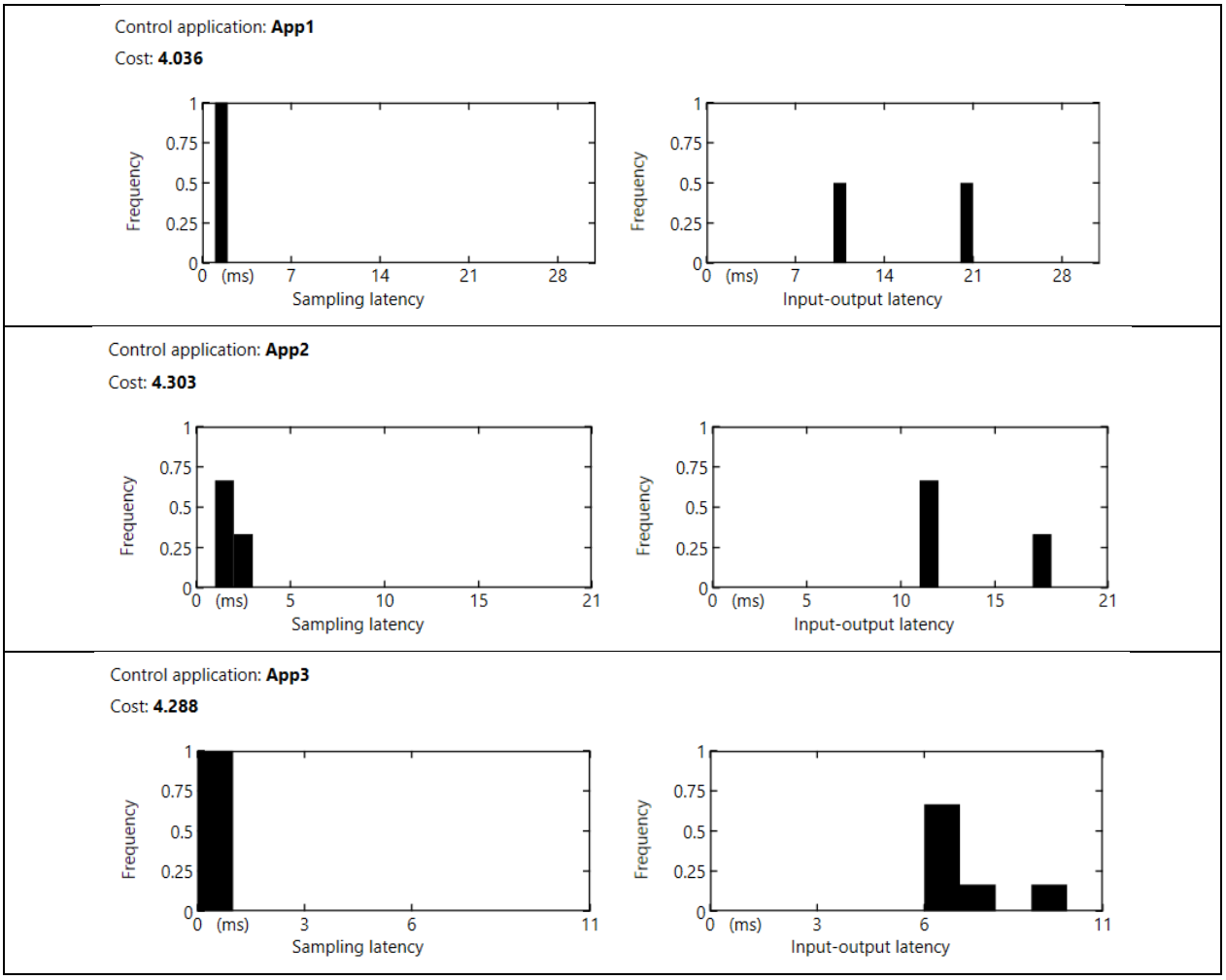


Case 2-3: Task Mapping 1 - Static-cyclic Scheduling and ET Frames

Designed to demonstrate the capabilities of the simulator, this case evaluates the performance of the control applications under static cyclic scheduling. The scheduling tables are given in Appendix B. The schedule was not optimized for performance. Instead it introduces more input-output jitter for all the three applications. This leads to increased costs, i.e. degraded control performance.

Table 4-10 Simulation results for Case 2-3





We consider now a different mapping of the tasks to the processing elements. This mapping is depicted in Figure 4.4.

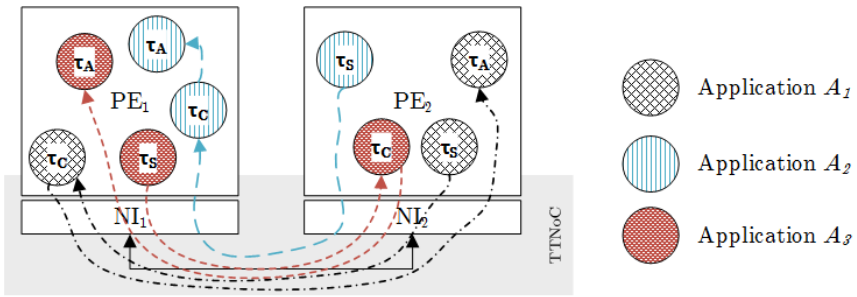


Figure 4.4 Task mapping 2

In order to ensure the data flow between the tasks we have to define new frames. They are described in Table 4-11.

Table 4-11 Frames corresponding to task mapping 2

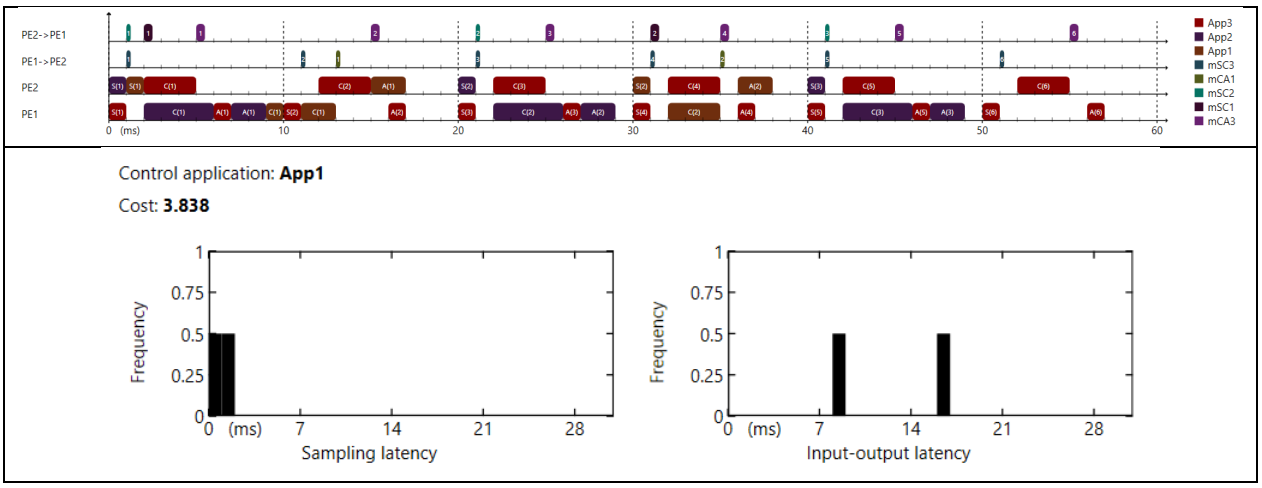
Frame	Associated application	Source task	Destination task	Transmission time [ms]	Priority	Route
mSC_1	A_1	τ_S	τ_C	0.5	1	N_2-N_1
mCA_1	A_1	τ_C	τ_A	0.25	1	N_1-N_2
mSC_2	A_2	τ_S	τ_C	0.25	2	N_2-N_1
mSC_3	A_3	τ_S	τ_C	0.25	3	N_1-N_2
mCA_3	A_3	τ_C	τ_A	0.5	3	N_2-N_1

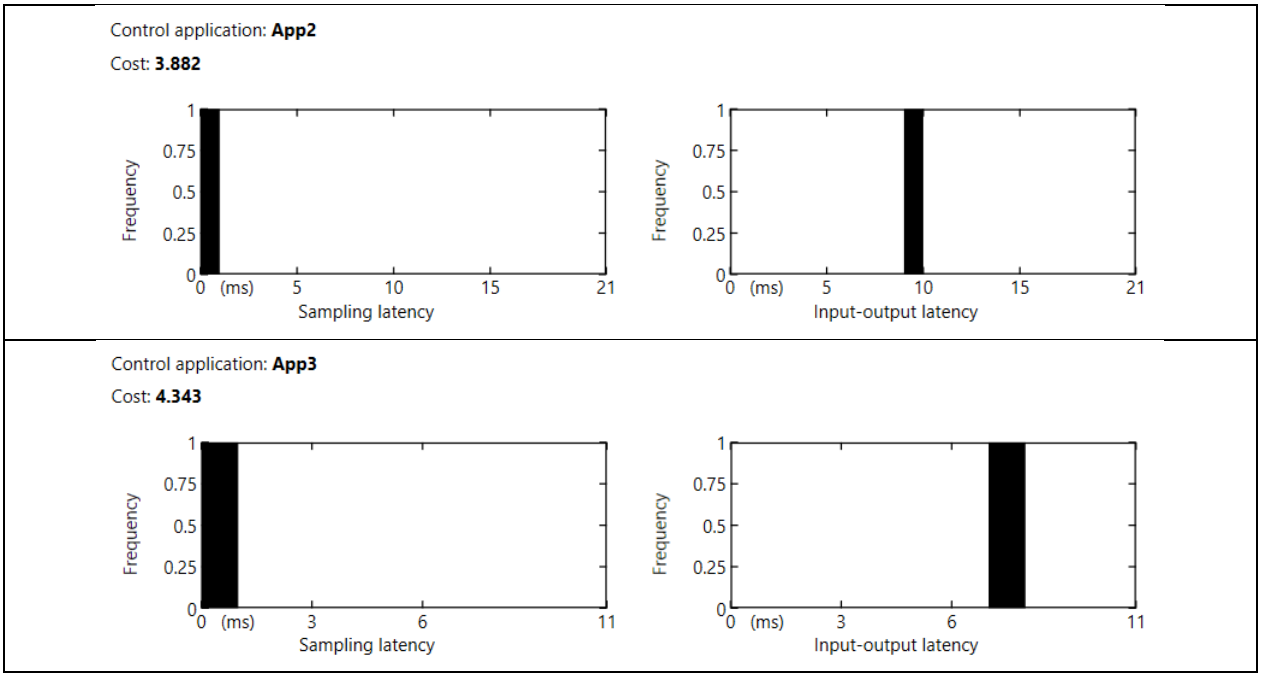
Case 2-4: Task Mapping 2 - RM Scheduling and ET Frames

The new system configuration was simulated employing RM scheduling policies on both processing elements. The frames were configured as event-triggered frames.

The results of the simulation show an improvement in the performance of applications A_1 and A_2. This is because their input-output latencies were decreased. Application A_3 performs worse because of the additional frame transmission times that contribute to an increased input-output latency.

Table 4-12 Simulation results for Case 2-4

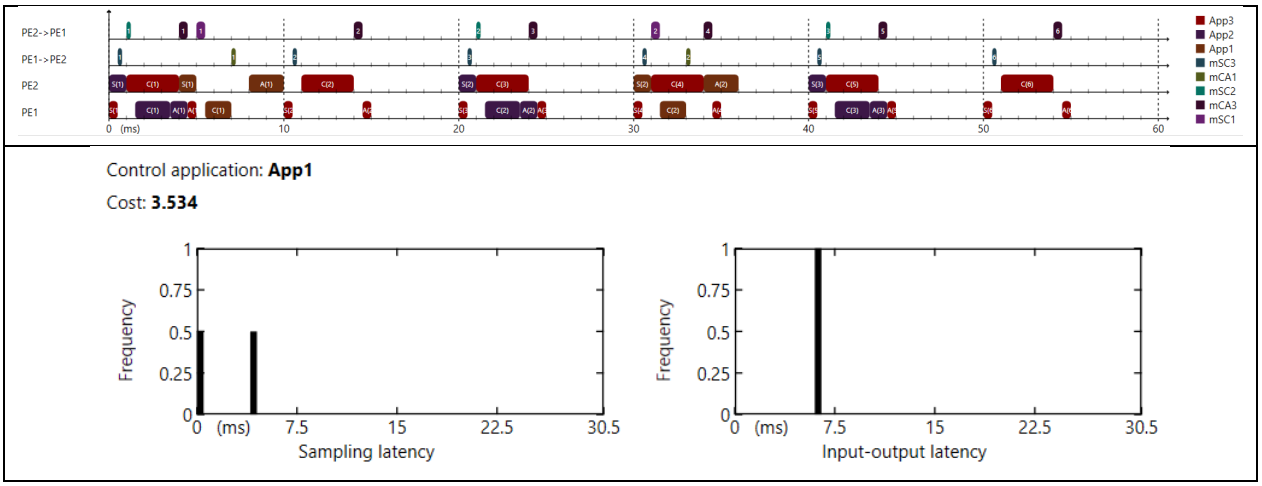


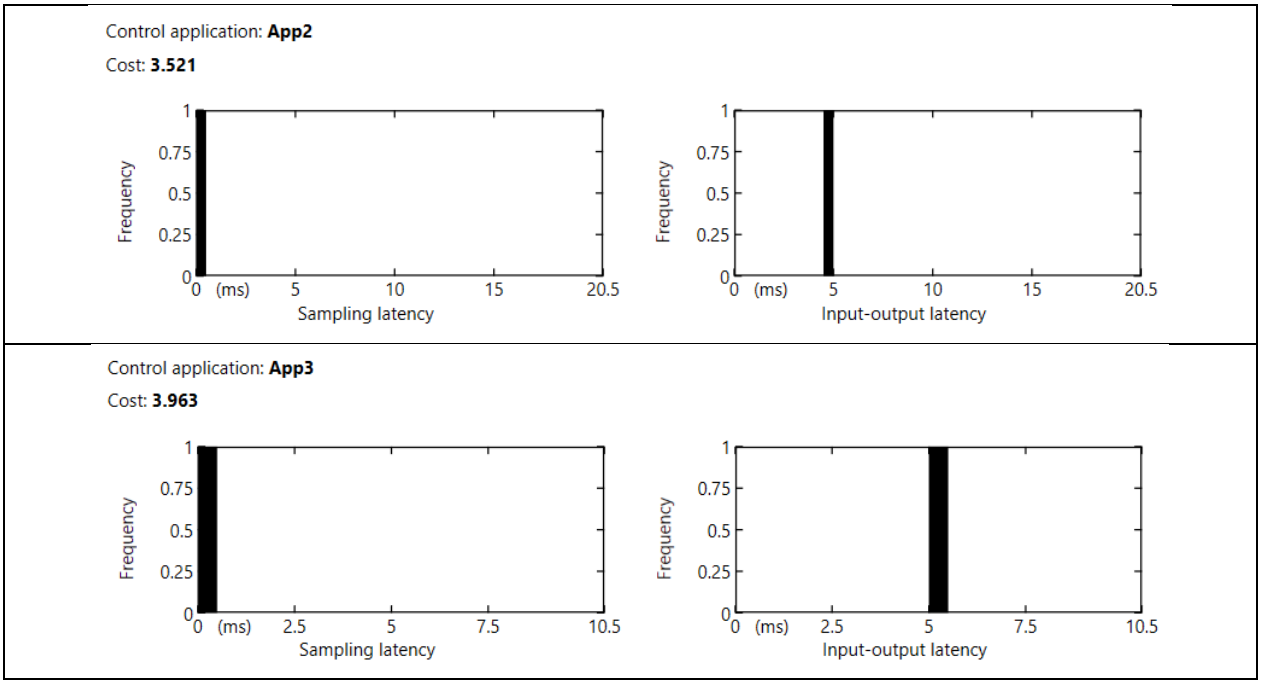


Case 2-5: Task Mapping 2 - RM Scheduling and ET Frames on a Heterogeneous System

In this simulation we cover the case of heterogeneous systems that contain processing elements that are clocked at different rates. We configure the clock period of PE_1 to 500ns, while PE_2 runs with a clock period of 1ms. The results show an improvement in the control performance of the applications because of the reduced input-output latencies.

Table 4-13 Simulation results for Case 2-5





4.1.3 Three Processing Elements Configuration

In this section we consider a hardware configuration that includes three processing elements, each of them having the clock period of 1ms. The TT network that connects them comprises three network switches. They are connected as depicted in Figure 4.5. The network is configured to run with a clock period of 250ns.

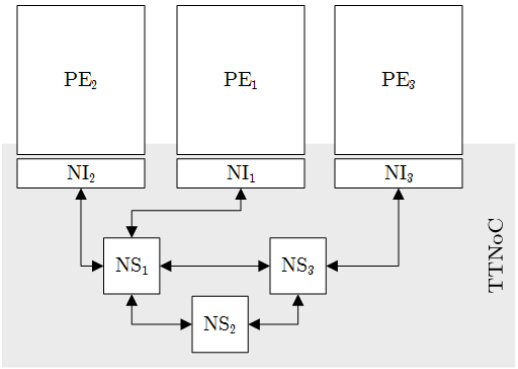


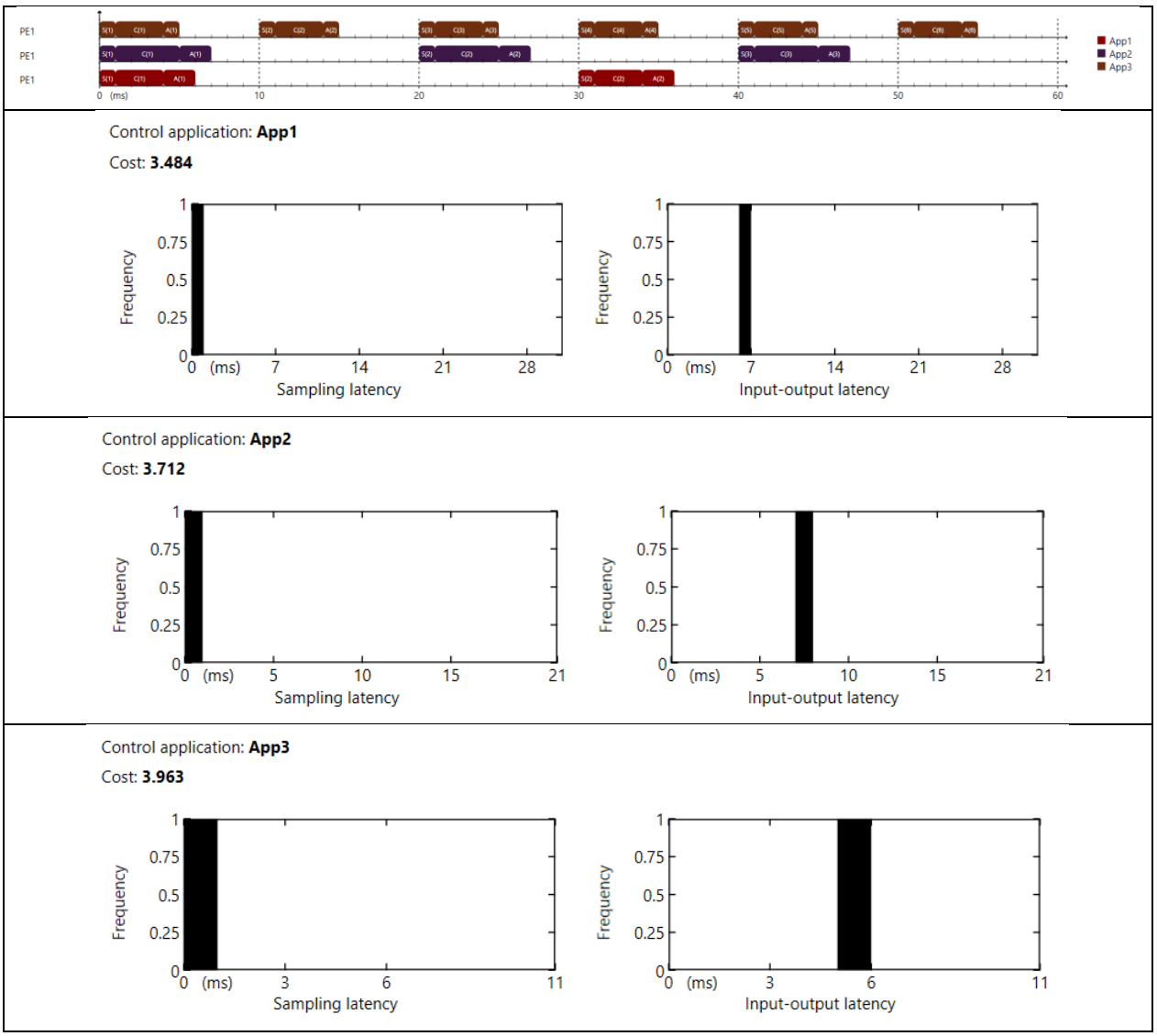
Figure 4.5 Multicore system comprising three processing elements

Case 3-1: Single Application per PE

We consider first the case in which the applications are mapped to different processing elements. As expected, the simulation results indicate an improved control performance. There are no scheduling- and no communication-induced latencies. This is the ideal mapping for this task set.

In a more realistic scenario, there would be more applications than the available processing elements, or the applications would contain tasks that could be executed simultaneously with other tasks, and therefore a different mapping would produce better results.

Table 4-14 Simulation results for Case 3-1



In the following cases we use the task-to-PE mapping depicted in Figure 4.6. We set the phase of application A_3 to 1ms. In order to determine how the frame routing influences the QoC, we plan to use two different routings for the frame mSC_3.

A first frame routing is depicted in Figure 4.6. The dashed lines in the picture indicate the dataflow paths followed by the frames that carry data between the connected tasks.

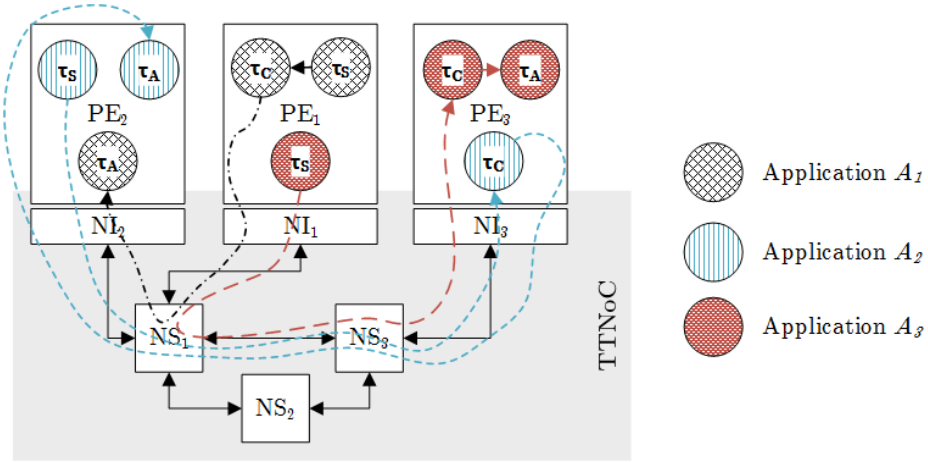


Figure 4.6 Frame Routing 1

Table 4-15 contains the configurations of the frames handled by the TT network. We have highlighted the table entry corresponding to frame mSC_3 as our further analysis focuses on the performance penalties that delays in the transmission of this frame bring.

Table 4-15 Frames configuration

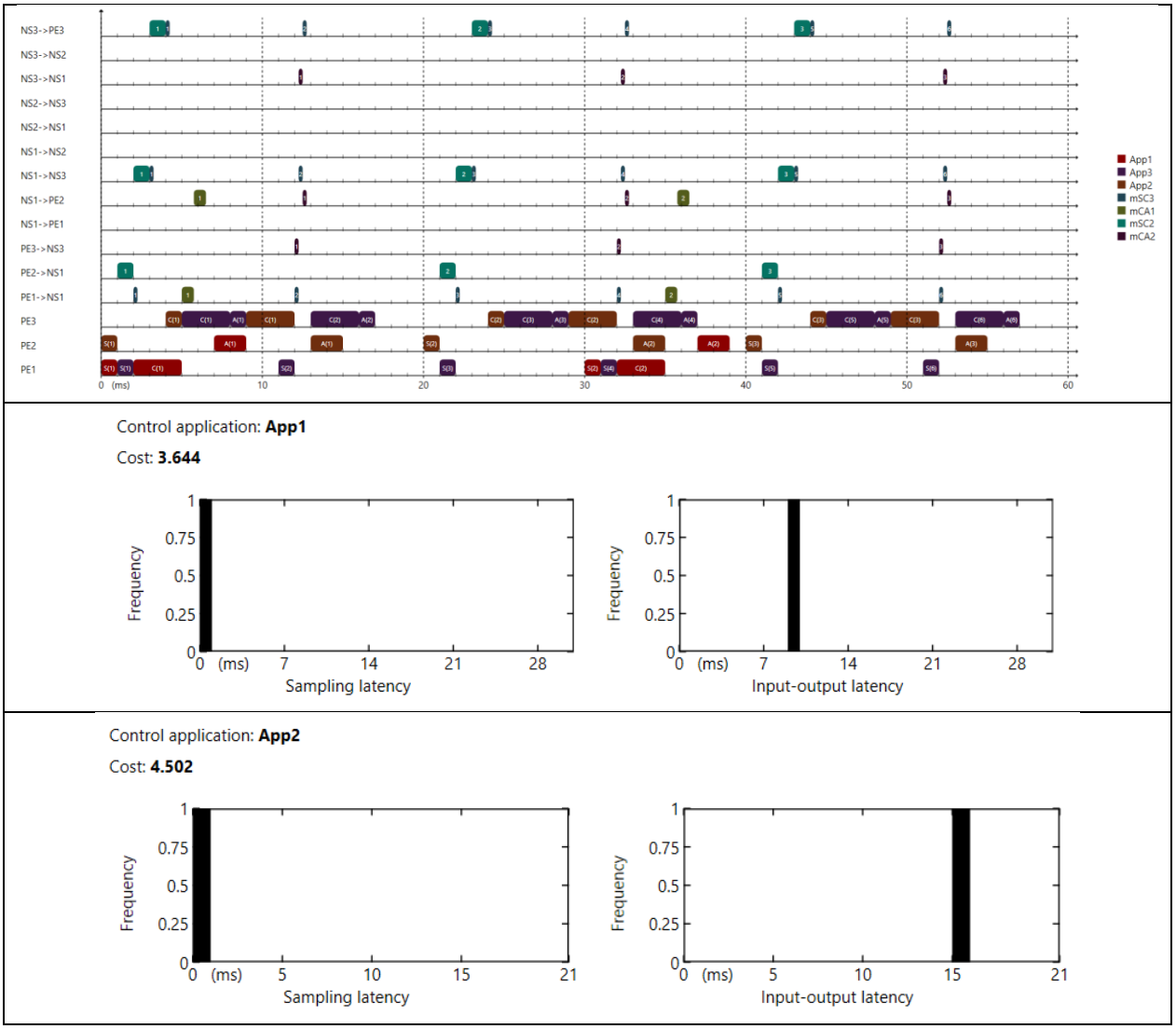
Frame	Associated application	Source task	Destination task	Sending time (ms)	Priority	Route
mCA_1	A_1	τ_C	τ_A	0.75	1	N_1-NS_1-N_2
mSC_3	A_3	τ_S	τ_C	0.25	2	N_1-NS_1-NS_3-N_3
mSC_2	A_2	τ_S	τ_C	1	1	N_2-NS_1-NS_3-N_3
mCA_2	A_2	τ_C	τ_A	0.25	1	N_3-NS_3-NS_1-N_2

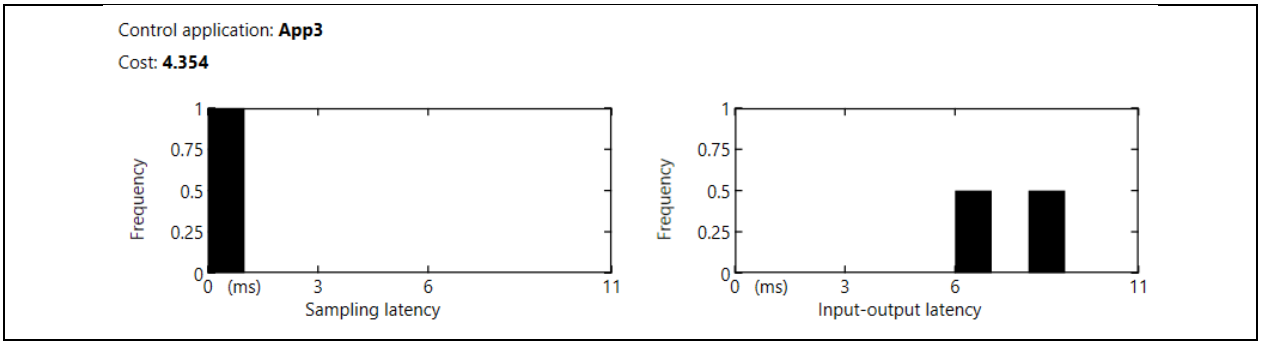
Case 3-2: Frame Routing 1 - RM scheduling and ET frames

In this case we employ rate-monotonic scheduling on the three PEs and we use event-triggered frames. The results for a 60ms simulation are given below.

We can notice the input-output jitter in the execution of application A_3, which appears to be caused by high transmission delays encountered by the first, the third and the fifth instances of frame mSC_3. By inspecting the frame transmission timing diagram in Table 4-16, we can see that frame mSC_3 is delayed by frame mSC_2. This happens because the two frames share a dataflow segment, consisting of dataflow links [NS_1-NS_3] and [NS_3-PE_3], which is entered by frame mSC_2 a little before frame mSC_3. As preemption is not supported in data transmission, frame mSC_3 has to wait for the other frame to be transmitted, even though it has a higher priority.

Table 4-16 Simulation results for Case 3-3





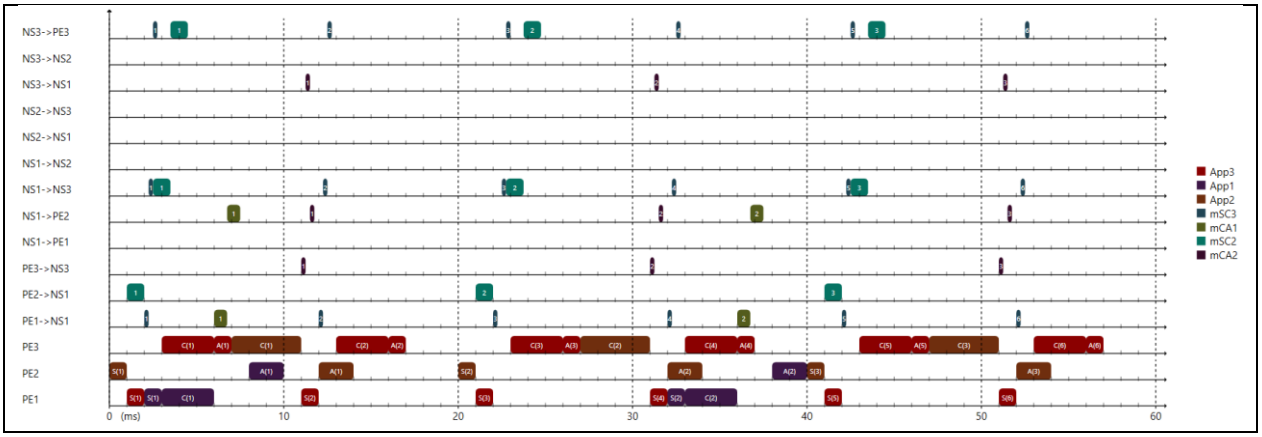
Case 3-3: Frame Routing 1 - Static-cyclic scheduling with TT and ET frames

As described in chapter 2, the TTNoC implements the timely block policy to deal with conflicts between TT and ET transmissions. Therefore one way reduce the transmission latency of frame mSC_3 is to configure mSC_3 as a TT frame while keeping mSC_2 as ET. This ensures that mSC_2 does not delay the transmission of mSC_3.

Besides mSC_3, we also configure mCA_1 and mCA_2 as TT frames and we employ a static task scheduler designed to reduce the input-output latencies for applications A_1 and A_2. Both the task and the communication scheduling tables are given in Appendix B.

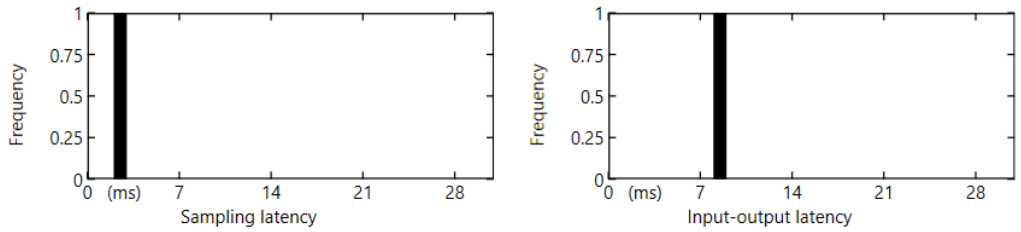
The results of the simulation, presented below, confirm an improvement in the control performance of application A_3, as its input-output latency is reduced and the jitter is removed completely. There is also an improvement in the QoC of applications A_1 and A_2.

Table 4-17 Simulation results for Case 3-5



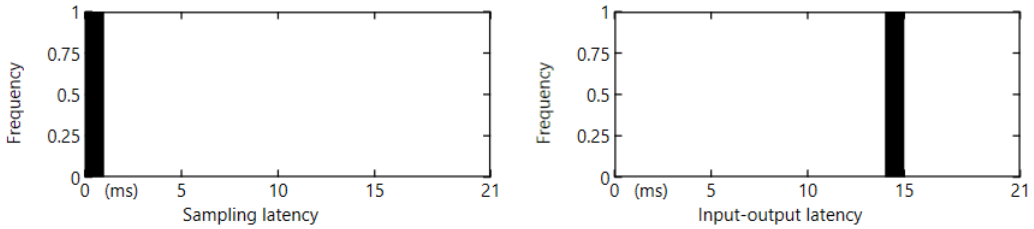
Control application: **App1**

Cost: **3.589**



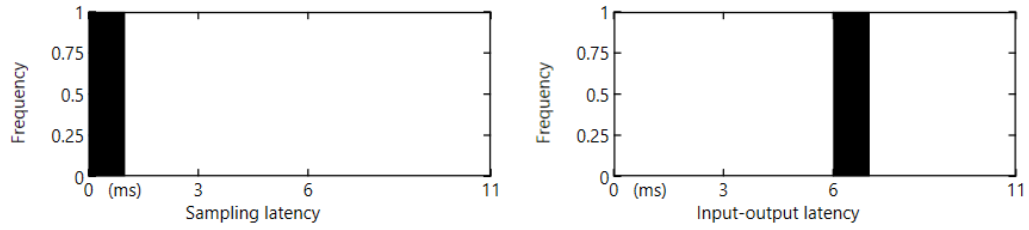
Control application: **App2**

Cost: **4.384**



Control application: **App3**

Cost: **4.144**



Case 3-4: Frame Routing 2 - RM scheduling and ET frames

Another possible way of reducing the transmission latency of frame mSC_3, in a context where all the frames are event-triggered, is to change its dataflow path as depicted in Figure 4.7. The rerouting reduces the length of the network segment shared with frame mSC_2 to only one dataflow link, i.e. [NS_3-N_3].

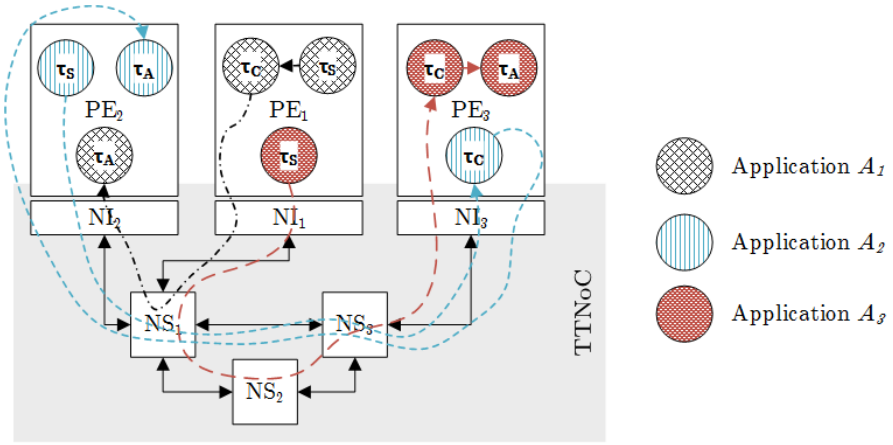


Figure 4.7 Frame Routing 2: frame mSC_3 routed via NS_2

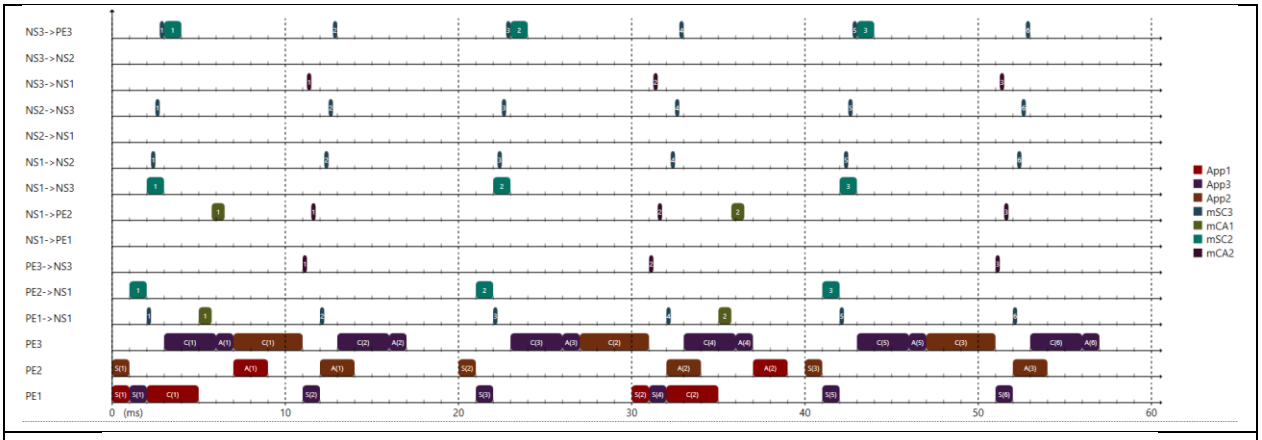
The new configuration of the frames is given in Table 4-18.

Table 4-18 Frame definitions

Frame	Associated application	Source task	Destination task	Transmission time [ms]	Priority	Route
mCA_1	A_1	τ_C	τ_A	0.75	1	N_1-NS_1-N_2
mSC_3	A_3	τ_S	τ_C	0.25	1	N_1-NS_1-NS_2-NS_3-N_3
mSC_2	A_2	τ_S	τ_C	1	2	N_2-NS_1-NS_3-N_3
mCA_2	A_2	τ_C	τ_A	0.25	1	N_3-NS_3-NS_1-N_2

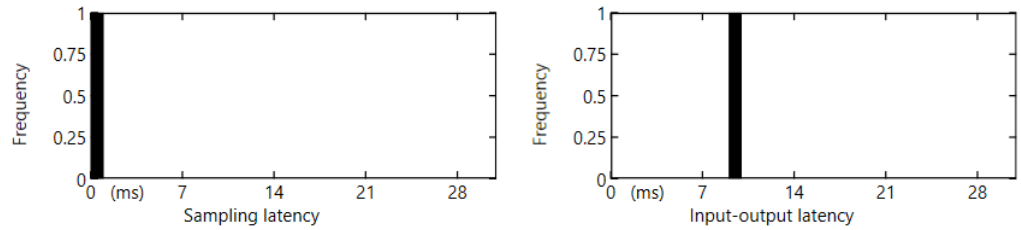
By simulating the system for a period of 60ms, we get the results from below. Because it has a shorter transmission time, frame mSC_3 manages to reach NS_3 before frame mSC_2 does, while taking a longer route.

Table 4-19 Simulation results for Case 3-2



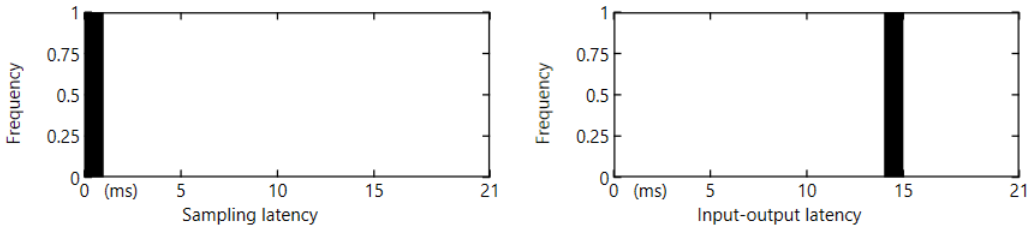
Control application: **App1**

Cost: **3.644**



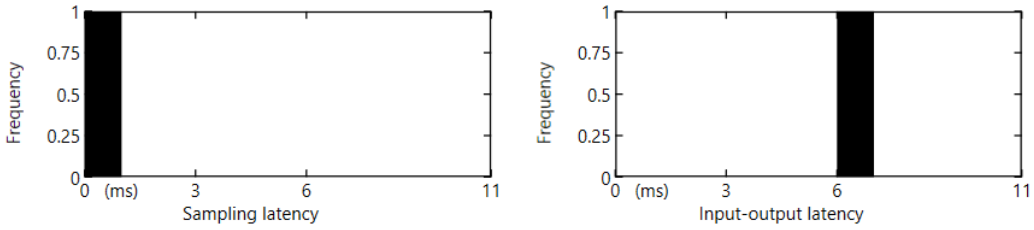
Control application: **App2**

Cost: **4.384**



Control application: **App3**

Cost: **4.144**



5. Conclusion

In this thesis we have proposed a tool that simulates control applications running on TTNoC based multicore systems. This tool provides an easy, quick and efficient way to evaluate the QoC of the simulated control applications. It supports highly customizable multicore system architectures.

We demonstrated de functionality of the SIMULATOR by simulating various system configurations. Based on the results of the simulations we could identify causes for degraded control performance and we suggested ways to address them. The simulations also revealed the importance of the design decisions concerning the hardware configuration, scheduling policies, task mapping, frame routing or frame scheduling.

6. Bibliography

[1] G. C. Buttazzo, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Boston: Kluwer Academic Publishers, 1997.

[2] F. Cottet, J. Delacroix and Z. Mammeri, Scheduling in Real-Time Systems, John Wiley & Sons Ltd, 2002.

[3] K. J. Åström and B. Wittenmark, Computer-controlled systems: theory and design, Prentice Hall, 1997.

[4] D. Tămaș–Selicean, *Design of Mixed-Criticality Applications on Distributed Real-Time Systems,* Kongens Lyngby, Denmak: Department of Applied Mathematics and Computer Science, DTU, 2014.

[5] A. Cervin and B. Lincoln, *Jitterbug 1.23 Reference Manual,* Lund, Sweden: Department of Automatic Control, Lund University, 2010.

[6] A. Cervin, *Integrated Control and RealTime Scheduling,* Lund, Sweden: Department of Automatic Control, Lund Institute of Technology, 2003.

[7] S. Hong, X. Sharon Hu and M. D. Lemmon, "Reducing Delay Jitter of Real-Time Control Tasks through Adaptive Deadline Adjustments," *Real-Time Systems (ECRTS), 22nd Euromicro Conference,* pp. 229-238, 2010.

[8] A. Cervin, D. Henriksson, B. Lincoln, J. Eker and K.-E. Årzén, *How Does Control Timing Affect Performance? Analysis and Simulation of Timing Using Jitterbug and TrueTime,* IEEE Control Systems Magazine, 2003, pp. 16-30.

[9] C. Paukovits and H. Kopetz, *Concepts of Switching in the Time-Triggered Network-on-Chip,* The 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2008, pp. 120 - 129.

[10] S. Samii, A. Cervin, P. Eles and Z. Peng, *Integrated Scheduling and Synthesis of*

Control Applications on Distributed Embedded Systems, Design, Automation & Test in Europe Conference & Exhibition, 2009, pp. 57-62.

Appendix A

System Configuration File – XML schema

Listing 0-1 System configuration file – XML schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">

  <xs:complexType name="genericApplicationType">
    <xs:sequence>
      <xs:element name="Tasks">
        <xs:complexType>
          <xs:sequence maxOccurs="1">
            <xs:element maxOccurs="unbounded" name="Task" type="genericAppTaskType"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="TaskDependencies">
        <xs:complexType>
          <xs:sequence maxOccurs="1" minOccurs="1">
            <xs:element maxOccurs="unbounded" minOccurs="0" name="TaskDependency"
              type="taskDependencyType"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="Id" type="xs:int" use="required"/>
    <xs:attribute name="Name" type="xs:string" use="required"/>
    <xs:attribute name="UseFixedTaskExecutionTimes" type="xs:boolean" use="required"/>
  </xs:complexType>
  <xs:complexType name="genericAppTaskType">
    <xs:attribute name="Id" type="xs:int" use="required"/>
    <xs:attribute name="Name" type="xs:string" use="required"/>
    <xs:attribute name="BCET" type="xs:int" use="required"/>
    <xs:attribute name="WCET" type="xs:int" use="required"/>
    <xs:attribute name="Priority" type="xs:int" use="optional"/>
    <xs:attribute name="Period" type="xs:decimal"/>
  </xs:complexType>
  <xs:complexType name="controlApplicationType">
    <xs:sequence>
      <xs:element name="Tasks">
        <xs:complexType>
          <xs:sequence maxOccurs="1">
```

```
            <xs:element maxOccurs="unbounded" name="Task" type="controlAppTaskType"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="TaskDependencies">
        <xs:complexType>
          <xs:sequence maxOccurs="1" minOccurs="1">
            <xs:element maxOccurs="unbounded" minOccurs="0" name="TaskDependency"
              type="taskDependencyType"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="Id" type="xs:int" use="required"/>
    <xs:attribute name="Name" type="xs:string" use="required"/>
    <xs:attribute name="UseFixedTaskExecutionTimes" type="xs:boolean" use="required"/>
    <xs:attribute name="Period" type="xs:decimal" use="required"/>
    <xs:attribute name="Phase" type="xs:decimal"/>
    <xs:attribute name="G" type="xs:string" use="required"/>
    <xs:attribute name="H_S" type="xs:string"/>
    <xs:attribute name="H_C" type="xs:string"/>
    <xs:attribute name="H_A" type="xs:string"/>
    <xs:attribute name="CostMatrix" type="xs:string" use="required"/>
    <xs:attribute name="InputNoiseCovarianceMatrix" type="xs:string" use="required"/>
    <xs:attribute name="MeasurementNoiseCovarianceMatrix" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:complexType name="controlAppTaskType">
    <xs:attribute name="Id" type="xs:int" use="required"/>
    <xs:attribute name="Name" type="xs:string" use="required"/>
    <xs:attribute name="BCET" type="xs:int" use="required"/>
    <xs:attribute name="WCET" type="xs:int" use="required"/>
    <xs:attribute name="Priority" type="xs:int" use="optional"/>
    <xs:attribute name="Type" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="Sampler"/>
          <xs:enumeration value="Controller"/>
          <xs:enumeration value="Actuator"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
  <xs:complexType name="taskDependencyType">
    <xs:attribute name="ProducerTaskId" type="xs:int" use="required"/>
    <xs:attribute name="ConsumerTaskId" type="xs:int" use="required"/>
  </xs:complexType>
  <xs:complexType name="processingElementType">
    <xs:sequence>
      <xs:element maxOccurs="1" minOccurs="1" name="HostedTasks">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0" name="HostedTask">
              <xs:complexType>
                <xs:attribute name="Id" type="xs:int" use="required"/>
                <xs:attribute name="ApplicationId" type="xs:int" use="required"/>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element minOccurs="0" name="SchedulingTable">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0" name="Entry">
              <xs:complexType>
                <xs:attribute name="Time" type="xs:decimal" use="required"/>
                <xs:attribute name="Duration" type="xs:decimal" use="required"/>
```

```
              <xs:attribute name="TaskId" type="xs:int" use="required"/>
              <xs:attribute name="ApplicationId" type="xs:int" use="required"/>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <xs:attribute name="Period" type="xs:decimal" use="required"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="Id" type="xs:int" use="required"/>
  <xs:attribute name="Name" type="xs:string" use="required"/>
  <xs:attribute name="ClockPeriod" type="xs:decimal" use="required"/>
  <xs:attribute name="SchedulingPolicy" use="required">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="FP"/>
        <xs:enumeration value="SC"/>
        <xs:enumeration value="EDF"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
</xs:complexType>
<xs:complexType name="networkComponentType">
  <xs:sequence>
    <xs:element name="SchedulingTable" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
          <xs:element maxOccurs="unbounded" minOccurs="0" name="Entry">
            <xs:complexType>
              <xs:attribute name="Time" type="xs:decimal" use="required"/>
              <xs:attribute name="Duration" type="xs:decimal" use="required"/>
              <xs:attribute name="FrameId" type="xs:int" use="required"/>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <xs:attribute name="Period" type="xs:decimal" use="required"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="Id" type="xs:int" use="required"/>
  <xs:attribute name="Name" type="xs:string"/>
</xs:complexType>
<xs:complexType name="frameType">
  <xs:sequence>
    <xs:element name="DataFlowPath">
      <xs:complexType>
        <xs:sequence>
          <xs:element maxOccurs="unbounded" name="NetwComp" minOccurs="2">
            <xs:complexType>
              <xs:attribute name="Id" type="xs:int" use="required"/>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="Id" type="xs:int" use="required"/>
  <xs:attribute name="Name" type="xs:string" use="required"/>
  <xs:attribute name="AssociatedAppId" type="xs:int" use="required"/>
  <xs:attribute name="SourceTaskId" type="xs:int" use="required"/>
  <xs:attribute name="DestinationTaskId" type="xs:int" use="required"/>
  <xs:attribute name="Size" type="xs:int" use="required"/>
  <xs:attribute name="Priority" type="xs:int"/>
  <xs:attribute name="Type" use="required">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="ET"/>
```

```
          <xs:enumeration value="TT"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
  <xs:element name="SystemConfiguration">
    <xs:complexType>
      <xs:all>
        <xs:element name="Applications">
          <xs:complexType>
            <xs:sequence>
              <xs:choice maxOccurs="unbounded">
                <xs:element name="ControlApplication" type="controlApplicationType"/>
                <xs:element name="GenericApplication" type="genericApplicationType"/>
              </xs:choice>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="ProcessingElements">
          <xs:complexType>
            <xs:sequence>
              <xs:element maxOccurs="unbounded" minOccurs="1" name="ProcessingElement"
                type="processingElementType"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="TTNetwork">
          <xs:complexType>
            <xs:all>
              <xs:element name="NetworkComponents">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element maxOccurs="unbounded" name="NetworkComponent"
                      type="networkComponentType" minOccurs="0"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element minOccurs="1" name="DataFlowLinks">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element maxOccurs="unbounded" minOccurs="0" name="DataFlowLink">
                      <xs:complexType>
                        <xs:attribute name="FromNetwCompId" type="xs:int" use="required"/>
                        <xs:attribute name="ToNetwCompId" type="xs:int" use="required"/>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:all>
            <xs:attribute name="ClockPeriod" type="xs:decimal" use="required"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="Frames">
          <xs:complexType>
            <xs:sequence>
              <xs:element maxOccurs="unbounded" minOccurs="0" name="Frame" type="frameType"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:all>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Appendix B

Static Scheduling Tables for Case 2-3

Table 0-1 Static task scheduling table on PE_1

Time [ms]	Duration [ms]	Application	Task
0	1	A_3	τ_S
1	1	A_1	τ_S
2	1	A_2	τ_S
10	1	A_3	τ_S
20	1	A_3	τ_S
21	1	A_2	τ_S
30	1	A_3	τ_S
31	1	A_1	τ_S
40	1	A_3	τ_S
41	1	A_2	τ_S
50	1	A_3	τ_S

Table 0-2 Static task scheduling table on PE_2

Time [ms]	Duration [ms]	Application	Task
2	3	A_3	τ_C
5	1	A_3	τ_A
6	3	A_1	τ_C

9	4	A_2	τ_C
13	3	A_3	τ_C
16	1	A_3	τ_A
17	2	A_2	τ_A
19	2	A_1	τ_A
22	3	A_3	τ_C
25	1	A_3	τ_A
26	4	A_2	τ_C
30	2	A_2	τ_A
32	3	A_3	τ_C
35	3	A_1	τ_C
38	1	A_3	τ_A
39	2	A_1	τ_A
42	3	A_3	τ_C
45	1	A_3	τ_A
46	4	A_2	τ_C
50	2	A_2	τ_A
52	3	A_3	τ_C
55	1	A_3	τ_A

Static Task and Communication Scheduling Tables for Case 3-3

Table 0-3 Static task scheduling table on PE_1

Time [ms]	Duration [ms]	Application	Task
1	1	A_3	τ_S
2	1	A_1	τ_S
3	3	A_1	τ_C
11	1	A_3	τ_S
21	1	A_3	τ_S
31	1	A_3	τ_S

32	1	A_1	τ_S
33	3	A_1	τ_C
41	1	A_3	τ_S
51	1	A_3	τ_S

Table 0-4 Static task schedule table on PE_2

Time [ms]	Duration [ms]	Application	Task
0	1	A_2	τ_S
8	2	A_1	τ_A
12	2	A_2	τ_A
20	1	A_2	τ_S
32	2	A_2	τ_A
38	2	A_1	τ_A
40	1	A_2	τ_S
52	2	A_2	τ_A

Table 0-5 Static task schedule table on PE_3

Time [ms]	Duration [ms]	Application	Task
3	3	A_3	τ_C
6	1	A_3	τ_A
7	4	A_2	τ_C
13	3	A_3	τ_C
16	1	A_3	τ_A
23	3	A_3	τ_C
26	1	A_3	τ_A
27	4	A_2	τ_C
33	3	A_3	τ_C
36	1	A_3	τ_A
43	3	A_3	τ_C
46	1	A_3	τ_A
47	4	A_2	τ_C

53	3	A_3	τ_C
56	1	A_3	τ_A

Table 0-6 Static frame schedule table on NI_1

Time [ms]	Duration [ms]	Frame
2	0.25	mSC_3
6	0.75	mCA_1
12	0.25	mSC_3
22	0.25	mSC_3
32	0.25	mSC_3
36	0.75	mCA_1
42	0.25	mSC_3
52	0.25	mSC_3

Table 0-7 Static frame schedule table on NI_3

Time [ms]	Duration [ms]	Frame
11	0.25	mCA_2
31	0.25	mCA_2
51	0.25	mCA_2

Table 0-8 Static frame schedule table on NS_1

Time [ms]	Duration [ms]	Frame
2.25	0.25	mSC_3
6.75	0.75	mCA_1
11.5	0.25	mCA_2
12.25	0.25	mSC_3
22.5	0.25	mSC_3
31.5	0.25	mCA_2
32.25	0.25	mSC_3

36.75	0.75	mCA_1
42.25	0.25	mSC_3
51.5	0.25	mCA_2
52.25	0.25	mSC_3

Table 0-9 Static frame schedule table on NS_3

Time [ms]	Duration [ms]	Frame
2.5	0.25	mSC_3
11.25	0.25	mCA_2
12.5	0.25	mSC_3
22.75	0.25	mSC_3
31.25	0.25	mCA_2
32.5	0.25	mSC_3
42.5	0.25	mSC_3
51.25	0.25	mCA_2
52.5	0.25	mSC_3