REAL-TIME SCHEDULING ANALYSIS FOR AUTOMOTIVE EMBEDDED SYSTEMS

by

Jorge Luis Martinez Garcia

A thesis submitted to the Faculty and the Board of Trustees of the University of Modena and Reggio Emilia in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Informatics).

Modena, Italy

January 19, 2021

Jorge Luis Martinez Garcia
Author

Dr. Marko Bertogna
Thesis Advisor

Dr. Ignacio Sañudo
Thesis Advisor

Dr. Martina Maggio
Examining Commission

Dr. Mitra Nasri
Examining Commission

Dr. Cristian Giardina
Head of PhD School
Department of Physics, Informatics and Mathematics

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AUTOSAR Extensible Markup Language	AXML
Actual Execution Curve	AEC
Application Programming Interface	API
Application Software	ASW
Automotive Open System Architecture	AUTOSAR
Basic Software	BSW
Controller Area Network	CAN
Deferrable Server	DS
Effect Chain	EC
Electronic Control Unit	ECU
Engine Management System	EMS
Harmonic Synchronous Communication	HSC
Hierarchical Fixed Priority Preemptive System	HFPPS
Human-Machine Interface	HMI
Logical Execution Time	LET
Microcontroller Abstraction Layer	MCAL
Non-Harmonic Synchronous Communication	NHSC
Operating System	OS
Original Equipment Manufacturer	OEM
Polling Periodic Server	PPS
Response Time Analysis	RTA
Runtime Environment	RTE
Software Component	SWC
Sporadic Server	SS
Worst-Case Execution Time	WCET

Worst-Case Response Time	WCRT

To my mother and to the memories of my father and my grandmother.

CHAPTER 1
INTRODUCTION

1.1 Automotive Embedded Systems

In recent years, the amount of electronics in automotive vehicles has risen dramatically, representing a significant share of the overall cost of the vehicle. Figure 1.1 shows part of the embedded mechatronic architecture of a vehicle illustrating computers, known as *Electronic Control Units* (ECUs), controlling different parts of the car, such as the engine. The technological reason behind such a change in the automotive industry lies in the increased number of safety and control functionalities that are being integrated in modern cars, as well as in the replacement of older hydraulic and mechanical direct actuation systems with modern by-wire counterparts, leading to an increased safety and comfort at a reduced unit cost. Well-known examples are anti-lock braking system (ASB), electronic stability program (ESP), active suspension, etc.

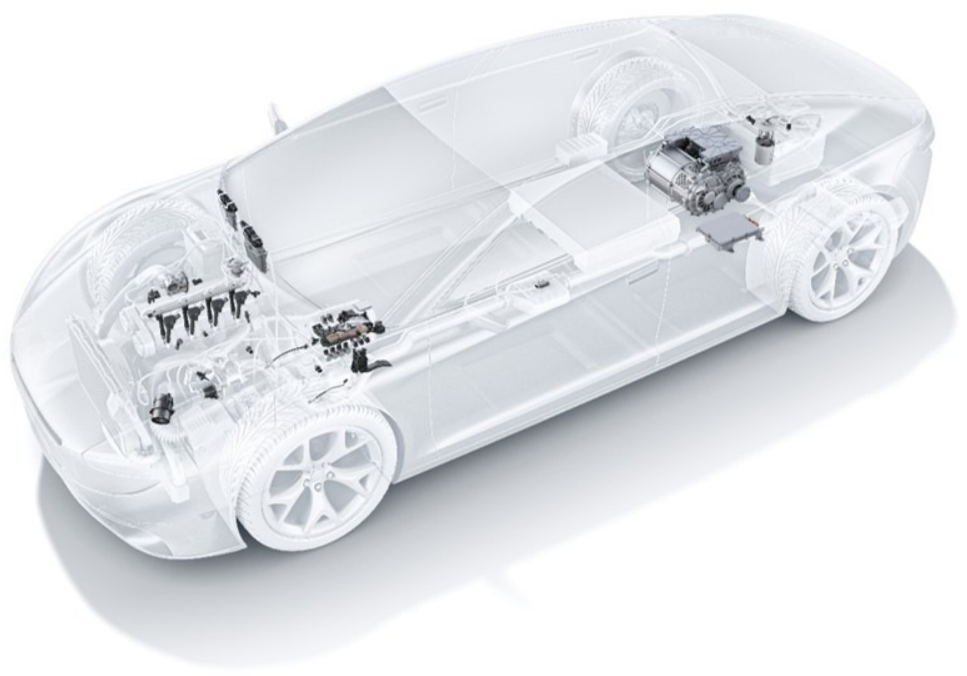


Figure 1.1 Part of the embedded mechatronic architecture of a vehicle.

On the other hand, carmakers distinguish between five functional domains, namely powertrain, chassis, body, HMI and telematics. The powertrain domain is related to the systems that participate in the longitudinal propulsion of the vehicle, including engine, transmission, and all subsidiary components. The chassis domain refers to the four wheels and their relative position and movement; in this domain, the systems

Figure 1.2 Electronic Control Unit (ECU).

are mainly steering and braking. The body domain includes the entities that do not belong to the vehicle dynamics, thus being those that support the car's user, such as airbag, wiper, lighting, window lifter, air conditioning, seat equipment, etc. The HMI domain includes the equipment allowing information exchange between electronic systems and the driver (displays and switches). Finally, the telematic domain is related to components allowing information exchange between the vehicle and the outside world (radio, navigation system, Internet access, payment)[1].

Note that from one domain to another, ECUs might have very different features. For example, while the powertrain and chassis domains both exhibit hard real-time constraints and a need for high computational power, the telematic domain presents requirements for high data throughput. As the given use cases [2] come from the powertrain domain, a basic understanding of real-time systems is needed.

1.2 Real-Time Systems

A real-time system is defined as any information processing system which has to respond to externally generated input stimuli within a finite and specified period: the correctness depends not only on the logical result but also on the time it was delivered; the failure to respond is as bad as the wrong response[3]. Nowadays these systems are present in avionic and automotive applications.

In particular, an Engine Management System (EMS), a type of ECU that controls the engine's fuel supply by means of sensors and actuators, is composed of real-time tasks with tight timing constraints[2]. A periodic real-time task τ_i releases an infinite sequence of jobs and typically presents the following parameters: computation time (C_i), offset (O_i), and period (T_i). While T_i is the constant rate at which τ_i is activated, O_i is the activation time of the first periodic instance, i.e. the first job. C_i denotes the time necessary to the processor for executing τ_i without interruption. See Figure 1.3.

As the performance and stability of control algorithms of automotive applications typically depend on the propagation delays induced by the communication between tasks, automotive embedded software engineers are especially concerned with optimizing end-to-end propagation latencies of input events that trigger a

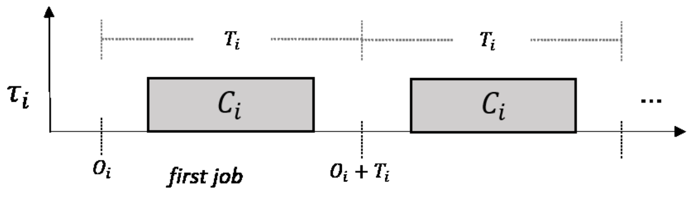


Figure 1.3 Typical parameters of a periodic real-time task.

chain of computations leading to a control action or final actuation[4], [5], [6]. Hence, automotive engineers are interested in the end-to-end latency study, or rather, characterization of *Effect Chains* (ECs). While an EC can be defined as a producer/consumer relationship between tasks, the definition of end-to-end latency depends on the specific end-to-end timing semantic used to characterized the timing delays of ECs. [7] describes four different semantics.

Figure 1.4 shows an EC formed by the communication of three tasks and triggered by a periodic sensor (incoming green arrows). The upper task reads the sensor data, elaborates it, and shares the result with the next task. And so on, until the end of the event chain. Green arrows denote when an input is propagated to the next task, *valid* input. Red arrows correspond to elaborations that are not propagated, *invalid* inputs, because they are overwritten before being read by the next task in the chain. From the figure, we can see that the age latency is defined as the delay between a valid sensor input until the last output related to this input in the EC.

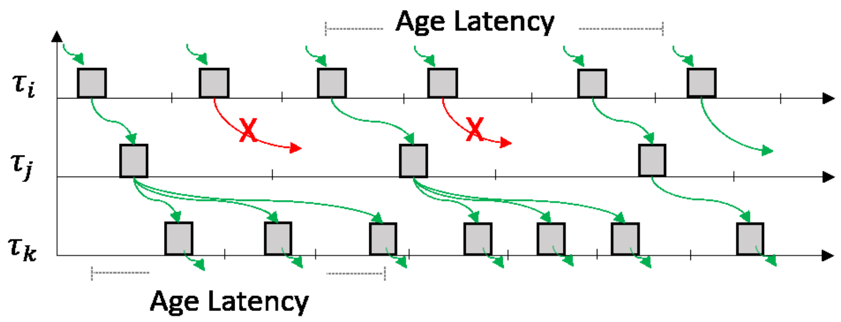


Figure 1.4 Age latency of an EC

1.3 The Logical Execution Time communication

In the automotive domain, tasks communicate by means of shared variables according to three different models: *Explicit, Implicit and Logical Execution Time (LET)* [6]. Each of these communication models has a different impact over the end-to-end latency experienced by a given EC.

Lately, there has been an increasing interest in the LET model in industrial domains, such as automotive [4] and avionics [8] [9], thanks to the improved determinism that can be achieved. In a real-time context, the LET semantics fixes the time it takes from reading task input to writing task output, regardless of the actual execution time of the task.

Due to its semantics, the LET communication may lengthen the end-to-end latency of an EC in comparison to its Implicit and Explicit counterparts [6]. Moreover, if the EC is composed of tasks with harmonic periods, then the end-to-end latency is always constant. However, if one pair has non-harmonic periods, then the end-to-end latency may vary due to the misalignment of the task periods. Hence, the goal of this dissertation is focused on the following two questions:

Q1 Which real-time mechanism may shorten the end-to-end latency of an EC composed of tasks following the LET semantics?

Q2 Which real-time technique can improve the determinism of the end-to-end latency of of an EC composed of tasks following the LET semantic, when this latency varies?

1.4 Fixed Priority Real-Time Servers

Compositional timing guarantees provided by server-based systems are becoming essential to the automotive domain to accommodate newer emerging setups, such as domain controllers, where software components with different timing requirements are designed by distinct vendors independently, and are eventually integrated and deployed by the *original equipment manufacturer* (OEM) on the the same platform. The need is therefore to be able to guarantee that the performance of these components is not degraded on integration and their timing properties are preserved. Additionally mechanisms to handle newer applications with dynamic, i.e. non-periodic, loads in a predictable and efficient manner are needed.

Unfortunately, the standard by which automotive software is developed, AUTOSAR (Refer to Chapter 2), does not propose any *application programming interface* (API) to implement compositional scheduling by means of servers and is essentially not designed to handle non-periodic requests efficiently. The existing practice in the automotive and avionic domain is to employ mechanisms like *Time Division Multiple Access* (TDMA) to provide temporal isolation among different applications[10] and although this approach is suitable for well-known periodic loads, it is inefficient by design due to its non work-conserving nature and not suitable for emerging applications with dynamic workloads. Thus, as real-time servers allow achieving timing isolation between previously isolated and functionally diverse applications by virtue of their design, there is a renewed interest for the adoption of fixed priority real-time servers in the automotive domain [11], as a way to implement more efficient reservation mechanisms than TDMA-based methods.

According to AUTOSAR, safety critical applications are to be scheduled by following a fixed priority policy due to its more deterministic and more predictable nature as well as its simpler implementation in comparison to dynamic priority scheduling [12]. Thus, it is not surprising that the use of fixed priority servers is of particular interest to the automotive domain[11].

With respect to fixed priority real-time servers, the research presented herein aims at answering the following questions:

Q3 Which existing Response Time Analysis (RTA) accurately captures the exact response time of jobs, i.e. the difference between its finishing time and its arrival time, released by tasks served by fixed priority servers?

Q4 How can we properly select the parameters of a given server?

Q5 Given the popularity of a particular kind of fixed priority server in real-time systems, namely the Sporadic Server[13], is there another kind of fixed priority server, such as the Polling Periodic Server or Deferrable Server[14], that performs equally well?

Q6 Is it possible to implement a fixed priority server without modifying the kernel of an AUTOSAR-compliant operating system (OS)?

1.5 Contributions

In the following, we briefly summarize the contributions presented in the subsequent chapters.

1.5.1 Introducing Offsets into the LET Communication Model

Offset assignment [15] is a well-known technique that has been adopted in the past to reduce the output jitter of a task, interact with slow devices, establish precedence constraints, obtain resource separation, increase feasibility bounds, and shorten WCRTs [16]. In Chapter 3 it is shown that by introducing tasks offsets, the end-to-end latency of non-harmonic tasks may shorten, getting closer to the constant end-to-end latency experienced in the harmonic case. In this way, the introduction of offsets not only may reduce response times and end-to-end latencies, but it also allows decreasing the jitter of important control parameters. This allows to answer Q1 and Q2. Thus, in this work, a formal overhead-aware analysis of the LET communication model for real-time systems composed of periodic tasks with harmonic and non-harmonic periods is provided, analytically characterizing the control performance of LET effect chains.

1.5.2 Offset Assignment Algorithm

As argued above, offset assignment is the real-time technique that responds to Q1 and Q2. However, this raises the following question: How can we assign offsets to tasks? Thus, in Chapter 3 a heuristic algorithm to obtain a set of offsets that might reduce end-to-end latencies is provided, improving the LET communication determinism. Furthermore, this technique is validated through an industrial case study consisting of an automotive engine control system provided by Robert Bosch GmbH [4].

1.6 Exact Response Time Analysis for Fixed Priority Servers

With respect to Fixed Priority Servers and in response to Q3, in Chapter 4 it is shown that existing analyses such as [17] and [18] are not capable of computing exact response times of jobs in a two-level fixed priority hierarchical setting, i.e. when fixed priority servers are scheduled at system-level (globally) and each server contains a set of tasks that are scheduled locally. Thus, a formal characterization of an exact RTA for fixed priority systems based on fixed priority servers in a two-level scheduling setting under preemptive scheduling is provided in Chapter 5. Moreover, an experimental comparison between the proposed RTA and existing sufficient schedulability tests proves that significant schedulabilty improvement can be obtained.

1.6.1 A Server Parameter Selection Technique

While the proposed analysis applies for Polling Periodic, extended Polling Periodic, Deferrable and Sporadic Servers; we focus on the last two due to their bandwidth-preserving nature: unlike the *periodic* servers, the Deferrable and Sporadic Servers preserve their reserved fraction of CPU bandwidth if no requests are pending upon their invocation. Thus, in Chapter 6 the analysis is extended so that the overhead induced by the implementation of these two bandwidth-preserving fixed priority servers can be taken into account. It is this extension that allows a proper investigation into the server parameter selection problem (Q4). Based on this study, in the same chapter a server parametrization heuristic is provided that results in schedulable systems with low utilization and small aggregated WCRTs.

1.6.2 Comparison between Bandwidth-Preserving Fixed Priority Servers

Examples of fixed priority servers include the Polling and Deferrable Servers, as well as the Sporadic Server. Although actual implementations of each of these servers can be found in hypervisors such as RT-XEN [19], the Sporadic Server is the only one specified in the IEEE Portable Operating System Interface (POSIX) standard[20], due to the fact that the Sporadic Server has been traditionally considered a better approach to the Deferrable Server due to its supposed higher achievable utilisation[21]. In Chapter 6, by means of the aforementioned overhead-aware RTA and the proposed server parameter selection technique, a comparison between the Deferrable Server and the Sporadic Server is drawn, proving that both servers perform equally well from an overhead as well as from a schedulability point of view (Q5).

1.6.3 Implementing a Bandwidth-Preserving Fixed Priority Server on top of a AUTOSAR-compliant OS

While there has been attempts[22], [23] to extend $\mu C/OS\text{-}II$[1] in order to support fixed priority hierarchical scheduling, μC/OS-II no longer complies with OSEK[23], a previous automotive standard. Hence, as another contribution of this thesis, Chapter 7 presents a method to implement a Deferrable Server on top of ETAS RTA-OS[2], a ubiquitous AUTOSAR-compliant OS, hence answering Q6. This type of server was chosen due to its similar performance (refer to Chapter 6) and less complicated implementation in comparison to the Sporadic Server. Moreover, it is believed that this work is the first one to present a fixed-priority server implementation on top of an AUTOSAR-compliant OS. The aforementioned implementation is then deployed on top of an Infineon's TC297 processor in order to serve tasks originally scheduled in the background so that they can meet their deadlines, thereby satisfying an industrial use case provided by Bosch as shown in the same chapter.

1.7 Organization

The remainder of this dissertation is organized as follows. Chapter 2 provides basic knowledge of AUTOSAR as well as some features of an AUTOSAR-compliant OS, namely RTA-OS, in order to provide the basic knowledge needed to understand the rest of this work. Chapter 3 introduces the rationale behind the use of LET in the automotive domain and provides a formal offset-aware analysis of the LET model for real-time systems. Additionally, a heuristic algorithm is presented in order to obtain a set of offsets that might reduce end-to-end latencies, improving the communication determinism offered by the LET communication model.

Chapters 4-7 present the fixed-priority-server-related contribution. In Chapter 4, a method to characterized real-time tasks and servers is proposed. This framework allows computing the exact RTA of jobs in a fixed priority hierarchical setting, as shown in Chapter 5. Chapter 6 extends the aforementioned analysis in order to take the effect of overheads into account. This extension allows a correct study of the server parametrization problem and a comparison between the Deferrable and Sporadic Server. Chapter 7 presents a method to implement a Deferrable Server on top of RTA-OS and proposes a heuristic to select its parameters. The effectiveness of the parametrization is the proven by applying the technique to an industrial case study consisting of an automotive engine control system. Finally, Chapter 8 summarizes the results, raises open questions, and discusses future work.

[1]www.micrium.com/rtos/kernels
[2]www.etas.com

CHAPTER 2
AUTOMOTIVE OPEN SYSTEM ARCHITECTURE

Automotive control software is developed according to the AUTomotive Open System ARchitecture (AUTOSAR [3]) standard. AUTOSAR establishes a uniform development methodology, a uniform terminology for automotive control software and provides a standardization of the interfaces between the different software layers giving a hierarchical organization of the software/hardware components deployed in the vehicle. Moreover, AUTOSAR looks at the different functionalities in a car network, combines them into logical clusters (software compositions), and finds functional atomic units (software components) that compose these clusters.

The smallest functional entity in AUTOSAR is called *runnable*. A software component (SWC) is made up of one or more runnables. Runnables having the same functional period are grouped into the same task. In the simplest case, one functionality is realized by means of a single runnable. However, more complex functionalities are typically accomplished using several communicating runnables, possibly distributed over multiple tasks.

In this chapter, AUTOSAR's architecture as well as some features of a particular implementation of a ubiquitous AUTOSAR-compliant OS, namely ETAS RTA-OS, are exposed. RTA-OS was chosen as it has been standardised upon by many of the world's leading automotive powertrain systems and chassis electronics suppliers, and is used in cars produced by nearly all of the world's major car manufacturers. This chapter is of vital importance to the presented work as the analyses and implementations presented in the following chapters are based on the concepts presented in this one.

2.1 AUTOSAR Architecture

The AUTOSAR architecture is composed of three main layers: (i) *Application Software* (ASW), (ii) *Run-Time Environment* (RTE), and (iii) *Basic Software* (BSW), as detailed in Figure 2.1(a).

1. The ASW consists of interconnected SWCs with well-defined interfaces, described and standardized within AUTOSAR, that are provided to communicate with other SWCs.

2. The communication between SWCs is enabled by the RTE. This layer makes SWCs independent from the mapping to a specific ECU and provides different communication paradigms between SWCs, such as sender-receiver, client-server, etc. In this work, we focus on the sender-receiver communication

[3]`https://www.autosar.org/`

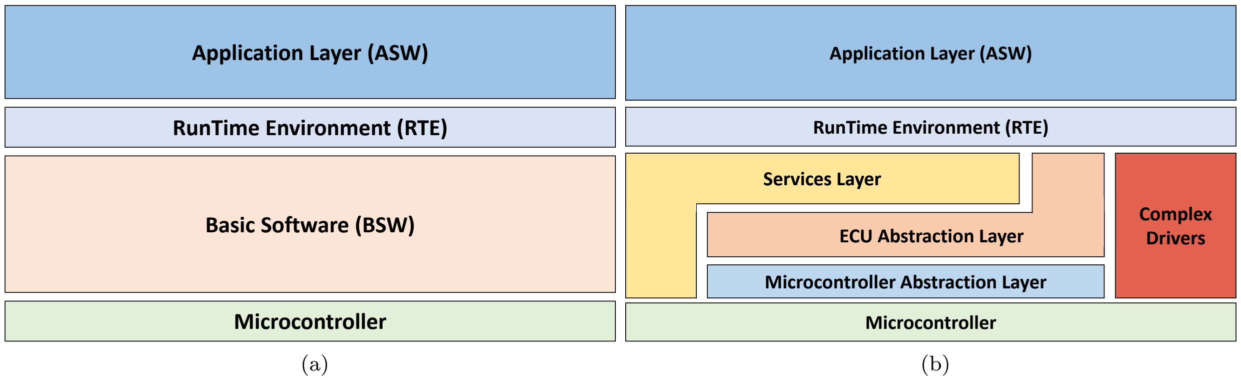


Figure 2.1 AUTOSAR architecture (a); BSW sublayers (b)[24]

paradigm, that is the memory sharing mechanism allowing tasks to communicate by means of shared variables, aka shared *labels*. For this sort of communication, the RTE supports two modes, namely Explicit and Implicit.

3. The BSW provides the infrastructural functionality for an ECU and is composed of the following sub-layers (see Figure 2.1(b)): the Microcontroller Abstraction layer (MCAL) which provides hardware drivers making upper software layers independent from the microcontroller; the ECU abstraction layer which provides APIs to access peripherals making upper software layers independent from the ECU hardware layout; and the Service Layer that provides operating system functionalities, memory services, diagnostic services, etc. Drivers that are not specified in AUTOSAR are to be found in the Complex Drivers layer.

2.2 AUTOSAR OS

The AUTOSAR OS standard has been adopted in all types of ECUs, from powertrain, chassis and body to multi-media devices, and defines a small, scalable, real-time operating system that is ideal in embedded systems with limited memory and dedicated functions. This OS manages real-time tasks, enhanced timer functions (referred to as alarms), shared resources, task synchronization using events, etc. Moreover, AUTOSAR OS is entirely statically defined using an offline configuration language called AXML (AUTOSAR Extensible Markup Language). Since all objects are known at system generation time, implementations can be extremely small and efficient.

The representation of time in the considered AUTOSAR OS implementation, RTAS-OS, is achieved by receiving a clock-tick interrupt at a fixed frequency so that time can be measured as a count of the number of times the clock-tick interrupt has occurred. The period between clock-tick interrupts is referred to as a *tick*. For example, if the clock-tick interrupt arrived every 100us then each tick would represent 100us.

2.2.1 Basic Tasks

Tasks are the main building block of AUTOSAR OS systems and have a statically defined priority. Moreover, while tasks can be scheduled by the OS either preemptively or non-preemptively, in AUTOSAR there is a third type of scheduling policy called cooperative where a non-preemptive task tells the OS when it could be preempted. Furthermore, a task can be either *basic* or *extended. Basic tasks* run to completion unless preempted by a higher priority task or interrupt. Basic tasks can exist in one of the following three states: *running, ready and suspended.* Transitions between states are modeled through four different processes: *activate, start, preempt, and terminate.* The state transition diagram for an AUTOSAR OS task is shown in Figure 2.2.

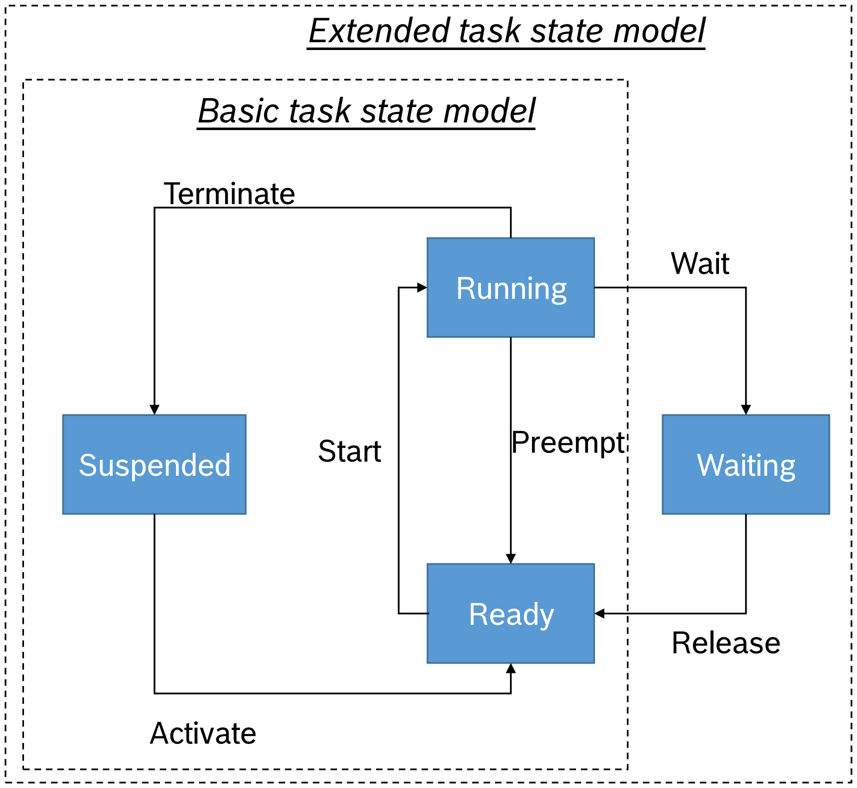


Figure 2.2 AUTOSAR OS Task Model[25]

The default state for all tasks is *suspended.* A task is moved into the ready state by the process of *activation*, when it is ready to run, e.g. due to the expiration of an alarm. When a task becomes the highest priority task, the OS moves this task into the *running* state and starts its execution. A task may be preempted during execution by other higher priority tasks that become ready. If a higher priority task becomes ready to run, the currently executing task is preempted and is moved from the running state into the ready state. This means that only one task can be in the running state at any one time. A task returns to the *suspended* state by terminating. The following code snippet shows an example of a basic task called *Basic_Task*. Notice that a basic task must make an API call, such as TerminateTask(), to tell the OS that this is happening.

Listing 2.1: A basic task

```
TASK(Basic_Task) {
  do_something();
  TerminateTask();
}
```

2.2.2 Extended Tasks

Extended tasks are similar to basic tasks except they have one additional state in their state transition diagram, *waiting*, and two additional transitions, *wait* and *release* (See Figure 2.2). An extended task moves from the running to the waiting state when it voluntarily suspends itself by waiting on an *event*. An event is simply an OS object that is used to provide an indicator for a system event. The following code snippet shows an example of an extended task called *Extended_Task* waiting for events. Notice that an extended task optionally terminates.

Listing 2.2: An extended task

```
TASK(Extended_Task) {
  while (TRUE) {
    do_something1();
    WaitEvent(My_Event);
    do_something2();
    ClearEvent(My_Event);
  }//The task never terminates
}
```

As shown in the previous example, an extended task waits for an event using the API call *WaitEvent()*. Observe that when a task waits on an event, and the event occurs, then a subsequent call to WaitEvent() for the same event will return immediately because the event is still set. Thus, before waiting for the event to take place again, the last occurrence of this event must be cleared. Events are cleared using the *ClearEvent()* API call.

2.2.3 Alarms and Counters

Other services available within the AUTOSAR OS API include the concept of *alarms* and *counters*. AUTOSAR makes use of an *alarm* to cover the functions of a timer and the unique need of an embedded system to take an action based on the occurrence of a series of events. This is accomplished by creating a counter object that is incremented whenever an event occurs. A *counter* is an OS object that keeps track of the number of ticks that have occurred.

In every AUTOSAR OS implementation, at least one counter must be based on either a hardware or software timer. This counter is used then by alarms as a system timer to accomplish the same function as a timer in other real-time OSs. For example, an alarm based on counter can be used to schedule a periodically executing task.

Alarms are AUTOSAR objects that are associated with counters. When defining alarms, each alarm is statically assigned to one counter and one task; however, multiple alarms can be assigned to a given counter. Whenever a counter is incremented, the currently active alarms assigned to that counter are compared to the counter value. If the values are equal, the alarm is triggered and it can activate a task, set an event for a task, execute a callback function or increment a software counter.

While counter-specific constants are defined in the AXML configuration file, a standardized API for managing alarms exists. For instance, *SetRelAlarm(AlarmID, increment, cycle)* sets the alarm to expire *increment* ticks from the current count value when the function is called. This means that *increment* is a tick offset from the current counter tick value. A cycle value of zero ticks indicates that the alarm is a singleshot alarm, which means that it will expire only once before being canceled. A cycle value greater than zero defines a cyclic alarm (Refer to Figure 2.3 for an example). This means that it will continue expiring every cycle ticks after the first expiry has occurred. Another alarm-related API call that allows us to cancel an alarm is *CancelAlarm()*. An alarm may, for example, need to be canceled to stop a particular task being executed.

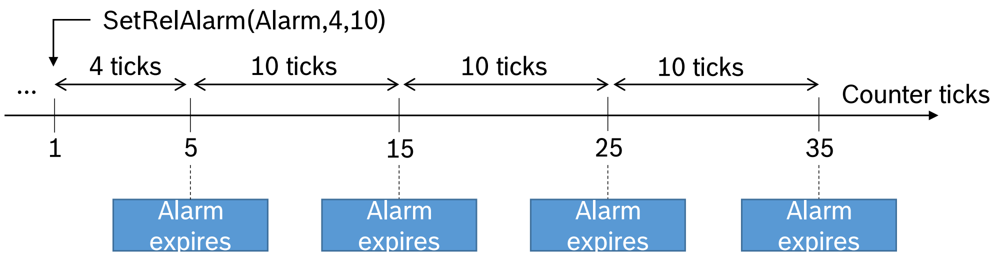


Figure 2.3 Illustration of a Relative Cyclic Alarm. Alarm expires 4 ticks from now and every 10 ticks thereafter.

2.2.4 Timing Monitoring

The AUTOSAR OS standard also provides API calls to monitor the execution time of a task. For example *Os_GetTaskElapsedTime()* returns the cumulative time spent by the OS executing a given task since the initialization of the OS or an explicit reset of the accumulated time. The latter can be achieved by means of the *Os_ResetTaskElapsedTime()* API call. Notice that the elapsed time is updated when the task finishes or is preempted.

2.3 Summary

In this chapter, AUTOSAR's architecture as well as some features of RTA-OS were presented. While basic tasks can exist in one of three states: running, ready and suspended; extended tasks have one extra state: waiting. Moreover, RTA-OS makes use of alarms and counters in order to cover the functions of timers. RTA-OS also offers two APIs, namely *Os_GetTaskElapsedTime()* and *Os_ResetTaskElapsedTime()*, to monitor the execution time of a task.

In the next chapter, based on the aforemntioned architecture, the LET inter-task communication model will be explored. Recall that an EC is defined as a chain of tasks, where each task has a runnable writing a shared label that is then read by a second task; this latter task processes the read variable, and then writes a different shared label, which is then read by a third task. And so on, until the end of the chain. The amount of time that elapses from the first input event until the end of the chain may significantly affect the control performance of the considered application. In particular, automotive engineers are especially concerned with optimizing end-to-end propagation latencies of ECs and this is the motivation of the following chapter.

CHAPTER 3
THE LOGICAL EXECUTION TIME COMMUNICATION MODEL

Modified from a journal paper[26] published in the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Jorge Martinez[4,5], Ignacio Sañudo[6,7] , Marko Bertogna[8]

In this chapter, the motivation behind the introduction of the LET as a task communication model in the automotive industry is discussed. Moreover, a possible implementation of the LET model is presented for the Harmonic and Non-Harmonic Synchronous Communication cases. Furthermore, an end-to-end latency analysis is provided, showing that communication determinism may be improved by combining static offset assignment with the LET model. To that end, a novel heuristic algorithm is proposed that assigns task offsets to reduce not only task WCRTs, but also end-to-end latency and jitter. It is demonstrated that the proposed algorithm may achieve comparable performance of a brute force method that explores the whole design space, but with a much more reasonable computational complexity. Finally, this technique is applied to an industrial case study consisting of an automotive engine control system.

3.1 Motivation

In AUTOSAR, runnables having the same functional period based on control dynamics are typically grouped into the same task. In the simplest case, one functionality is realized by means of a single runnable. Nevertheless, more complex functionalities are typically accomplished using several communicating runnables, possibly distributed over multiple tasks. Given an existing operational system, new functionalities are typically added by the addition or replacement of runnables, potentially modifying task computation times. These modifications may have a big impact on the end-to-end latency of a given effect chain.

Consider the example in Figure 3.1, where an effect chain composed of τ_1, τ_2 and τ_3 is shown. Task τ_1 has a runnable writing a label that is then read by τ_2; this latter task processes the read variable, and then writes a different label, which is then read by a runnable in τ_3. In the end, this runnable outputs an actuation signal that completes the effect chain. In this case, the amount of time that elapses from the first input event until the end of the chain, also known as the end-to-end latency, is 3. If the computation time

[4]Graduate student at the University of Modena and Reggio Emilia
[5]Primary researcher and author
[6]Postgraduate researcher at the University of Modena and Reggio Emilia
[7]Author for correspondence
[8]Full Professor at the University of Modena and Reggio Emilia

of some runnables is modified, or more runnables are added as in Figure 3.2, the end-to-end latency may increase (19 for the case in the figure).

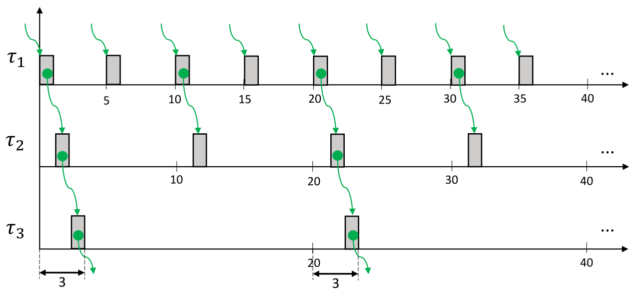


Figure 3.1 End-to-end effect chains composed of three tasks with parameters $T_1 = 5, T_2 = 10, T_3 = 20$ and $C_1 = C_2 = C_3 = 1$.

Control tasks are typically executed periodically, i.e. at a given sampling period. The resulting control performance is highly dependent on task jitter, task response times, scheduling policy and end-to-end latency of effect chains. Even a small change in one of these parameters might be detrimental to control performance, potentially requiring a system redesign, with related additional cost and time.

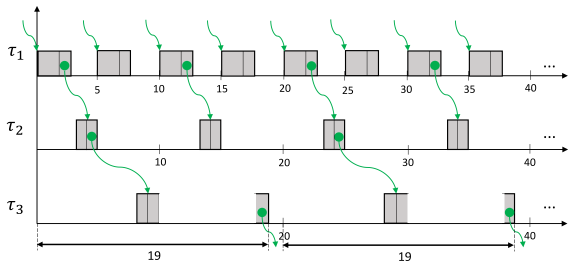


Figure 3.2 End-to-end effect chains composed of three tasks with parameters $T_1 = 5, C_1 = 3, T_2 = 10, C_2 = 2, T_3 = 20$ and $C_3 = 3$.

Even with constant execution times, different instances of the same task might have different response times, leading to variable end-to-end latencies of an effect chain. An example is shown in Figure 3.3. The LET concept has been introduced in the automotive industry to explicitly address this issue. The LET semantics decouples control algorithms from task jitter, task response times, scheduling policy and hardware dependence, enabling more robust algorithms and more deterministic and predictable systems.

3.2 Related Work

The LET paradigm has been proposed within the time-triggered programming language Giotto [27]. This communication pattern allows determining the time it takes from reading program input to writing program output, regardless of the actual execution time of a real-time program. As stated in [28], LET evolved

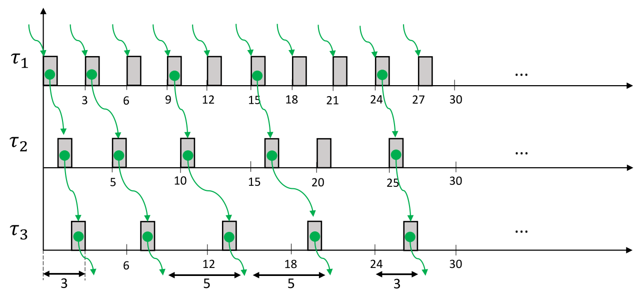


Figure 3.3 End-to-end effect chains composed of three tasks with parameters $T_1 = 3, T_2 = 5, T_3 = 6$ and $C_1 = C_2 = C_3 = 1$.

from a highly controversial idea to a well-understood principle of real-time programming, motivated by the observation that the relevant behavior of real-time programs is determined by when inputs are read and outputs are written. This concept has been adopted by the automotive and avionics industry as a way of introducing determinism in their systems.

In [5], an overview of the different communication patterns adopted in the automotive domain is provided, highlighting the importance of end-to-end latency of effect chains in an engine management system. A method to transform LET into a corresponding direct communication is also presented, allowing the use of classic tools (such as SymTA/S[9]) to determine end-to-end latencies and communication overhead. In [29], an end-to-end timing latency analysis for effect chains with specified age-constraints is presented. The analysis is based on deriving all possible data propagation paths which are used to compute the minimum and maximum end-to-end latency of effect chains. In [30], the analysis is extended to include the Logical Execution Time paradigm, providing an algorithm to derive the maximum data age of cause-effect chains. However, none of these works takes offset assignment into consideration in order to compute the end-to-end latency of effect chains.

As previously mentioned, offset assignment is a well-known method to reduce the output jitter of tasks, improving system schedulability and shortening the WCRT of tasks. A proper selection of task offsets may increase the predictability of the system by better distributing the workload over time. In [15], Tindell introduced the idea of using task offsets to model periodic transactions of different tasks. An exact response time analysis (RTA) was proposed for tasks with static offsets, showing that offsets can be used to reduce the pessimism of the classic response time analysis. Unfortunately, the presented RTA is computationally intractable but for small tasks sets. Therefore, an approximate RTA was also proposed. Later on, Palencia and Harbour [31] extended the approximate RTA of Tindell by analyzing tasks with static and dynamic offsets for distributed systems. While the static analysis assumes that offsets are fixed from the transaction

[9]`https://auto.luxoft.com/uth/timing-analysis-tools/`

release, dynamic offset analysis considers that offsets may change from one activation to another. In [32], a method is described to perform exact RTA for fixed priority tasks with offsets and release jitter based on the work in [31]. Recently, a RTA aware of end-to-end timing requirements has been published by Palencia et al. [33]. In this work, a method is presented to perform an offset-based RTA for time-partitioned distributed systems. Authors also considered effect chains with precedence constraints.

In [34], Goossens distinguished between three types of periodic task sets: (i) synchronous, where the offsets are fixed and all equal to 0 ($O_1 = O_2 = ... = O_n = 0$); (ii) asynchronous, where offsets are determined by the constrains of the system; and (iii) offset-free, where offsets are chosen by the scheduling algorithm. A method to assign offsets is presented, proposing different heuristics to determine a static offset for each task.

The offset assignment problem has also been studied for the automotive domain. In [35], Grenier et al. proposed the use of offsets to improve the task schedulability of body and chassis networks considering CAN-bus related delays. This technique is used to minimize the WCRT by distributing the workload over time. An offset assignment algorithm tailored for automotive CAN networks is presented to improve task WCRT. Based on this algorithm, Monot et al. proposed in [36] runnable-to-task allocation heuristics for multi-core platforms, balancing the CPU load over the system through offset assignment. Recently, Nasri et. al [37] presented an offset assignment technique for FIFO scheduling in order to obtain schedulability performance comparable to non-preemptive fixed priority scheduling, while incurring a smaller overhead.

To the best of our knowledge, the present work is the first study that formally defines an exact offset-aware schedulability analysis for the LET inter-task communication model. The impact of an offset-aware LET model on the end-to-end latency of effect chains is thoroughly analyzed, proposing a heuristic algorithm to obtain a convenient offset assignment.

3.3 System Model

The model is assumed to be comprised of m identical cores, with periodic tasks and runnables statically partitioned to the cores, and no migration support. Each task τ_i is specified by a tuple $(C_i, D_i, T_i, O_i, \Pi_i)$, where C_i stands for the WCET, D_i is the relative deadline, T_i is the period ($D_i = T_i$), O_i, is the initial offset, and Π_i is the priority. Hence, each task τ_i releases an infinite sequence of jobs, with the first job released at time O_i, and subsequent jobs periodically released at time $r_{i,k} = O_i + kT_i$. Without loss of generality, it is assumed that $O_i < T_i$ for all tasks τ_i.

The *hyperperiod* of the task system is the least common multiple of the task periods. Tasks communicate through shared labels in such a fashion that they abstract a message-passing communication mechanism implemented with a shared memory. Regarding the type of access, a task can be either a sender or a receiver of a label. A sender is a task that writes a label. It is assumed that there is only one sender per label, while there may be multiple receiving tasks reading one label.

3.4 Logical Execution Time

In the context of hard real-time systems, the LET semantics enforces task communications at deterministic times, corresponding to task activation times. LET fixes the time it takes from reading task input to writing task output, regardless of the actual execution time of the task. Inputs and outputs are logically updated at the beginning and at the end of their LET, respectively, see Figure 3.4. In this work it is assumed that the LET equals the task period. It is worth mentioning that the LET communication model assumes these updates incur zero computation time.

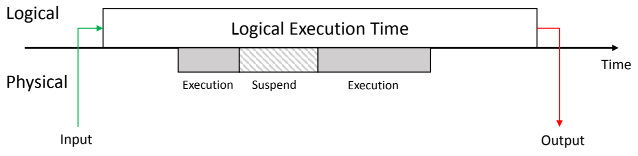


Figure 3.4 Logical Execution Time model.

The communication between the writer and one of the readers is hereafter considered. Assume the writer and the reader have period $T_W = 2$ and $T_R = 5$, respectively, as in Figure 3.5. While τ_W may repeatedly write the considered labels, these updates are not visible to the concurrently executing reader, until a *publishing point* $P^n_{W,R}$, where the values are updated for the next reader instance. This point corresponds to the first upcoming writer release that directly precedes a reader release, i.e., where no other write release appears before the arrival of the following reader instance. We call *publishing instance* the writing instance that updates the shared values for the next reading instance, i.e., the writer's job that directly precedes a publishing point. Note that not all writing instances are publishing instances. See Figure 3.5, where publishing instances are marked in bold red.

It is also convenient to define *reading points* $Q^n_{R,W}$, which correspond to the arrival of the reading instance that will first use the new data published in the preceding publishing point $P^n_{R,W}$. Figure 3.6 shows publishing and reading points for a case where $T_W = 5$ and $T_R = 2$.

The publishing and reading points of two communicating tasks can be computed as a function of their periods, as shown in the next theorem.

Theorem 1. *Given two communicating tasks τ_W and τ_R, the publishing and the reading points can be computed as*

$$P^n_{W,R} = \left\lfloor \frac{nT_{\max}}{T_W} \right\rfloor T_W \tag{3.1}$$

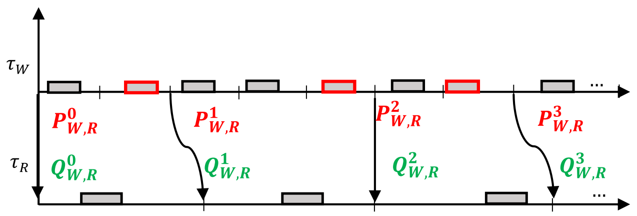


Figure 3.5 Publishing and reading points when the reader has larger period than the writer.

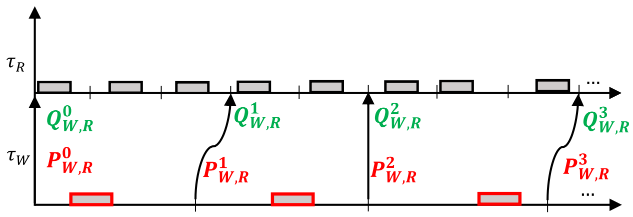


Figure 3.6 Publishing and reading points when the reader has smaller period than the writer.

$$Q^n_{W,R} = \left\lceil \frac{nT_{\max}}{T_R} \right\rceil T_R \tag{3.2}$$

where $T_{\max} = \max(T_W, T_R)$

Proof. If the writer τ_W has a smaller or equal period than the reader τ_R, i.e., $T_W \leq T_R$ as in Figure 3.5, there is one publishing and one reading point for each *reading* instance. Reading points trivially correspond to each reading task release, i.e., $Q^n_{W,R} = n \cdot T_R$, while publishing points correspond to the last writer release before such a reading instance, i.e., $P^n_{W,R} = \lfloor n \cdot T_R/T_W \rfloor \cdot T_W$.

Otherwise, when the writer τ_W has a larger period than the reader τ_R, i.e., $T_W \geq T_R$ as in Figure 3.6, there is one publishing and one reading point for each *writing* instance. Publishing points trivially correspond to each writing task release, i.e., $P^n_{W,R} = n \cdot T_W$, while reading points correspond to the last reader release before such a writing instance, i.e., $Q^n_{W,R} = \lceil n \cdot T_W/T_R \rceil \cdot T_R$.

It is easy to see that, in both cases $T_W \leq T_R$ and $T_W \geq T_R$, the formulas for $P^n_{W,R}$ and $Q^n_{W,R}$ are generalized by Equations (3.1) and (3.2). Note that, when $T_W = T_R$, $P^n_{W,R} = Q^n_{W,R} = nT_W$. □

3.4.1 Harmonic Synchronous Communication (HSC)

Two communicating tasks τ_W and τ_R have harmonic periods if the period of one of them is an integer multiple of the other. When a harmonic synchronous communication (HSC) is established, the following relations hold: $LCM(T_W, T_R) = T_{\max}$, and $P^n_{W,R} = Q^n_{W,R} = nT_{\max}$, i.e., publishing and reading points are integer multiples of the largest period of the communicating tasks.

Consider the example in Figure 3.7, where two tasks τ_l and τ_s, with $T_l = 2T_s$, both read shared labels L_1 and L_2. Moreover, τ_l writes L_1, while τ_s writes L_2. The proposal suggests that τ_s and τ_l are to read $L_{s,1}$ and $L_{l,2}$ instead of the original labels. Notice that τ_l and τ_s directly modify L_1 and L_2, respectively, instead of working with local copies. These copies are to be updated by a communication-specific runnable, either τ_s^{last} or τ_l^{last}, depending on whichever job finishes last before the next publishing point. In other words, the responsibility to update the copies is given either to the reader or to the writer, depending on which one completes last in the communication. The first reader instance after the publishing point is the first one that accesses the updated value. Such a value will be used by all reading instances until the next reading point.

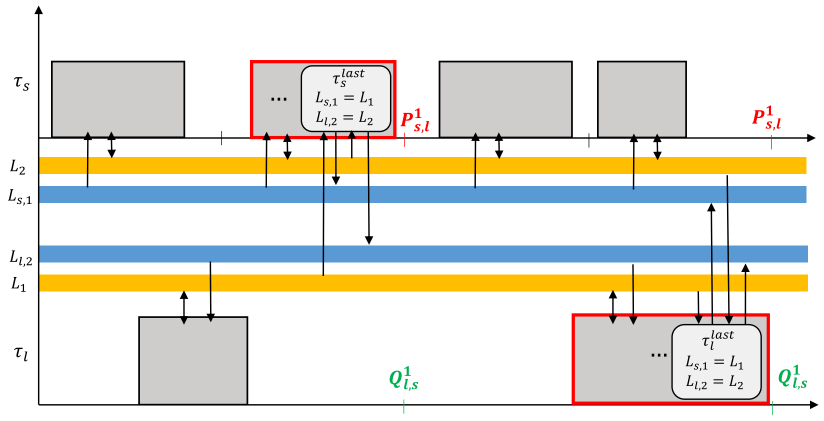


Figure 3.7 LET harmonic communication.

3.4.2 Non-Harmonic Synchronous Communication (NHSC)

When two communicating tasks do not have harmonic periods, a non-harmonic synchronous communication (NHSC) is established. The general formulas of Section 3.4 apply.

Like in the HSC case, the reading task of a shared label accesses a local copy instead of the original label. However, due to the misaligned activations of the communicating tasks, at least two copies of the same shared label are needed. A task-specific runnable is to be inserted at the end of the writer in order to update the copies of $I_{W,R}$ before the publishing point. If only one copy was used, a task could be writing it while the reader is reading it, leading to an inconsistent state. With two copies, instead, a reader reads a local copy, while the writer may freely write a new value for the next reading instance in a different buffer.

For example, consider a reading task τ_R and a writing task τ_W communicating through a shared variable L_2, with $2T_R = 5T_W$ as in Figure 3.8. There are two τ_R-local copies, $L_{R,2,1}$ and $L_{R,2,2}$, of the shared label L_2. The reading task τ_R reads from one of these copies instead of the original label. These copies are to be updated by the last runnable τ_W^{last} of the writing task. Note that τ_W directly writes to L_2 instead of a local copy.

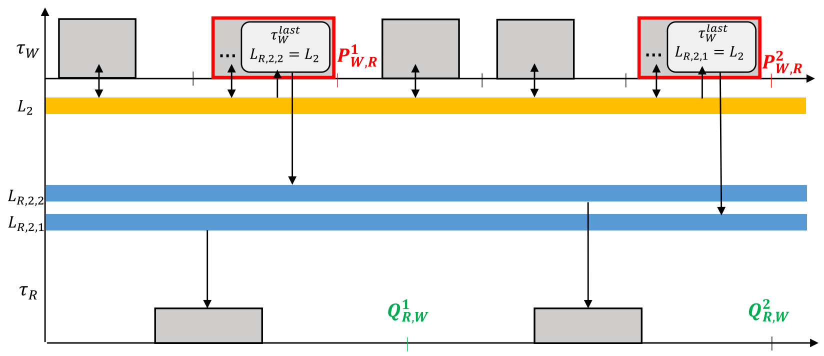


Figure 3.8 Non harmonic (NHSC): $2T_R = 5T_W$.

There might also be cases where three copies per labels are needed in order to fulfill the required determinism. Consider Figure 3.9 where $5T_R = 2T_W$. Note that τ_W may directly access L_1, while τ_R reads from one of the three copies $L_{R,1,1}$, $L_{R,1,2}$ or $L_{R,1,3}$, which are to be updated by runnable τ_W^{last}. An extra copy of L_1 is needed because the value computed by the second writing instance may be available either before or after the next reading point $Q^1_{R,W}$, depending on the response time of τ_W. If the second instance of τ_W finishes before (resp. after) $Q^1_{R,W}$, the reading instance after $Q^1_{R,W}$ would read the data of the second (resp. first) writing instance. Therefore, the value read at $Q^1_{R,W}$ is not deterministic, as it might correspond either to the first or to the second writing instance. Introducing a third buffer allows obtaining a deterministic behavior, where the values published by the first and second writing instances are always read at $Q^1_{R,W}$ and $Q^2_{R,W}$, respectively.

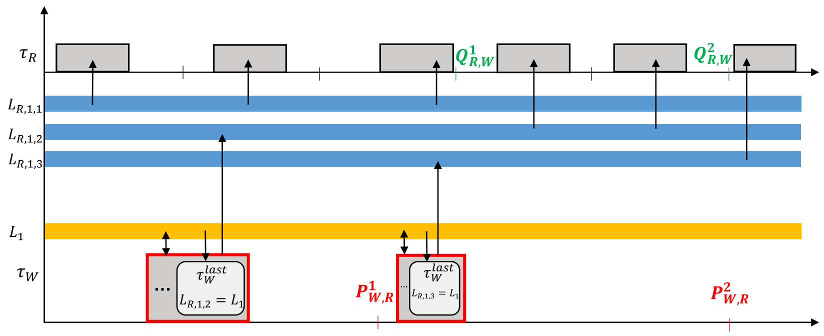


Figure 3.9 Non harmonic (NHSC): $5T_R = 2T_W$.

In general, this happens when a publishing instance has a best-case finishing time that precedes the next reading point. Let us define $w^n_{W,R}$ as the window of time between a publishing point $P^n_{W,R}$ and the next reading point $Q^n_{W,R}$. Then, using Equations (3.1) and (3.2),

$$w^n_{W,R} = Q^n_{W,R} - P^n_{W,R} = \left\lceil \frac{nT_{\max}}{T_R} \right\rceil T_R - \left\lfloor \frac{nT_{\max}}{T_W} \right\rfloor T_W. \tag{3.3}$$

It is worth pointing out that if a *HSC* is established, then $w^n_{W,R} = 0$. Furthermore, if the best-case response time of a publishing instance is smaller than the corresponding $w^n_{W,R}$, a third buffer is needed to store the new value.

As the type of LET communication is defined by the periods of the communicating task pair, a given tasks T_i can establish a HSC with one task and a NHSC with another. Thus, depending on the type(s) of estabished communication, the additional memory occupancy is given by the total number of local copies created for each label in I_i.

3.5 End-To-End Latency Analysis

In [7], four different end-to-end timing semantics are described to characterize the timing delays of effect chains given by multi-rate tasks communicating by means of shared variables. Depending on the application requirements, different end-to-end delay metrics can be of interest. Control systems driving external actuators are interested in the *age* of an input data, i.e., for how long a given sensor data will be used to take actuation decisions. For example, how long a radar or camera frame will be used as a valid reference by a localization or object detection system to perceive the environment: the older the frame, the less precise the system. Similar considerations are valid for an engine control or a fuel injection system, where correct actuation decisions depend on the *freshness* of sensed data.

Another metric of interest is the *reaction* latency to a change of the input, i.e., how long does it take for the system to react to a new sensed data. Multiple body and chassis automotive applications are concerned with this metric. For example, for a door locking system, it is important to know the time it takes to effectively lock the doors after receiving the corresponding signal. While, in this work. the main focus in on age latency, similar results apply also for reaction latency.

In [7], age latency is also referred to as last-to-last (L2L). However, no method is presented to formally compute these metrics.

As discussed in the previous subsection, the LET model requires that inputs and outputs be logically updated at reading and publishing points, respectively. To see its effect on end-to-end latency, let's apply its semantics to the examples shown in Figure 3.1 and Figure 3.2. The results are shown in Figure 3.10 and Figure 3.11, where it is easy to see that the age latency is the same in both cases. Clearly, this communication pattern allows not only deterministically setting publishing and reading points, but also setting the age latency of an effect chain to a fixed value, regardless of the actual execution time and core allocation of the involved communicating tasks. In this way, it is possible to achieve a higher level of predictability and a stronger consistency between the timing constraints (logical model) and the task execution (physical model), thus facilitating the design, implementation, test and certification process [38].

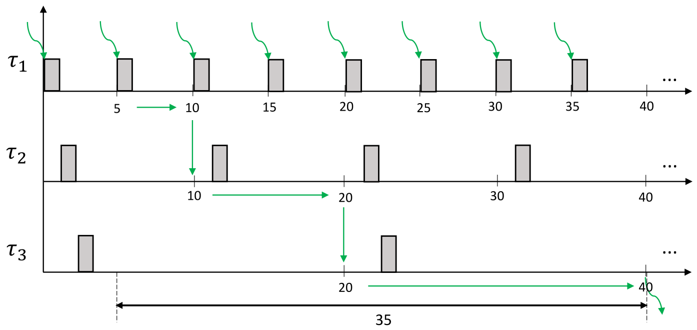


Figure 3.10 End-to-end effect chain with LET composed of three tasks with parameters: $T_1 = 5, T_2 = 10, T_3 = 20$ with $C_1 = C_2 = C_3 = 1$

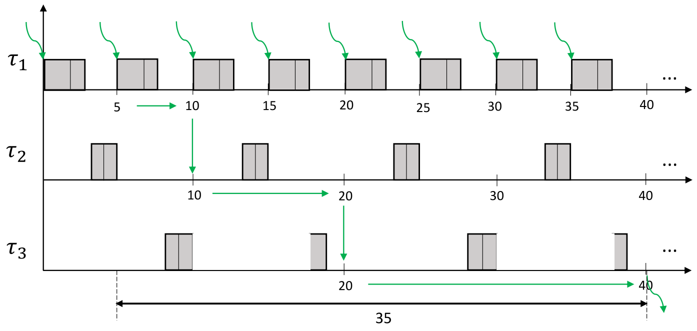


Figure 3.11 End-to-end effect chain with LET composed of three tasks with parameters: $T_1 = 5, T_2 = 10, T_3 = 20$ with $C_1 = 3, C_2 = 2, C_3 = 3$.

However, in the NHSC case, the above property does not hold. Consider the example shown in Figure 3.12, end-to-end latencies are either 18 or 21, with a worst-case age latency of 21. However, assigning an offset of 1 to τ_3, as depicted in Figure 3.13, reduces the worst-case age latency to 19, with zero jitter. This shows that by properly assigning offsets it is possible to improve control performance of NHSC, reducing the predictability gap in comparison with HSC by decreasing worst-case age latency and reducing jitter.

In order to understand how to properly assign offsets, Theorem 2 is generalized to consider offsets.

Theorem 2. *Given two communicating tasks τ_W and τ_R, with offsets O_W and O_R, respectively, the publishing and the reading points can be computed as*

$$P^n_{W,R} = O_W + \left\lfloor \frac{nT_{\max} + O_{max} - O_W}{T_W} \right\rfloor T_W \tag{3.4}$$

$$Q^n_{W,R} = O_R + \left\lceil \frac{nT_{\max} + O_{max} - O_R}{T_R} \right\rceil T_R \tag{3.5}$$

where $T_{\max} = \max(T_W, T_R)$, and $O_{\max}$ is the offset of the task with the largest period in the pair.

Proof. The proof is very similar to that of Theorem 1. If the writer τ_W has a smaller or equal period than the reader τ_R, i.e., $T_W \leq T_R$ as in Figure 3.14, there is one publishing and one reading point for each

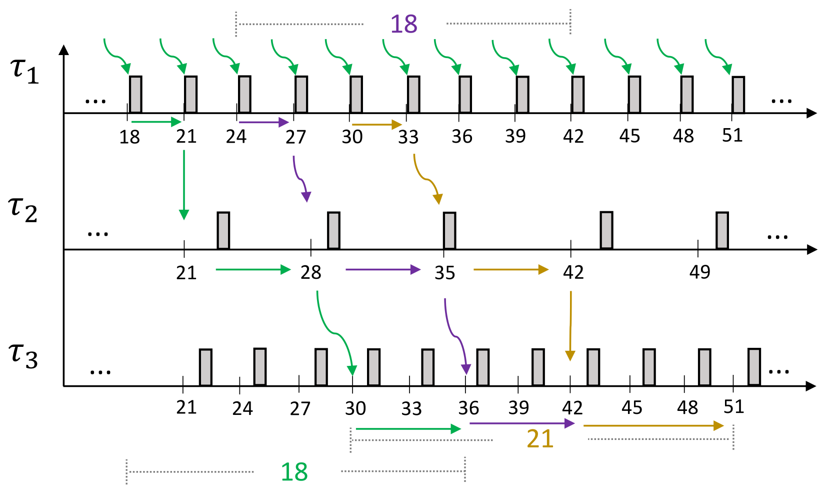


Figure 3.12 End-to-end effect chains with LET composed of three tasks with parameters $T_1 = 3, O_1 = 0, T_2 = 7, O_2 = 0, T_3 = 3, O_3 = 0$ with $C_1 = C_2 = C_3 = 1$

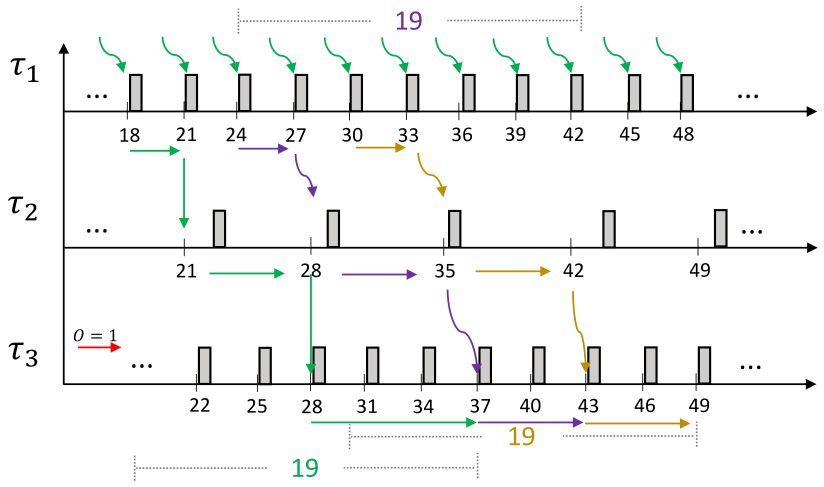


Figure 3.13 End-to-end effect chains with LET composed of three tasks with parameters $T_1 = 3, O_1 = 0, T_2 = 7, O_2 = 0, T_3 = 3, O_3 = 1$ with $C_1 = C_2 = C_3 = 1$

reading instance. Reading points again correspond to each reading task release, this time including offset: $Q^n_{W,R} = O_R + n \cdot T_R$, while publishing points correspond to the last writer release before such a reading instance, i.e., $P^n_{W,R} = O_W + \lfloor (n \cdot T_R + O_R - O_W)/T_W \rfloor \cdot T_W$.

Otherwise, when the writer τ_W has a larger period than the reader τ_R, i.e., $T_W \geq T_R$ as in Figure 3.15, there is one publishing and one reading point for each *writing* instance. Publishing points correspond to each writing task release, including offset: $P^n_{W,R} = O_W + n \cdot T_W$, while reading points correspond to the last reader release before such a writing instance, i.e., $Q^n_{W,R} = O_R + \lceil (n \cdot T_W + O_W - O_R)/T_R \rceil \cdot T_R$.

In both cases, the formula for $P^n_{W,R}$ and $Q^n_{W,R}$ are generalized by Equations (3.4) and (3.5). □

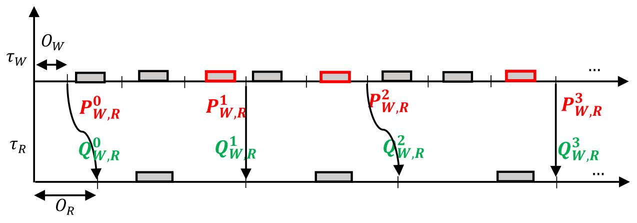


Figure 3.14 Publishing and reading points **with offsets** with $T_W = 2, O_W = 1, T_R = 5, O_R = 2$.

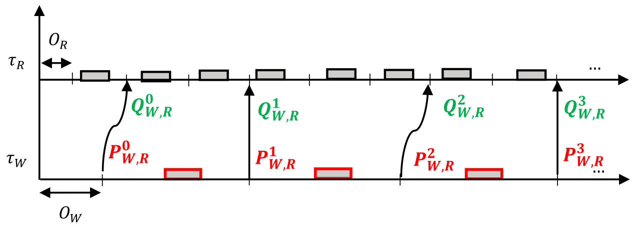


Figure 3.15 Publishing and reading points **with offsets** with $T_W = 5, O_W = 2, T_R = 2, O_R = 1$.

Clearly, the above theorem generalizes Theorem 1. When $T_W = T_R$, it can again be verified that each writing (resp. reading) task release correspond to a publishing (resp. reading) point.

In the following, an EC composed of η tasks is considered, where tasks are ordered according to their appearance in the considered effect chain, i.e., τ_1 is the first (writing) task, while τ_η is the last (reading) task in the EC. Let us define the hyperperiod H_{EC} of an EC as the least common multiple of the periods of the tasks composing the chain, i.e., $H_{EC} = LCM_{i=1}^{\eta}(T_i)$. Given all the publishing and reading points of the tasks composing an EC in its hyperperiod H_{EC}, the age latency of this chain is to be computed. There is a fixed number of possible communication paths in H_{EC}. To characterize them, we define the notion of *basic path*, as an interval starting from the end of the period of the first task in the *EC*, and finishing with the release of the last task in the *EC*. For example, in the EC of Figure 3.16 there are three basic paths in the highlighted hyperperiod $H_{EC} = 21$: $[21, 30], [27, 36]$ and $[33, 42]$. Note that if all tasks in the EC have harmonic periods, then there is only one basic path in the hyperperiod. In this case, the length of the basic path equals the sum of the periods of all tasks in the EC excluding the first task in the chain. In the examples of Figure 3.10 and Figure 3.11, there is only one basic path $[10, 20]$.

To determine basic path boundaries, given a reading point $Q^{x_\eta}_{\eta-1,\eta}$ at the end of an EC, Algorithm 1 shows how to derive the corresponding starting point $P^{x_2}_{1,2}$ at the beginning of the considered EC. As an example, consider the EC shown in Figure 3.16, where the communication between the last two tasks in the chain, τ_2 and τ_3, exhibits the three reading points highlighted in bold: 30, 36 and 42. The EC is formed by three tasks, i.e., $\eta = 3$, with no offsets. Algorithm 1 performs the following steps:

Algorithm 1 Calculating the start of a basic path

```
1: Input: Task set $\{\tau_i\}$, $Q^{x_\eta}_{\eta-1,\eta}$
2: Find $x_\eta$ in $Q^{x_\eta}_{\eta-1,\eta}$ using Eq. (3.5)
3: Compute $P^{x_\eta}_{\eta-1,\eta}$ using Eq. (3.4)
4: for i=$\eta$...3 do
5:     Find the largest $x_{i-1}$ in $Q^{x_{i-1}}_{i-2,i-1} < P^{x_i}_{i-1,i}$ using Eq. (3.5)
6:     Compute $P^{x_{i-1}}_{i-2,i-1}$ using Eq. (3.4)
7: Return $P^{x_2}_{1,2}$
```

(i) solve $Q^{x_3}_{2,3} = \lceil x_3 \cdot max(T_2, T_3)/T_3 \rceil \cdot T_3$ for x_3;

(ii) compute the corresponding $P^{x_3}_{2,3}$;

(iii) find the reading point $Q^{x_2}_{1,2}$ preceding $P^{x_3}_{2,3}$, by deriving the largest x_2 that satisfies $Q^{x_2}_{1,2} = \lceil x_2 \cdot max(T_1, T_2)/T_2 \rceil \cdot T_2 < P^{x_3}_{2,3}$; and

(iv) compute the start $P^{x_2}_{1,2}$ of the basic path.

Consider the example for reading point $Q^{x_3}_{2,3} = 30$. We then obtain:

(i) $30 = \lceil x_3 \cdot max(7, 3)/3 \rceil \cdot 3 = \lceil x_3 \cdot 7/3 \rceil \cdot 3$, and so $x_3 = 4$;

(ii) $P^{x_3}_{2,3} = P^4_{2,3} = \lfloor 4 \cdot max(7, 3)/7 \rfloor \cdot 7 = 28$;

(iii) solve $Q^{x_2}_{1,2} = \lceil x_2 \cdot max(3, 7)/7 \rceil \cdot 7 = 7 \cdot x_2 < P^4_{2,3} = 28$ for x_2, where the largest x_2 that satisfies the inequality is $x_2 = 3$;

(iv) compute $P^{x_2}_{1,2} = P^3_{1,2} = \lfloor 3 \cdot max(3, 7)/3 \rfloor \cdot 3 = 21$.

Repeating the same steps with $Q^{x_3}_{2,3} = 36$ (resp. $Q^{x_3}_{2,3} = 42$), we obtain $P^4_{1,2} = 27$ (resp. $P^5_{1,2} = 33$), matching the values in Figure 3.16.

Applying Algorithm 1 to all the reading points $Q^{x_\eta}_{\eta-1,\eta}$ corresponding to the communication between $\tau_{\eta-1}$ and τ_η in a given hyperperiod H_{EC} provides the boundaries of all meaningful basic paths to consider. Observe that paths starting with the same publishing point $P^{x_2}_{1,2}$ of a previous path are not to be considered.

Let us define $\dot{P}^n_{W,R}$ (resp. $\dot{Q}^n_{W,R}$) as the publishing (resp. reading) point between two tasks τ_W and τ_R in the n-th basic path of an EC. Then, the n-th basic path in the EC starts at $\dot{P}^n_{1,2}$ and ends at $\dot{Q}^n_{\eta-1,\eta}$. See Figure 3.16. In the example, the first basic path in this H_{EC} is defined by $\dot{P}^1_{1,2} = P^3_{1,2} = 21$ and $\dot{Q}^1_{2,3} = Q^4_{2,3} = 30$. Similarly, the bounds of the second (resp. third) basic path are $\dot{P}^2_{1,2} = P^4_{1,2} = 27$ (resp. $\dot{P}^3_{1,2} = P^5_{1,2} = 33$), and $\dot{Q}^2_{2,3} = Q^5_{2,3} = 36$ (resp. $\dot{Q}^3_{2,3} = Q^6_{2,3} = 42$).

Once the boundaries of the n-th basic path are known, its length θ^n_{EC} can be simply computed as $\theta^n_{EC} = \dot{Q}^n_{\eta-1,\eta} - \dot{P}^n_{1,2}$. If we assume the EC is triggered by the release of the first task in the chain, the age

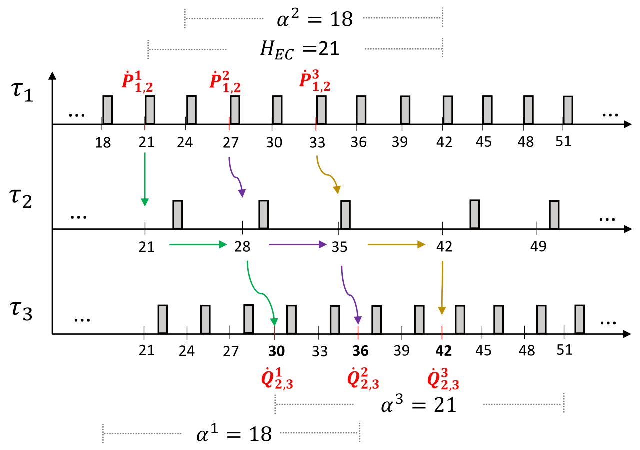


Figure 3.16 End-to-end effect chain characterization with LET composed of three tasks with parameters $T_1 = 3, O_1 = 0, T_2 = 7, O_2 = 0, T_3 = 3, O_3 = 0$.

latency α^n associated to the n-th basic path can then be computed by adding to the basic path length (i) the period T_1 of the first task in the EC, and (ii) the distance to the end of the next $(n+1)$-th basic path, where the output of the EC will eventually reflect a new input signal. That is,

$$\alpha^n = T_1 + \theta^n_{EC} + \dot{Q}^{n+1}_{\eta-1,\eta} - \dot{Q}^n_{\eta-1,\eta}. \tag{3.6}$$

The worst-case age latency $\alpha(EC)$ of the EC is then given by the maximum α^n over all basic paths in a hyperperiod of the EC.

$$\alpha(EC) = \max_{\forall n \in H_{EC}} \alpha^n. \tag{3.7}$$

In our previous example, $\theta^1_{EC} = \theta^2_{EC} = \theta^3_{EC} = 9$. Moreover,

$$\alpha^1 = T_1 + \theta^1_{EC} + \dot{Q}^2_{2,3} - \dot{Q}^1_{2,3} = 3 + 9 + 36 - 30 = 18,$$

$$\alpha^2 = T_1 + \theta^2_{EC} + \dot{Q}^3_{2,3} - \dot{Q}^2_{2,3} = 3 + 9 + 42 - 36 = 18, \text{ and}$$

$$\alpha^3 = T_1 + \theta^3_{EC} + \dot{Q}^4_{2,3} - \dot{Q}^3_{2,3} = 3 + 9 + 51 - 42 = 21,$$

as illustrated in Figure 3.16. Thus, $\alpha(EC) = max(\alpha^1, \alpha^2, \alpha^3) = max(18, 18, 21) = 21$.

3.6 Heuristics

In the previous sections, it was shown how an offset-aware LET analysis may be used to improve real-time performance. For this reason, given a schedulable task set, an offset assignment method that shortens the age latency of a selected EC, possibly making it constant throughout the whole execution of the tasks involved, is of interest. This could be particularly useful for automotive applications where there are no design constraints on offsets. It is worth mentioning that while offset assignment can shorten the age latency of a particular EC, it might also potentially lengthen the end-to-end latency of another chain. On the other hand, effect chains, very much like tasks, are also prioritized, i.e. not all latencies have the same importance.

One EC might be particularly important for the stability and control of the system, while other ones may be related to less critical activities. The main focus is hereafter on the latency minimization problem of a selected EC. The method can also be applied to multiple ECs as long as they have no tasks in common. The latency minimization problem of multiple ECs having one or more tasks in common is left as a future work.

Without loss of generality, offsets can be normalized assuming $O_1 = 0$ and $O_i \in [0, T_i\rangle, \forall i \in [2, \eta]$. A brute force approach is not desirable for longer chains or when the periods of the tasks involved are large, since the number of combinations can get up to $\prod_{i=2}^{\eta} T_i = \mathcal{O}((max_{j=2}^{\eta} T_j)^{\eta-1})$ for chains composed of different tasks. We therefore derive a heuristics for a convenient offset assignment that can be conveniently used to improve control performance within a reasonable computational complexity.

Equation 3.6 can be rewritten as

$$\alpha^n = T_1 + \dot{Q}^n_{\eta-1,\eta} - \dot{P}^n_{1,2} + \dot{Q}^{n+1}_{\eta-1,\eta} - \dot{Q}^n_{\eta-1,\eta} = T_1 + \dot{Q}^{n+1}_{\eta-1,\eta} - \dot{P}^n_{1,2}$$

From Theorem 2, it follows that

$$\alpha^n = T_1 + \left\lceil \frac{(n'+1)max(T_{\eta-1}, T_\eta) + O^{\eta-1,\eta}_{max} - O_\eta}{T_\eta} \right\rceil T_\eta + O_\eta - \left\lfloor \frac{n''max(T_1, T_2) + O^{1,2}_{max} - O_1}{T_1} \right\rfloor T_1 - O_1$$

where $O^{i,j}_{max}$ is the offset of the task with the largest period among τ_i and τ_j, while n' and n'' are numbers defined by the alignment, periods and offsets of the tasks composing the n-th basic path of the EC. Let us define two integer values $k' = \left\lceil \frac{(n'+1)max(T_{\eta-1}, T_\eta) + O^{n-1,n}_{max} - O_\eta}{T_\eta} \right\rceil$ and $k'' = 1 - \left\lfloor \frac{n''max(T_1, T_2) + O^{1,2}_{max} - O_1}{T_1} \right\rfloor$. Then,

$$\alpha^n = k'T_\eta + k''T_1 + O_\eta - O_1$$

Recalling that $O_1 = 0$,

$$\alpha^n = k'T_\eta + k''T_1 + O_\eta$$

The last equation shows that the age latency of an EC can be computed as the sum of a multiple of the period of the first and of the last task in the chain, plus the offset of the last task. This does not mean that the tasks in the middle of the chain have no influence on the age latency. Their contribution is hidden within k' and k'', which may increase or decrease the age latency by integer multiples of the period of the first and last task in the EC.

Algorithm 2 proposes a heuristic approach to assign offsets that considers only the last d tasks in the EC, starting from the last task τ_η. The remaining $\eta - d$ tasks are assumed to have a null offset. In this way, the total number of combinations is reduced to $\prod_{i=\eta-d+1}^{\eta} T_i = \mathcal{O}((max_{j=\eta-d+1}^{\eta} T_j)^d)$. Note that $d < \eta$ and $O_1 = 0$. Furthermore, $d = \eta - 1$ is equivalent to the brute force approach.

The complexity can be further reduced by considering only non-equivalent offset assignments. Two asynchronous situations are defined to be equivalent, if they have the same periodic behavior. For two tasks τ_1 and τ_2, two choices O_2 and O'_2 are equivalent if they produce the same relative phasing, i.e.,

$$\exists k \in \mathbb{N} : O_2 \mod T_1 = (O'_2 + kT_2) \mod T_1.$$

As an example, consider τ_1 and τ_2 with $T_1 = 8$, $T_2 = 12$, $O_1 = 0$ and $O_2 < T_2$. The offset assignment $O_2 = 0$ is equivalent to $O'_2 = 4$ and to $O''_2 = 8$, since they all lead to the same job interleaving throughout the hyperperiod $LCM(T_1, T_2) = 24$. Similarly, $O_2 = 1$ is equivalent to $O'_2 = 5$ and to $O''_2 = 9$. In general, two offset assignments $O2$ and $O2'$ are equivalent if $O_2 = O'_2 \mod GCD(T1, T2)$, as shown in [34]. Therefore, it makes sense to consider only the offsets in $[0, GCD(T_1, T_2)\rangle$.

For later tasks in the effect chain, similar considerations apply by considering their alignment with respect to the hyperperiod of earlier tasks. For example, for task τ_3, it is sufficient to consider its non-equivalent alignments with respect to the hyperperiod of τ_1 and τ_2, i.e., $O_3 \in [0, GCD\{T_3, LCM(T_1, T_2)\}\rangle$. In general, assuming the offsets $O_1, \ldots, O_{i-1}$ have been set, for τ_i it is sufficient to consider

$$O_i \in [0, GCD\{T_i, LCM_{j=1}^{i-1} T_j\}\rangle, \forall i \in [2, \eta]$$

Thus, the number of possible combinations of the brute force approach is reduced to

$$\prod_{i=2}^{\eta} GCD\left\{T_i, LCM_{j=1}^{i-1} T_j\right\}.$$

Since $x \cdot y = GCD(x, y) \cdot LCM(x, y)$, this simplifies to

$$\prod_{i=2}^{\eta} \frac{T_i \cdot LCM_{j=1}^{i-1} T_j}{LCM(T_i, LCM_{j=1}^{i-1} T_j)} = \prod_{i=2}^{\eta} \frac{T_i \cdot LCM_{j=1}^{i-1} T_j}{LCM_{j=1}^{i} T_j} = \frac{\prod_{i=1}^{\eta} T_i}{LCM_{i=1}^{\eta} T_i}.$$

The complexity of the brute force approach is then $\prod_{i=1}^{\eta} T_i / H_{EC}$. This entails a significant reduction in the complexity, especially in case of mutually prime periods. Note that in case all periods are mutually prime, there is only one configuration to check.

Similarly, the number of offset assignments leading to non-equivalent asynchronous situations given by the d-offset assignment algorithm can be derived as

$$\prod_{i=\eta-d+1}^{\eta} GCD\left\{T_i, LCM_{j=1}^{i-1} T_j\right\} = \prod_{i=\eta-d+1}^{\eta} \frac{T_i \cdot LCM_{j=1}^{i-1} T_j}{LCM(T_i, LCM_{j=1}^{i-1} T_j)}$$

$$= \prod_{i=\eta-d+1}^{\eta} \frac{T_i \cdot LCM_{j=1}^{i-1} T_j}{LCM_{j=1}^{i} T_j} = \frac{LCM_{i=1}^{\eta-d} T_i \cdot \prod_{i=\eta-d+1}^{\eta} T_i}{LCM_{i=1}^{\eta} T_i}.$$

Let $H_d = H_{EC} / LCM_{i=1}^{\eta-d} T_i$. The complexity of the d-offset assignment algorithm is then

$$\frac{\prod_{i=\eta-d+1}^{\eta} T_i}{H_d}.$$

Algorithm 2 d-Offset assignment

```
1: Input: Task set $\{\tau_i\}$, depth $d$
2: Assign $O_i = 0$, $\forall i \in [1, \eta - d]$
3: Consider all combinations of offset assignments leading to non-equivalent asynchronous situations $\forall \tau_i, i \in [\eta - d + 1, \eta]$
4: for each combination do
5:     Compute the worst-case age latency of this combination using Equation (3.7)
6: Return the maximum age latency among all combinations
```

3.7 Evaluation

Having established a thorough analytical characterization of the end-to-end latencies of effect chains under the Logical Execution Time communication model, an experimental characterization of the effectiveness of LET in improving the control performance by reducing the variability of the end-to-end latency is hereafter provided. Moreover, it is shown how the proposed offset assignment technique can be adopted to further reduce such a variability in case an even tighter control performance is needed.

To this end, two sets of experiments are performed. The first set considers an industrial case study from the automotive domain, providing a characterization of the analytical performance of LET in a representative setting. The second set of experiments is based on randomly generated effect chains composed of tasks with a different period distribution, to characterize the effectiveness of the offset assignment methods in further reducing jitter. The experiments were conducted on top of a quad-core processor i7-4720HQ @ 2.6 GHz with 16GB of RAM.

It is worth pointing out that while none of the previous works exposed in Section 3.2 takes offsets into consideration to compute the end-to-end latency of LET ECs, the existing offset assignment techniques presented in the same section, such as [34], cannot be applied, as they have a different target, i.e., making a task set schedulable, or reducing the worst-case response time of an already schedulable task set, on a uniprocessor. Hence, the next experiments are not compared to the related work.

3.7.1 Industrial Case Study

To provide a representative characterization of the end-to-end age latency introduced by the LET model, an automotive application representing an engine control system, as detailed by Kramer et al. in [2], is considered. The application is composed of multiple tasks partitioned onto four cores. The periods of the tasks are {1, 2, 5, 10, 20, 50, 100, 200, 1000}ms. Tasks are composed of 1250 runnables that access about 1500 different labels. The effect chains created by tasks reading/writing a common shared variable are considered. Based on this setting, there are over 500 ECs with length $3 \leq \eta \leq 8$.

Figure 3.17 shows the average value of the worst-case age latency $\alpha(EC)$ obtained with LET among the considered effect chains for each EC length. As can be expected, the age latency increases proportionally with the length of the chain. An analysis on the individual EC shows that the worst-case age latency is never smaller than the sum of the periods of the tasks composing the considered EC.

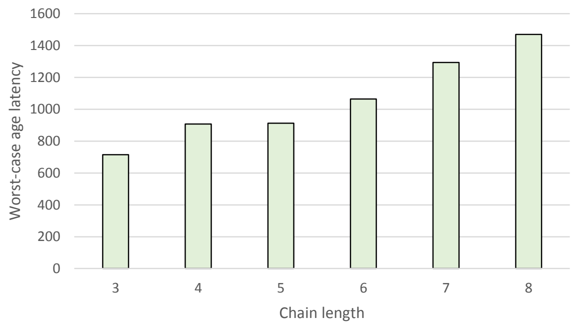


Figure 3.17 Average value of the worst-case age latency for the considered effect chains.

More interestingly, the LET model allows significantly reducing the jitter of the end-to-end latency of an effect chain. Let us define the jitter of an EC as

$$J(EC) = \max_{\forall n \in H_{EC}} \alpha^n - \min_{\forall n \in H_{EC}} \alpha^n.$$

Figure 3.18 shows the normalized jitter $(J(EC)/\alpha(EC))$, i.e., the ratio of the jitter over the age latency. Both average and worst-case values over all effect chains are shown for each considered length. The average jitter is always below 1%, confirming that LET is very effective in reducing end-to-end latency variability, with longer chains exposing a slightly smaller normalized jitter. However, for all considered EC lengths, there are different cases where the jitter is above 10% of the overall age latency.

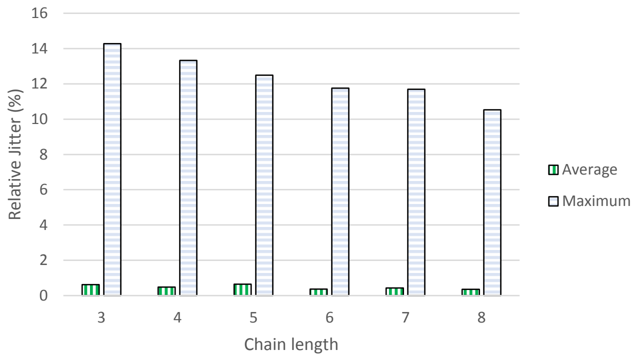


Figure 3.18 Average and maximum values of the normalized jitter for the considered effect chains.

In order to further improve the end-to-end control performances, the offset assignment method of Algorithm 2 is applied. To compute the offset for *all* the effect chains, the algorithm required about 30 minutes for $d = 1$ and 3 hours for $d = 2$. Even using a small depth $d = 1$ (resp. $d = 2$) allowed improving the worst-case age latency for 206 (resp. 377) out of the 577 considered effect chains. The improvement obtained for these ECs is shown in Figure 3.19 both for $d = 1$ and $d = 2$. In general, a small depth allows significantly improving the age latency of shorter chains (10% on average, 30% in the best case). A larger depth value allows improving the latency of longer chains, by paying a higher computational cost.

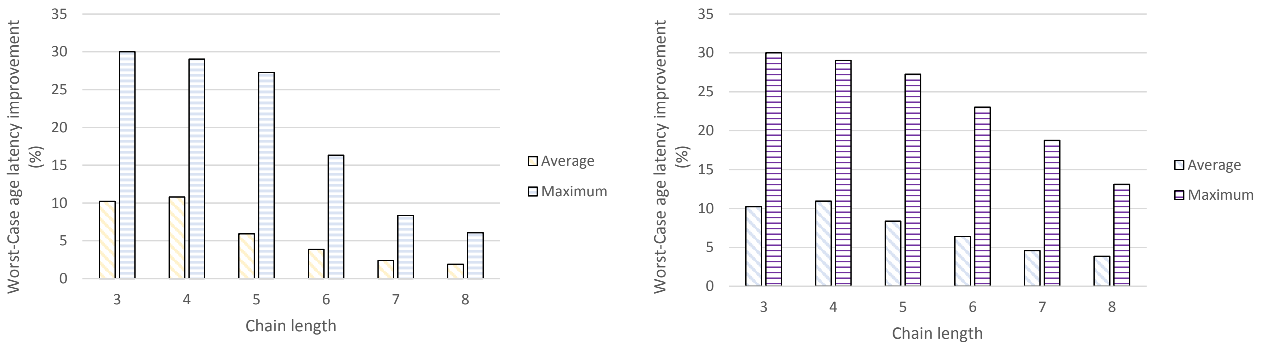


Figure 3.19 Average and maximum age latency improvement provided by the offset assignment heuristics with depth $d = 1$ (left) and $d = 2$ (right).

Another interesting effect of the offset assignment technique is to decrease the jitter. Note that effect chains composed of harmonic tasks have all a null jitter. In the considered automotive use case, the great majority of effects chains are harmonic, due to the selection of task periods. Therefore, the average and maximum jitter shown in Figure 3.18 is due to a few non harmonic effect chains, 32 of which had a non null jitter. With the suggested offset assignment method, the jitter is reduced to zero for 9 of them with $d = 1$. Figure 3.20 shows the average and best-case improvement in the jitter normalized with respect to the age latency, i.e., $\Delta J(EC)/\alpha(EC)$, for the case with $d = 2$.

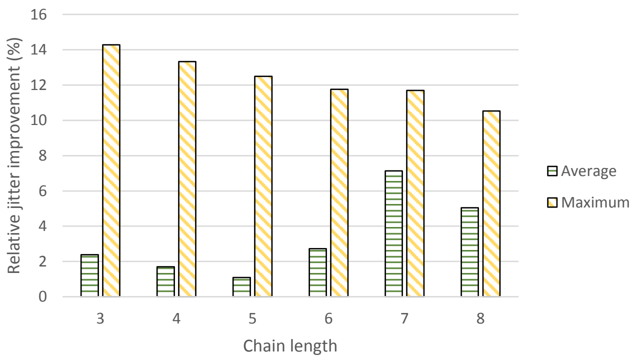


Figure 3.20 Average and maximum normalized jitter improvement provided by the offset assignment heuristics with depth $d = 2$

3.7.2 Randomly Generated Workloads

A second set of experiments is provided to characterize the efficiency of the proposed heuristics with respect to a brute force approach. Unfortunately, the industrial use case adopted in the previous section is not amenable to a brute force approach because of the large range of task periods, which makes it too computationally expensive. Therefore, 500 ECs are synthetically generated. These ECs are composed of randomly generated tasks with periods uniformly distributed in $[1, 10]$. ECs with $\eta \in [3, 6]$ are only considered. Note that there is no need to generate utilizations and execution times, since tasks are assumed to always complete before their (implicit) deadlines.

To understand the performance of the proposed heuristics in exploring the design space to select an optimal offset assignment, a characterization based on the depth d value that allows achieving an optimal end-to-end latency is provided. In this experiment, first the optimal offsets were computed by using a brute force approach. Then, Algorithm 2 was run with increasing depth values, starting with $d = 1$, to compare the resulting worst-case age latency with that of the brute force algorithm. When they matched, the algorithm was stopped recording the d value. Figure 3.21 shows the normalized depth r, defined as the ratio between the resulting d and the length of the EC, i.e., $r = d/\eta$. Interestingly, an optimal assignment is obtained even with a very small depth. In more than 60 % of the cases, r is lower than or equal to 1/3, indicating that the proposed heuristics can be conveniently adopted to reduce age latencies even using a small depth d. The computation time of the offset minimization algorithm was variable, varying from a few seconds to a few hours, depending on the chain depth and on the periods of the communicating tasks.

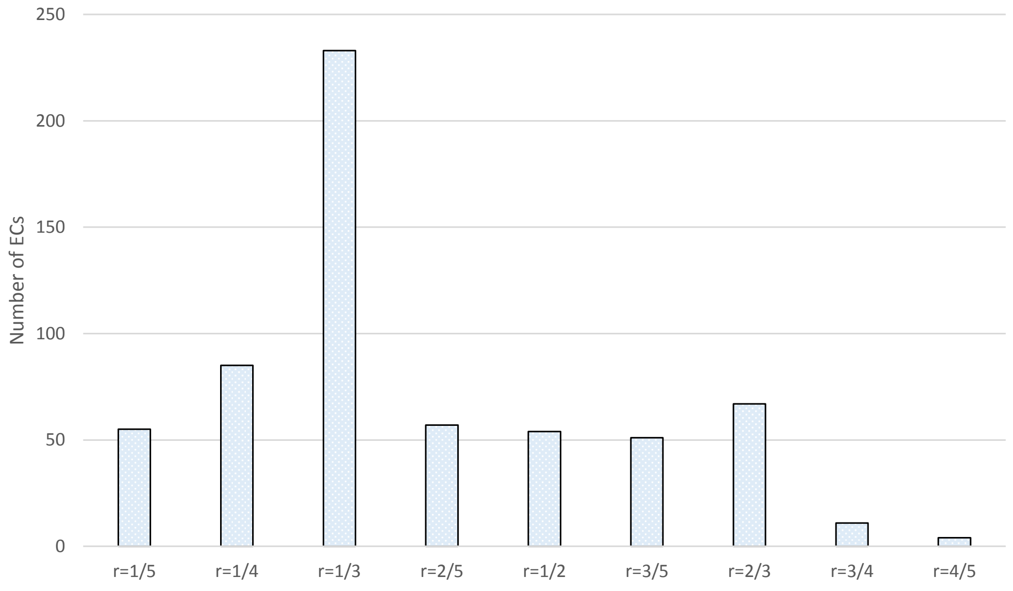


Figure 3.21 Heuristics vs. Brute force approach.

3.8 Summary

In this chapter, an analytical characterization of the end-to-end latency of effect chains composed of periodic tasks communicating using the LET model was presented. A closed formula expression was provided to compute reading and publishing points where the actual communication between tasks takes place. Based on these points, the end-to-end latency may be computed considering the basic paths of an effect chain within a hyperperiod of the communicating tasks. The analysis was then extended to consider task offsets. An offset assignment method was suggested to further improve the determinism of the end-to-end latency, reducing control jitter.

We also showed the effectiveness of the LET model in achieving a more deterministic end-to-end communication delay for an industrial case study from the automotive domain. We presented a set of experiments showing that the jitter of the end-to-end latency with the LET model is in average within 1% for representative task sets, analytically confirming the control determinism of the LET model. However, non harmonic effect chains may have significantly higher jitters. In these cases, a considerable jitter reduction can be obtained using the proposed offset assignment heuristics. Due to the fact that a deterministic and stable EC latency is very important for control algorithms, from now on, it is assumed that communication among periodic tasks takes place by means of the LET communication model.

While in this chapter, the motive behind the presented analysis was the need for a deterministic end-to-end latency for automotive control algorithms, in the following chapter, another trend in the automotive industry will lead to the introduction of fixed priority servers as a means to provide temporal isolation of automotive applications.

CHAPTER 4
FIXED PRIORITY SERVERS

Modified from a journal paper[39] published in the Journal of Systems Architecture: Embedded Software Design (JSA)

Jorge Martinez[10,11], Dakshina Dasari[12], Arne Hamann[13], Ignacio Sañudo[14] , Marko Bertogna[15]

In the automotive domain, existing scheduling primitives do not suffice for complex integration scenarios involving heterogeneous applications with diverse timing requirements. Thus, there is a renewed interest to establish server-based scheduling as a mainstream scheduling paradigm in automotive software development, as they facilitate timing isolation between different software components[11].

Real-time servers have been widely explored in the scheduling literature to predictably execute aperiodic activities, as well as to allow hierarchical scheduling settings. Due to their intrinsic temporal properties, as previously mentioned, there is a considerable interest for the adoption of fixed priority real-time servers in the automotive domain, as a way to implement more efficient reservation mechanisms than TDMA-based methods [10].

In this chapter, the focus of interest is on Fixed Priority Servers, namely Polling Periodic (PPS), extended Polling Periodic, Deferrable (DS) and Sporadic Server (SS). Despite their popularity, only sufficient schedulability conditions exist for real-time systems scheduled with these kinds of servers. Thus, formal operations and basic concepts, that are the pillars of the exact response time analysis presented in the next chapter, are introduced.

4.1 Background and Related Work

Different fixed priority server algorithms have been proposed in the literature. Lehoczky et al. introduced the *Polling Periodic Server* and *Deferrable Server* in [14], defining a capacity to schedule aperiodic jobs. If there are no requests at the time of invocation, the capacity of a polling server is depleted until the next server period. The major drawback of the PPS is that the capacity of the server is lost whenever no aperiodic request exists when the server becomes active. A simple improvement consists to decrease, and not reset, the capacity in the situation described above. This type of server is known as extended Polling Periodic

[10]Graduate student at the University of Modena and Reggio Emilia
[11]Primary researcher and author
[12]Researcher at Robert Bosch GmbH
[13]Researcher at Robert Bosch GmbH
[14]Postgraduate researcher at the University of Modena and Reggio Emilia
[15]Full Professor at the Univeristy of Modena and Reggio Emilia

Server [40]. The DS also overcome the aforementioned drawback of the PPS by preserving its capacity until the end of the period. In any of the three cases, the server capacity is fully restored at each server period.

In the *Sporadic Server* introduced in [13], the server budget is instead replenished one period after the server activation, and only by the amount of capacity that has been consumed in that time interval. In more detail, a sporadic server S, with budget C_s and period T_s, works as follows:

1. The server is in an *Active* state, when it has pending jobs to execute and it has a positive remaining budget.

2. The server is in an *Idle* state, when there is no workload or its budget is exhausted.

3. Initially, the server is *Idle* and its budget is C_s. When the server becomes *Active* at time t, its replenishment time is set to $t + T_S$.

4. When the server becomes *Idle* at time t', the replenishment amount corresponding to the last replenishment time is computed as the amount of capacity consumed by S since its last activation, i.e. in $[t, t')$.

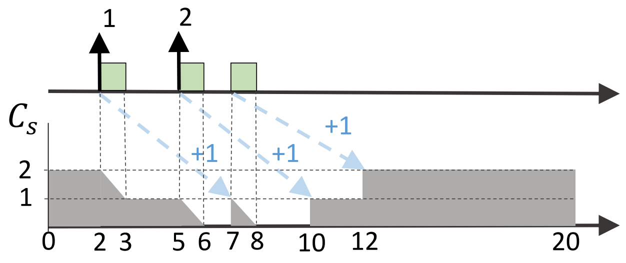


Figure 4.1 Example of a Sporadic Server ($C_s = 2$, $T_s = 5$.)

An example of a Sporadic Server with budget $C_s = 2$ and period $T_s = 5$ serving two jobs, whose arrivals are represented by upward arrows, is shown in Figure 4.1. Numbers beside the arrows indicate the computation times associated with the requests. At time $t = 2$, the first request arrives, and since $C_s > 0$, the server becomes *active* and the request receives immediate service. Thus, a replenishment time is set to $t + T_s = 2 + 5 = 7$. The job is completed at $t' = 3$, and so the corresponding replenishment amount is equal to the budget consumed in $[2, 3)$, i.e. 1. Replenishments are depicted by the blue arrows in the inset. At time $t = 5$, the server becomes active again and a new replenishment time is set to $t + T_s = 10$. At $t' = 6$, the budget is exhausted and so the server becomes *idle*. The corresponding replenishment amount is then the capacity consumed in $[5, 6)$, i.e. 1. At $t = 7$, the budget is replenished, and hence the job can continue executing until its completion at $t' = 8$. In this way, a new replenishment time is set to $t + T_s = 12$ with a replenishment amount of 1 as shown in the figure.

Bernat and Burns in [21] showed through simulations that the Sporadic and the Deferrable server perform equivalently well. Later, Faggioli et al. proposed in [41] an *extended* Sporadic Server algorithm to improve the runtime behavior reducing the system overhead related to replenishing events.

When servers are used to handle aperiodic requests, it is essential to validate if the performance constraints of these requests are met. To this end, in [42] Buttazzo introduced the notion of *aperiodic guarantee* which specifies a condition to verify whether an aperiodic job of known execution time, arriving at a certain time instant, meets its deadline. Since such aperiodic guarantees consider only a single job instead of a stream of jobs, they nullify the effect of self-interference. On the other hand, Abeni and Buttazzo in [43] presented another approach to quantify the service provided by servers by using statistical analysis in order to compute *Quality of Service* (QoS) guarantees expressed in terms of probability for each served job to meet a deadline. This work was done for a dynamic priority server algorithm proposed in [44], namely the *Constant Bandwidth Server*, and is not suitable for hard real-time tasks. In [45] Kumar et al. proposed a resource model of the constant bandwidth server to calculate the worst-case delay suffered by a given stream of jobs when scheduled via a server. They also showed that a similar approach can be used to characterize fixed priority servers. As these upper bounds are independent of the rest of the system, they yield pessimistic results. Furthermore, they assumed that only one aperiodic task is scheduled per server.

In [46], Kuo and Li introduced the idea of using sporadic servers to implement a two-level hierarchical scheme for scheduling independently developed real-time applications inspired by the *open system architecture* developed by Deng and Liu [47]. Later, Saewong et al [48] provided a response time analysis for hierarchical systems composed of Deferrable or Sporadic servers, assuming a worst-case scenario where a server's capacity has been depleted when the task of interest arrives.

Based on the same pessimistic assumption, Lipari and Bini [49] provided an alternative response time formulation using a server availability function. Almeida and Pedreiras [50] further improved previous work by considering the maximum latency that a server could suffer. Inspired by this work, Davis and Burns [17] made use of the busy-window analysis to provide tighter bounds by refining the critical instant, i.e. the pattern of server and task execution that leads to the worst-case response time of the task. Balbastre et al. [51] pointed out scenarios where the previous critical instant of Davis and Burns does not hold, and provided a method to reproduce a plausible worst-case scenario by assigning offsets to servers and tasks. As the number of servers increases, however, this approach becomes computationally intractable as signaled by the authors. While the work in [51] covers Polling and Deferrable Servers, none of the aforementioned works provides an exact and computationally tractable response time analysis of tasks scheduled by fixed priority servers. Thus, this work presents a framework in order to analyze tasks mapped onto, as well as tasks co-scheduled with, fixed priority servers.

4.2 System Model

The model is assumed to be composed of tasks and servers scheduled according to a fixed priority preemptive policy. A task is said to be *bound*, if the task is scheduled by a server, otherwise it is *unbound*. A server can host multiple bound tasks. Each task τ_i, consists of a stream of jobs, $J_{i,j}$, characterized by an arrival time $a_{i,j}$, a start time $s_{i,j}$, a finishing time $f_{i,j}$, and a computation time $c_{i,j}$. Moreover, each periodic task, τ_i, is characterized by a tuple $(T_i, C_i, D_i, O_i, \Pi_s)$, where T_i is its period, $C_i = \max_j\{c_{i,j}\}$ its WCET, D_i its deadline, O_i is the initial offset, and Π_s is its priority. Unless otherwise stated, deadlines are implicit, i.e $D_i = T_i$. The utilization of a task, U_i, is the ratio of its WCET to its period, i.e. $U_i = C_i/T_i$. Note that unlike previous chapters, aperiodic tasks are now considered as well.

Furthermore, a server s is characterized by a tuple (T_s, C_s, Π_s), where T_s is its period, C_s its capacity or budget, and Π_s its priority. The budget is replenished according to a fixed priority server algorithm. The utilization of a server, U_s, is the quotient of its budget and its period, i.e. $U_s = C_s/T_s$, and $\overline{U}_s$ represents the aggregated utilization of its tasks, i.e. $\overline{U}_s = \sum U_i \ \forall i$ in s. The *hyperperiod*, is the least common multiple (LCM) of all servers and tasks' periods.

4.3 Motivation

A highly desirable property of Hierarchical Fixed Priority Preemptive Systems based on any kind of fixed priority server is that an application that has been assigned a specific amount of a CPU time should have access to this regardless of other applications co-scheduled on the system. This property is called temporal isolation and is of utmost importance to the automotive domain [11].

Thus, considerable effort has been made in order to provide an exact schedulability analysis to hierarchical systems with fixed priority scheduling policies used both at global and local schedulers. In particular, Davis and Burns [17] derived a response-time analysis for a task executing under either a Polling Periodic, or Deferrable or Sporadic Server. In the case of a task τ_i served by a Sporadic Server s, the analysis is as follows:

1. Determine the *critical instant* for τ_i, i.e. the pattern of execution of other tasks and servers leading to the worst-case response time of τ_i.

2. Obtain the load due to the execution of task τ_i and tasks of higher priority in the server, $L_i(w)$, released in a busy window of length w starting at the critical instant. This task load is given by:

$$L_i(w) = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w + J_j}{T_j} \right\rceil C_j$$

where $hp(i)$ is the set of tasks that have priorities higher than τ_i, and J_j is the release jitter of a task τ_j in $hp(i)$. If τ_j is only ever released at the same time as its server, $J_j = 0$. Otherwise $J_j = T_s - C_s$.

3. Compute the gaps left by the server in any complete period entirely contained in the considered busy window: $(\lceil L_i(w)/C_s \rceil - 1)(T_s - C_s)$.

4. Calculate the interference $I(w)$ from higher priority servers in the final server period at the end of the busy window. This interference is given by:

$$I(w) = \sum_{\forall x \in hp(S)} \left\lceil \frac{max(0, w - (\left\lceil \frac{L_i(w)}{C_s} \right\rceil - 1)T_s)}{T_x} \right\rceil C_x$$

where $hp(S)$ is the set of servers with higher priority than server S.

Consider a system comprising two Sporadic Servers, with parameters given in Table 7.1 (left), where the two highest priority tasks associated with the lower priority server LP are characterized as in Table 7.1 (right)[16]. To compute the worst-case response time of τ_1, R_1, the critical instant, according to [17], occurs when:

i The capacity of LP has been exhausted by lower priority task τ_2 as early as possible.

ii τ_1 arrives just after the server's capacity has been exhausted.

iii The capacity of the LP server is replenished at the start of its next period but with a delayed execution of the server due to interference from the HP server.

Table 4.1 Parameters of the servers and tasks of the counterexample.

Server	C_s	T_s
HP	2	5
LP	8	20

Task	C_i	T_i
1	10	50
2	8	100

Given the aforementioned critical instant, R_1 is given by $w + T_{LP} - C_{LP}$, where the first addend is obtained by solving $w = L_1(w) + (\lceil L_1(w)/C_{LP} \rceil - 1)(T_{LP} - C_{LP}) + I(w)$. Since τ_1 is the highest priority server in LP and HP is the only server with priority higher than LP, $L_1(w) = C_1$ and $I(w) = (\lceil max(0, w - (\lceil C_1/C_{LP} \rceil - 1)T_{LP})/T_{HP} \rceil)C_{HP}$. Hence, $w = C_1 + (\lceil C_1/C_{LP} \rceil - 1)(T_{LP} - C_{LP}) + (\lceil max(0, w - (\lceil C_1/C_{LP} \rceil - 1)T_{LP})/T_{HP} \rceil)C_{HP} = 10 + (\lceil 10/8 \rceil - 1)(20-8) + (\lceil max(0, w - (\lceil 10/8 \rceil - 1)20)/5 \rceil)2 = 22 + 2(\lceil max(0, w - 20)/5 \rceil)$. The recurrence starts with $w = C_1 + (\lceil C_1/C_{LP} \rceil - 1)(T_{LP} - C_{LP}) = 22$ and ends with $w = 24$. As a result $R_1 = w + T_{LP} - C_{LP} = 24 + 20 - 8 = 36$.

[16]A setting similar to the one shown in Table 7.1 but comprising two Deferrable Servers is to be found in [17]

A similar criteria is used to obtain the worst-case response time, R_2. While use of [17] results in $R_1 = 36$ and $R_2 = 68$, the Gantt chart of Figure 4.2 reveals that R_1 and R_2 (highlighted in bold red text) are actually 24 and 44 respectively. Moreover, the inset shows that the aforementioned critical instant does not occur. In particular it can be seen that conditions (i) and (ii) of the critical instant for τ_1 do not hold, as the capacity of the LP server is never exhausted by τ_2 upon arrival of τ_1.

A similar explanation justifies the pessimism of R_2. This example and experiments conducted later clearly illustrate that the existing analysis is only sufficient but not necessary for any of the three kinds of servers mentioned above. Note that in [17], it is assumed that each server hosts at least one soft real-time task that may consume the capacity of the server making conditions (i) and (ii) more plausible. Hence [17] cannot be exact without the aforementioned assumption. Furthermore, while the analysis in [17] can be applied to hard real-time tasks only, the method presented in this article can be applied to both soft and hard real-time tasks.

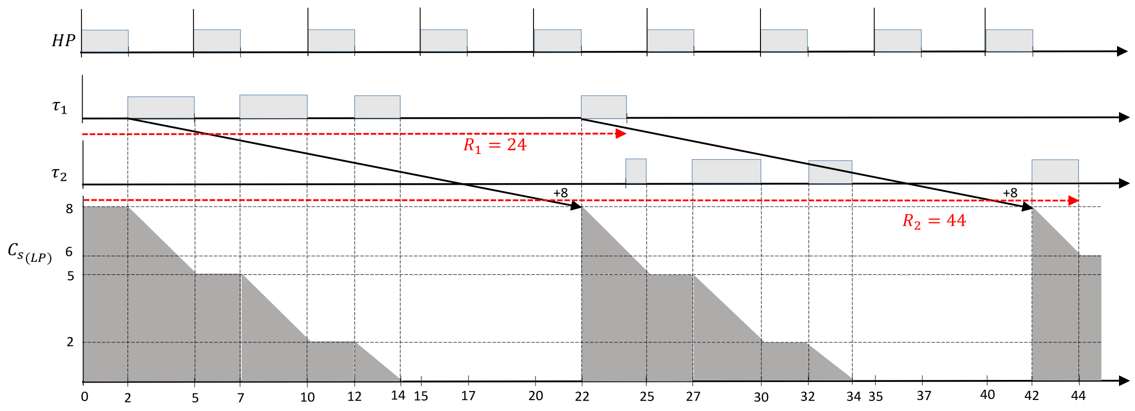


Figure 4.2 Gantt chart of the system described in Table 7.1

4.4 Overview of the Analysis

As mentioned earlier, the proposed analysis is based on a demand-and-supply abstraction that allows computing the response time of jobs released by served and unserved tasks. To provide a general overview of the analysis, consider the system described in Table 4.2, where S_1, the unserved task τ_2, and S_3 have the highest, medium, and lowest priority in the system respectively. Moreover, in the case of served tasks, a task with a smaller index indicates a higher priority.

Thus, in order to compute the response time of the jobs released by a task served by a server, e.g. τ_5, the next steps are to be followed:

1. Model the *demand*, i.e. the cumulative execution requirement, of each task in the server (Definition 4.12). In the example, S_2 serves τ_5, and so the demand of every task served by S_2, i.e. τ_3, τ_4, and τ_5,

Server	C_s	T_s
S_1	1	6
S_2	2	5

Task	C_i	T_i	O_i	*Server*
τ_1	1	6	0	S_1
τ_2	2	5	4	-
τ_3	1	30	2	S_2
τ_4	2	30	5	S_2
τ_5	4	30	10	S_2

Table 4.2 System parameters of the system

has to be modeled (*Task Demand* in Figure 4.3).

2. Model the demand of the server by aggregating the demand of all its constituent tasks (Definition 4.6) and shaping it according to the server's algorithm (Definition 4.17). The outcome, known under the name of *constrained demand curve*, represents the exact windows of time when the server serves its aggregated demand in the absence of higher priority interference (Theorem 6). In our case, the demand of S_2 is given by the aggregated demand of τ_3, τ_4, and τ_5 (*Aggregated Demand of* S_2 in Figure 4.3) and is shaped according to the Sporadic Server algorithm (*Constrained Demand of* S_2 in Figure 4.3).

3. Model the *aggregated higher priority interference demand* of the server by aggregating the demand of its higher priority unserved tasks and the constrained demand of its higher priority servers (Definition 4.18). In our case, this is given by the aggregation of the demand of τ_2, plus the constrained demand of S_1 (*Aggregated Higher Priority Demand of* S_2 in Figure 4.4).

4. Model the available *supply*, i.e. the cumulative execution provision of the server by subtracting its aggregated higher priority demand from the total available supply. The outcome is the *unconstrained execution curve* of the server (Definition 4.19). This calculation is in line with the fact that under fixed priority preemptive scheduling, the server can execute, when no other higher priority unserved task or server is executing (*Unconstrained Execution Curve of* S_2 in Figure 4.4).

5. Given the server's constrained demand and unconstrained execution curve, model its *actual execution curve*, i.e. the exact windows of time when the server serves its aggregated demand taking account of any higher priority interference (Definition 4.20).

6. Compute the response time of the jobs of interest by distributing the windows of time of the server's actual execution curve (*Actual Execution Curve of* S_2 in Figure 4.4) among its constituent tasks on a priority basis (Theorem 11).

In case of unserved tasks, the analysis is similar. Indeed, the available supply of the task, i.e. the unconstrained execution curve, is obtained in the same way as for the servers (Definition 4.19). The actual

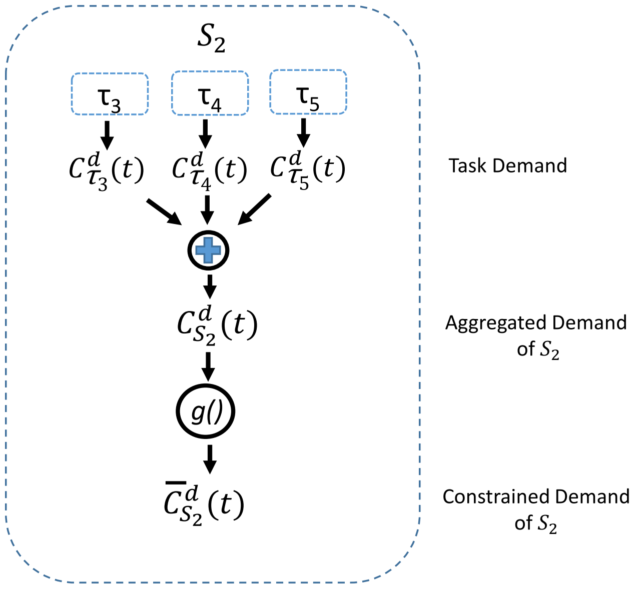

S_2
τ_3
τ_4
τ_5
$C^d_{\tau_3}(t)$ $C^d_{\tau_4}(t)$ $C^d_{\tau_5}(t)$
Task Demand
$C^d_{S_2}(t)$
Aggregated Demand of S_2
g()
$\overline{C}^d_{S_2}(t)$
Constrained Demand of S_2

Figure 4.3 *Constrained Demand Curve* of S_2.

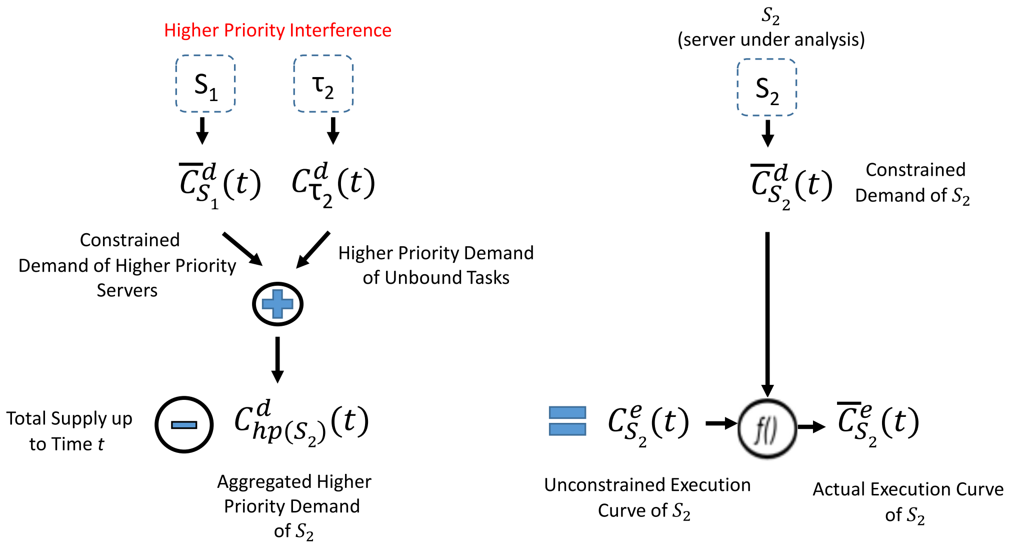

Higher Priority Interference
S_1
τ_2
$\overline{C}^d_{S_1}(t)$ $C^d_{\tau_2}(t)$
Constrained Demand of Higher Priority Servers
Higher Priority Demand of Unbound Tasks
Total Supply up to Time t
$C^d_{hp(S_2)}(t)$
Aggregated Higher Priority Demand of S_2
S_2 (server under analysis)
S_2
$\overline{C}^d_{S_2}(t)$
Constrained Demand of S_2
$C^e_{S_2}(t)$
f()
$\overline{C}^e_{S_2}(t)$
Unconstrained Execution Curve of S_2
Actual Execution Curve of S_2

Figure 4.4 *Actual Execution Curve* of S_2.

execution curve is calculated by deducting (Definition 4.7) the demand from the unconstrained execution curve (Definition 4.23). Eventually, the actual response time is computed as per Theorem 11.

4.5 Window-Curve model

The following section introduces the basic operations and definitions needed to model the building blocks of our analysis, i.e. the demand of tasks and servers as well as their characteristic curves. In particular, the notion of *window* (see Figure 4.5) aims at modeling the execution requirement of a job as well as the execution provision of a server by means of intervals.

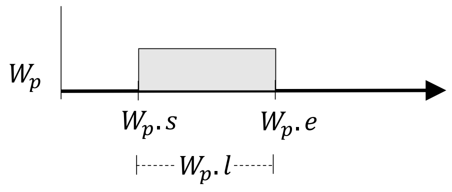


Figure 4.5 A window W_p, its start time $W_p.s$, its end time $W_p.e$, and its length $W_p.l$

Definition 4.1. *Window: A window W_p, denoted by the unit set $\{(W_p.s, W_p.e)\}$, has a start time $W_p.s$ and an end time $W_p.e$ such that $W_p.s < W_p.e$ and its length $W_p.l$ is computed as $W_p.l = W_p.e - W_p.s$. Any window with $W_p.e \leq W_p.s$ is considered empty, i.e. $W_p = \emptyset$.*

The next definitions present the basic operations for windows, which allow us to manipulate them respecting their sequential nature. The *overlap* between two windows tells us if the demand of two jobs overlap so that they are to be merged by means of the *aggregation* operation.

Definition 4.2. *Window Overlap: Given two windows W_p and W_q, the existence of an overlap between them, $W_p \Omega W_q$, is decided as follows:*

$$W_p \Omega W_q = \begin{cases} false & if\ W_p.e < W_q.s\ or\ W_q.e < W_p.s \\ true & otherwise \end{cases}$$

Definition 4.3. *Aggregation of two windows: Given two windows, W_p and W_q, if $W_p \Omega W_q = true$, their aggregation, $W_p \oplus W_q$, results in a new window $\{(x, y)\}$, where $x = \min(W_p.s, W_q.s)$ and $y = \min(W_p.s, W_q.s) + W_p.l + W_q.l$. Otherwise, $W_p \oplus W_q = W_p \cup W_q$.*

The aggregation operation merges any two overlapping or adjacent windows into a new one with length equal to the sum of the lengths of the original windows, as in Example 4.1 (Figure 4.6). In this way any overlapping job demand is preserved.

Example 4.1. *If $W_p = \{(24, 25)\}$ and $W_q = \{(24, 26)\}$, then $W_p \Omega W_q = true$, $x = \min(W_p.s, W_q.s) = \min(24, 24) = 24$, and $y = 24 + W_p.l + W_q.l = 24 + (25 - 24) + (26 - 24) = 27$. Thus $W_p \oplus W_q = \{(24, 27)\}$*

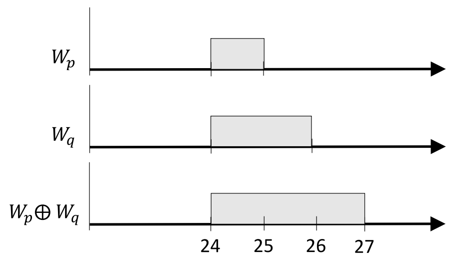


Figure 4.6 $W_p \oplus W_q$ of Example 4.1

The next operation fits, if possible, a demand window W_q into a supply window W_p (See Figure 4.7). A request of W_q can only be serviced by W_p if either there is an overlap or the supply starts later than the demand, i.e. $W_q.s < W_p.e$. Intuitively, if the supply finishes before the demand starts, the demand window cannot be served.

Definition 4.4. *Delta of two windows: Given a supply window W_p and a demand window W_q, their delta, $W_p \ominus W_q$, returns a 3-tuple (R_p, W_o, R_q), where:*

$$W_o = \begin{cases} \emptyset & \textit{if } W_p.e \leq W_q.s \\ \{(W_o.s, W_o.e)\} & \textit{otherwise} \end{cases}$$

Where $W_o.s = \max(W_p.s, W_q.s)$, and $W_o.e = W_o.s + \min(W_q.l, W_p.e - W_o.s)$.

$$R_q = \begin{cases} W_q & \textit{if } W_p.e \leq W_q.s \\ \{(W_q.s + W_o.l, W_q.e)\} & \textit{otherwise} \end{cases}$$

$$R_p = \begin{cases} W_p & \textit{if } W_p.e \leq W_q.s \\ \{(W_p.s, W_o.s)\} \cup \{(W_o.e, W_p.e)\} & \textit{otherwise} \end{cases}$$

Example 4.2. *Given a supply window $W_p = \{(11, 12)\}$ and a demand window $W_q = \{(10, 12)\}$, $W_o.s = \max(11, 10) = 11$, $W_o.e = W_o.s + \min(W_q.l, W_p.e - W_o.s) = 11 + \min(2, 12 - 11) = 12$, $R_q = \{(W_q.s + W_o.l, W_q.e)\} = \{(10 + 1, 12)\} = \{(11, 12)\}$, and $R_p = \{(W_p.s, W_o.s)\} \cup \{(W_o.e, W_p.e)\} = \{(11, 11)\} \cup \{(12, 12)\} = \emptyset$. Thus $W_p \ominus W_q = (\emptyset, \{(11, 12)\}, \{(11, 12)\})$ as shown in Figure 4.7.*

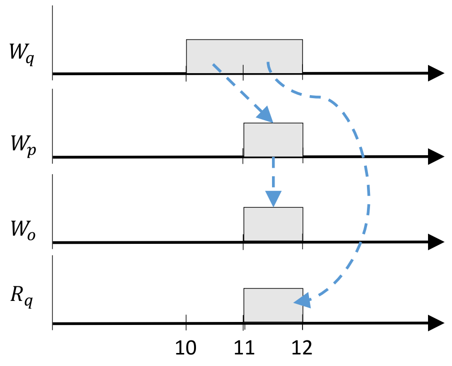


Figure 4.7 $W_p \ominus W_q$ tries to fit the demand W_q into the supply W_p. While W_o and R_q represent the satisfied and unsatisfied demand respectively, R_p denotes the unused supply. In Example 4.2 $R_p = \emptyset$.

Theorem 3. *Given a demand window W_q with start time earlier than the end time of a supply window W_p, the resultant W_o of their delta, $W_p \ominus W_q$, represents the maximum possible satisfied demand executed at the earliest possible time.*

Proof. Since $W_p.e > W_q.s$, according to Definition 4.4, W_o is equal to $\{(W_o.s, W_o.e)\}$, where $W_o.s = \max(W_p.s, W_q.s)$ and $W_o.e = W_o.s + \min(W_q.l, W_p.e - W_o.s)$. Thus, in order to prove the theorem, we first show that the start of W_o coincides with the earliest possible time at which the demand can be executed. Indeed, if the supply starts later than the demand, i.e. $W_p.s > W_q.s$, then the demand will be served as soon as there is any supply available, i.e at $W_p.s$. In this scenario, Definition 4.4 yields $W_o.s = W_p.s$. However, if the demand starts later than, or as the same time as, the supply, i.e. $W_q.s \geq W_p.s$, then the demand is immediately executed at $W_q.s$. In this case, Definition 4.4 dictates that $W_o.s = W_q.s$, which proves the first part of the theorem. To complete the proof, we show that the length of the satisfied demand is equal to that of W_o, i.e. $W_o.l$. From the definition it follows that $W_o.e - W_o.s = \min(W_q.l, W_p.e - W_o.s) = W_o.l$. Hence, if there is enough supply to serve the demand from the time the demand starts to execute, $W_o.s$, i.e. $W_p.e - W_o.s \geq W_q.l$, the whole demand is executed. Under the circumstances, $W_o.l = \min(W_q.l, W_p.e - W_o.s) = W_q.l$. On the other hand, if the supply cannot serve all the demand, i.e. $W_q.l > W_p.e - W_o.s$, then the demand is executed until the end of the supply, i.e $W_p.e$. Correspondingly, Definition 4.4 yields $W_o.l = \min(W_q.l, W_p.e - W_o.s) = W_p.e - W_o.s$.

□

Theorem 4. *Given a demand window W_q with start time earlier than the end time of a supply window W_p, the resultant R_q and R_p of their delta, $W_p \ominus W_q$, denote the unsatisfied demand and the remaining unused supply respectively.*

Proof. According to Theorem 3, W_o is the maximum satisfied demand, and so the unsatisfied demand, if any, must be given by a window whose end time is equal to that of the demand, i.e. $W_q.e$. Moreover, let x be the point in time at which the unsatisfied demand starts, then its length, $W_q.e - x$, is given by subtracting the length of the satisfied demand, $W_o.l$, from that of the demand, $W_q.l$, i.e. $W_q.e - x = W_q.l - W_o.l = W_q.e - W_q.s - W_o.l$. Thus, $x = W_q.s + W_o.l$ and as per Definition 4.4 R_q must be the unsatisfied demand. On the other hand, any satisfied demand W_o, must lie within the limits of W_p, i.e. $W_p.s \leq W_o.s$ and $W_o.e \leq W_p.e$, and so any unused supply lying to the left of $W_o.s$ is given by $\{(W_p.s, W_o.s)\}$. Likewise, any remaining supply lying to the right of $W_o.e$ is computed as $\{(W_o.e, W_p.e)\}$. In agreement with the definition, R_p is given by the union of these two windows, thereby representing the remaining unused supply. $\square$

Observe that as expected Definition 4.4 suggests that the demand should start before the demand ends, i.e. $W_p.e > W_q.s$ so that the demand can be executed at all. The previous example, Example 4.2, depicts a scenario where the supply is exhausted, i.e $R_p = \emptyset$, and so the total demand remains unsatisfied, i.e. $R_q \neq \emptyset$.

The following group of definitions allows us to model the execution demand of tasks and servers, as well as to extend the previous operation to a set of *windows* by the name of *curves*.

Definition 4.5. *Curve: A curve of length t, $\mathcal{C}_p(t)$, is modeled by a set of n nonoverlapping windows $\mathcal{C}_p(t) = \bigcup_{i=1}^{n} W_{p,i}$ s.t. $W_{p,n}.e \leq t$ ordered by their start times and its total capacity, cap$(\mathcal{C}_p(t))$, is the sum of its constituent window lengths, i.e. cap$(\mathcal{C}_p(t)) = \sum_{i=1}^{n} W_{p,i}.l$.*

The next operation enable us to model the demand of a server by *aggregating* the demand of each of its tasks.

Definition 4.6. *Aggregation of two curves: Given two curves $\mathcal{C}_p(t)$ and $\mathcal{C}_q(t)$, their aggregation, $\mathcal{C}_p(t) \oplus \mathcal{C}_q(t)$, results in a new curve $\mathcal{C}_k(t)$, defined by Algorithm 3.*

The algorithm merges all the windows, including duplicates (Line 2), into a new curve that presents no adjacent or overlapping windows (Lines $6-7$).

Example 4.3. *Given $\mathcal{C}_p(t) = \{(0,1),(6,7),(12,13),(18,19),(24,25)\}$ and $\mathcal{C}_q(t) = \{(4,6),(9,11),(14,16),(19,21),(24,26)\}$ with $t = 27$ as shown in Figure 4.8, Algorithm 3 performs the following steps:*

1. *Group all the windows into $\mathcal{C} = \{(0,1),(6,7),(12,13),(18,19),(24,25),(4,6),(9,11),(14,16),(19,21),(24,26)\}$.*

2. *Sort the windows according to their start time $\mathcal{C} = \{(0,1),(4,6),(6,7),(9,11),(12,13),(14,16),(18,19),(19,21),(24,25),(24,26)\}$.*

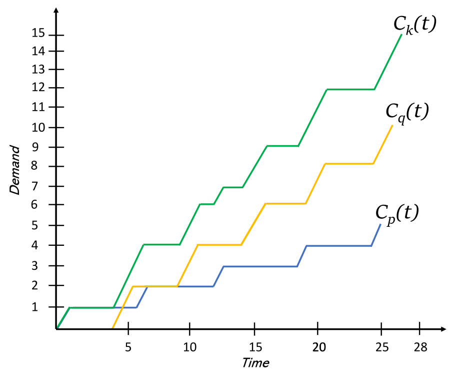


Figure 4.8 Two curves and their aggregation.

Algorithm 3 Aggregation of two curves

```
1:  Input: $\mathcal{C}_p(t), \mathcal{C}_q(t)$
2:  Group all the windows of $\mathcal{C}_p(t)$ and $\mathcal{C}_q(t)$, keeping duplicates, into a new set $\mathcal{C}$.
3:  Sort the windows of the resultant set, $\mathcal{C}$, in ascending order of start time, in order that $\mathcal{C} = \{W_1, W_2, \dots\}$
    s.t. $W_1.s \leq W_2.s \leq \dots$.
4:  $\mathcal{C}_k(t) = \emptyset$
5:  while the number of windows in $\mathcal{C} \neq 1$ do
6:      if $W_1 \Omega W_2 = true$ then
7:          Replace $W_1$ and $W_2$ with $W_1 \oplus W_2$.
8:      else
9:          Remove $W_1$ from $\mathcal{C}$
10:         $\mathcal{C}_k(t) = \mathcal{C}_k(t) \cup W_1$
11: $\mathcal{C}_k(t) = \mathcal{C}_k(t) \cup W_1$
12: Return $\mathcal{C}_k(t)$
```

3. *For the very first pair of windows in $\mathcal{C}$, check if they overlap or are adjacent. If so, replace them with their aggregation. Otherwise, move the first window from $\mathcal{C}$ to $\mathcal{C}_k(t)$. In the example, $\{(0,1)\}$ and $\{(4,6)\}$, do not overlap, and so $\mathcal{C}_k(t) = \{(0,1)\}$ and $\mathcal{C} = \{(4,6),(6,7),(9,11),(12,13),(14,16),(18,19),(19,21),(24,25),(24,26)\}$. The new first pair of windows, $\{(4,6)\}$ and $\{(6,7)\}$, do overlap, and hence they are replaced with $\{(4,6)\} \oplus \{(6,7)\} = \{(4,7)\}$ so that $\mathcal{C} = \{(4,7),(9,11),(12,13),(14,16),(18,19),(19,21),(24,25),(24,26)\}$. Eventually, $\mathcal{C}_k(t) = \{(0,1),(4,7),(9,11),(12,13),(14,16),(18,21),(24,27)\}$.*

Note that, as shown by Figure 4.9, curves can also be represented by a Gantt chart. Moreover, akin to extending the aggregation operation to curves, the delta operation can also be extended so that a demand curve is fit, whenever possible, into a supply curve, as described in the next definition.

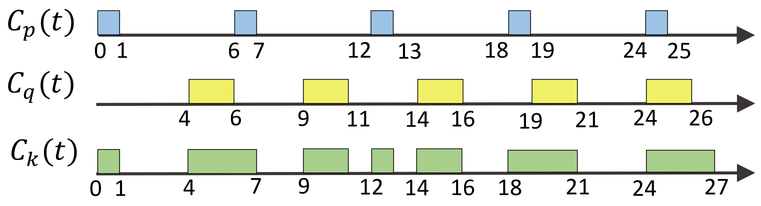


Figure 4.9 Aggregation of $\mathcal{C}_p(t)$ and $\mathcal{C}_q(t)$ in a Gantt chart.

Definition 4.7. *Delta of two curves: Given a supply curve $\mathcal{C}_p(t)$ and a demand curve $\mathcal{C}_q(t)$, their delta, $\mathcal{C}_p(t) \ominus \mathcal{C}_q(t)$, results in a 3-tuple $(\mathcal{C}'_p(t), \mathcal{C}_o(t), \mathcal{C}'_q(t))$ defined by Algorithm 4.*

Algorithm 4 Delta of two curves

```
1: Input: $\mathcal{C}_p(t), \mathcal{C}_q(t)$
2: $\mathcal{C}_o(t) = \mathcal{C}'_p(t) = \mathcal{C}'_q(t) = \emptyset$
3: while $\mathcal{C}_p(t) \neq \emptyset$ and $\mathcal{C}_q(t) \neq \emptyset$ do
4:     if $\exists i$ s.t. $W_{q,1}.s < W_{p,i}.e$ then
5:         Get the smallest index $i^*$ s.t. $W_{q,1}.s < W_{p,i^*}.e$
6:         Compute $(R_p, W_o, R_q) = W_{p,i^*} \ominus W_{q,1}$
7:         $\mathcal{C}_o(t) = \mathcal{C}_o(t) \oplus W_o$
8:         $\mathcal{C}_p(t) = \mathcal{C}_p(t) \setminus W_{p,i^*} \cup R_p$
9:         $\mathcal{C}_q(t) = \mathcal{C}_q(t) \setminus W_{q,1} \cup R_q$
10:    else
11:        Exit while loop
12: $\mathcal{C}'_p(t) = \mathcal{C}_p(t)$
13: $\mathcal{C}'_q(t) = \mathcal{C}_q(t)$
14: Return $\mathcal{C}'_p(t)$, $\mathcal{C}_o(t)$, $\mathcal{C}'_q(t)$
```

As shown by the blue arrows in Figure 4, the algorithm starts by fitting the first demand window into the first supply window that can serve this demand (Lines $4-6$). It then stores the resultant W_o on $\mathcal{C}_o(t)$ (Line 7), and updates the supply (Line 8) and demand (Line 9). These steps are repeated until the demand is completely empty or there is no more supply that can execute the demand.

Example 4.4. *Given a supply curve $\mathcal{C}_p(t) = \{(1,4),(7,9),(11,12),(13,14),(16,18),(21,24)\}$ with $t = 27$ and a demand curve $\mathcal{C}_q(t) = \{(2,3),(5,7),(10,12),(15,17)\}$ as depicted in Figure 4.10, Algorithm 4 performs the following steps:*

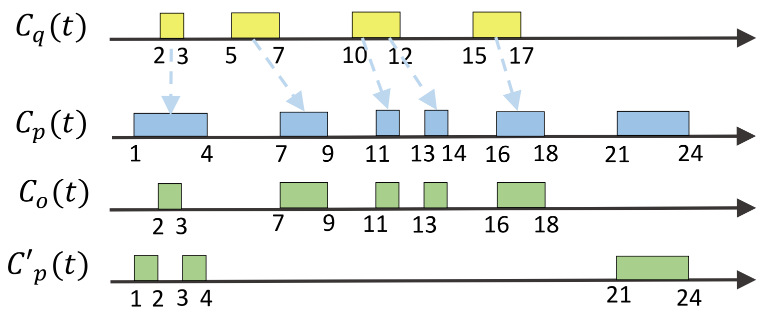


Figure 4.10 Satisfied demand, $\mathcal{C}_o(t)$, and unused supply, $\mathcal{C}'_p(t)$, after performing $\mathcal{C}_p(t) \ominus \mathcal{C}_q(t)$. The unsatisfied demand is empty.

1. *Look for the first window of $\mathcal{C}_p(t)$ that ends after the first window of $\mathcal{C}_q(t)$, $W_{q,1}$, starts, and get their delta. In the example, $\{(1,4)\}$ is the first window to end after $W_{q,1} = \{(2,3)\}$ starts, and so $\{(1,4)\} \ominus \{(2,3)\} = (\{(1,2),(3,4)\}, \{(2,3)\}, \emptyset)$.*

2. *Store the satisfied demand on $\mathcal{C}_o(t)$. Thus, $\mathcal{C}_o(t) = \{(2,3)\}$.*

3. *Update the supply by replacing $(1,4)$ with the unused supply, $(1,2)$ and $(3,4)$. In this way $\mathcal{C}_p(t) = \{(1,2),(3,4),(7,9),(11,12),(13,14),(16,18),(21,24)\}$.*

4. *Update the demand by replacing $W_{q,1}$ with the unsatisfied demand. In our example, as $W_{q,1}$ is completely satisfied, $\mathcal{C}_q(t) = \{(5,7),(10,12),(15,17)\}$.*

5. *Repeat the previous steps until either $\mathcal{C}_p(t)$ or $\mathcal{C}_q(t)$ is empty. Eventually, $\mathcal{C}'_p(t) = \mathcal{C}_p(t) = \{(1,2),(3,4),(21,24)\}$, $\mathcal{C}_o(t) = \{(2,3),(7,9),(11,12),(13,14),(16,18)\}$, and $\mathcal{C}'_q(t) = \mathcal{C}_q(t) = \emptyset$.*

Theorem 5. *Given a supply curve $\mathcal{C}_p(t)$ and a demand curve $\mathcal{C}_q(t)$, their delta, $\mathcal{C}_p(t) \ominus \mathcal{C}_q(t)$, inserts the maximum possible demand into the supply at the earliest possible time and stores it on $\mathcal{C}_o(t)$. The unused supply and unsatisfied demand are saved in $\mathcal{C}'_p(t)$ and $\mathcal{C}'_q(t)$ respectively.*

Proof. It is proven that the theorem holds by showing that the following loop invariant is true before and after each iteration in Algorithm 4. **Loop invariant**: At the start of every iteration of the loop, $\mathcal{C}_o(t)$, $\mathcal{C}_q(t)$ and $\mathcal{C}_p(t)$ contain, up to this point, the maximum satisfied demand executed at the earliest, the current demand and the current supply respectively. **Initialization**: Before the first iteration of the loop, no window of the current demand, $\mathcal{C}_q(t)$, has been fit into the current supply, $\mathcal{C}_p(t)$, and so there is no satisfied demand, i.e. $\mathcal{C}_o(t) = \mathcal{C}'_p(t) = \mathcal{C}'_q(t) = \emptyset$. Then the loop invariant holds true trivially. **Maintenance**: Assume that the loop invariant holds at the start of a given iteration. Let $\mathcal{C}_q(t) = \{W_{q,1}, W_{q,2}, \dots\}$ at this point in time. Let also W_{p,i^*} be the first window in $\mathcal{C}_p(t)$ subject to $W_{p,i^*}.e > W_{q,1}.s$. Then the algorithm computes $W_{p,i^*} \ominus W_{q,1}$ and adds the resultant satisfied demand, W_o, to $\mathcal{C}_o(t)$. According to Theorem 3, the demand window $W_{q,1}$

is maximally served at the earliest, and as a result either $W_{q,1}$ is completely satisfied or satisfied within the limits of W_{p,i^*}. Whereas in the former case, $W_{q,1}$ is completely removed from the demand curve, in the latter case this window is replaced with the remaining unsatisfied demand R_q (Theorem 4). Furthermore, if the delta operation results in some unused supply, R_p (Theorem 4), W_{p,i^*} is replaced with it. Otherwise W_{p,i^*} is removed from the supply. In any case, while the maximum satisfied demand up to this instant is stored on $\mathcal{C}_o(t)$, the curves $\mathcal{C}_q(t)$ and $\mathcal{C}_p(t)$ hold the updated demand and supply. Thus, the loop invariant holds again at the beginning of the next loop. **Termination**: The loop terminates when either there is no more supply, i.e. $\mathcal{C}_p(t) = \emptyset$, or there is no supply window in $\mathcal{C}_p(t)$ that can serve the current demand $\mathcal{C}_q(t)$. In such a case, $\mathcal{C}_o(t)$ contains all the possible demand that could be served at the earliest possible time within the given supply. The loop is also broken if all demand is satisfied, i.e. $\mathcal{C}_q(t) = \emptyset$, in which case $\mathcal{C}_o(t)$ holds all the demand executed at the earliest and $\mathcal{C}_p(t)$ contains the current supply. In all the aforementioned cases, the results at the end of the loop are inline with the loop invariant. From this invariant, we can conclude that after exiting the *while* loop, the current demand $\mathcal{C}_q(t)$ and the current supply $\mathcal{C}_q(t)$ indeed represent the unsatisfied demand and unused supply, which are then passed on to $\mathcal{C}'_q(t)$ and $\mathcal{C}'_p(t)$ respectively. □

Although different kind of servers differ in the strategy to handle their capacity, any server S enforces at most an execution of C_s every T_s. Thus, the following operations, namely the *truncation* (Definition 4.8) and *split* (Definition 4.9), allow a curve to be divided into *sub-curves* delimited by integer multiples of the period of its server, so that these sub-curves can be studied separately in accordance with the budget replenishment algorithm of its server.

Definition 4.8. *Truncation of a curve: Given a curve $\mathcal{C}_p(t)$, truncating the curve to a particular point in time t', denoted by $trunc(\mathcal{C}_p(t), t')$, returns a 2-tuple $(\mathcal{C}^l_p(t), \mathcal{C}^r_p(t))$, where $\mathcal{C}^l_p(t)$ is the curve composed of windows (including any partial window) of $\mathcal{C}_p(t)$ lying to the left of t', and $\mathcal{C}^r_p(t)$ is made up of the remaining windows, as defined by Algorithm 5.*

Example 4.5. *Given $\mathcal{C}_p(t) = \{(1,4),(7,9),(11,12),(13,14),(16,18),(21,24),(27,29),(32,34)\}$ with $t = 34$ as shown in Figure 4.11, in order to calculate $trunc(\mathcal{C}_p(t), t')$, with $t' = 5$, Algorithm 5 performs the next steps:*

1. *Get the first window, W_{p,i^*}, in $\mathcal{C}_p(t)$, whose end time is greater than t'. In the example, the second window, $W_{p,2}$, is the first one to fulfill the condition ($t' = 5 < W_{p,2}.e = 9$).*

2. *Check if the start time of W_{p,i^*} is larger than t'. If so, $\mathcal{C}^l_p(t)$ is composed of all the windows preceding W_{p,i^*}. Otherwise, $\mathcal{C}^l_p(t)$ is made up of all the windows preceding W_{p,i^*} plus a window going from $W_{p,i^*}.s$*

Algorithm 5 Truncation of a curve

```
1:  Input: $\mathcal{C}_p(t), t'$
2:  $\mathcal{C}_p^l(t) = \mathcal{C}_p^r(t) = \emptyset$
3:  if $\exists i$ s.t. $t' < W_{p,i}.e$ then
4:      Get the smallest index $i^*$, satisfying $t' < W_{p,i^*}.e$
5:      if $t' \leq W_{p,i^*}.s$ then
6:          $\mathcal{C}_p^l(t) = \bigcup_{j=1}^{i^*-1} W_{p,j}$
7:          $\mathcal{C}_r^l(t) = \bigcup_{j=i^*}^{n} W_{p,j}$
8:      else
9:          $\mathcal{C}_p^l(t) = \bigcup_{j=1}^{i^*-1} W_{p,j} \cup \{(W_{p,i^*}.s, t')\}$
10:         $\mathcal{C}_p^r(t) = \bigcup_{j=i^*+1}^{n} W_{p,j} \cup \{(t', W_{p,i^*}.e)\}$
11: else
12:     $\mathcal{C}_p^l(t) = \mathcal{C}_p(t)$
13: Return $\mathcal{C}_p^l(t), \mathcal{C}_p^r(t)$
```

to t'. In the example, $t' = 5 < W_{p,2}.s = 7$, and so $\mathcal{C}_p^l(t) = \{(1,4)\}$ and $\mathcal{C}_p^r(t) = \{(7,9),(11,12),(13,14),(16,18),(21,24),(27,29),(32,34)\}$.

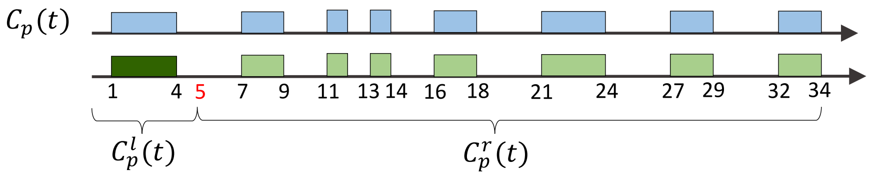


Figure 4.11 *trunc*$(\mathcal{C}_p(t), 5)$ returns $\mathcal{C}_p^l(t)$ and $\mathcal{C}_p^r(t)$ lying to the left and right of $t' = 5$, respectively.

This operation also provides the basis for truncating any demand once the budget of the Polling Server has been decreased to zero, which in turn imitates the behavior of the Polling Server.

Definition 4.9. *Split of a curve: Given a curve $\mathcal{C}_p(t)$ and an interval T, the split operation on the curve, split$(\mathcal{C}_p(t), T)$, produces a set, denoted by $\tilde{\mathcal{C}}_p(t)$, composed of sub-curves so that each and every window in a sub-curve $\tilde{\mathcal{C}}_{p,k}(t)$ lies within $[k \cdot T, (k+1) \cdot T)$, as defined by Algorithm 6.*

The algorithm truncates $\mathcal{C}_p(t)$ to $t' = (k+1) \cdot T$ (Line 5), stores the resultant $\mathcal{C}_p^l(t)$ on $\tilde{\mathcal{C}}_{p,k}(t)$ (Line 6), and updates $\mathcal{C}_p(t)$ by removing $\mathcal{C}_p^l(t)$ from it (Line 7). These steps are repeated for every k in $[0, v]$, where $v = \lceil t/T \rceil - 1$.

Example 4.6. *Given $\mathcal{C}_p(t) = \{(1,4),(7,9),(11,12),(13,14),(16,18),(21,24),(27,29),(32,34)\}$ with $t = 34$ as depicted in Figure 4.12, to calculate split$(\mathcal{C}_p(t), T)$, with $T = 5$, Algorithm 6 takes the next steps:*

1. *Compute $v = \lceil t/T \rceil - 1 = \lceil 34/5 \rceil - 1 = 6$. Then $\mathcal{C}_p(t)$ will be split into $(v+1)$ sub-curves.*

Algorithm 6 Split of a curve

```
1: Input: $\mathcal{C}_p(t), T$
2: $\tilde{\mathcal{C}}_p(t) = \emptyset$
3: Compute $v = \lceil t/T \rceil - 1$
4: for $k \leftarrow 0$ to $v$ do
5:     $\{\mathcal{C}_p^l(t), \mathcal{C}_p^r(t)\} = trunc(\mathcal{C}_p(t), ((k+1)\cdot T))$
6:     $\tilde{\mathcal{C}}_{p,k}(t) = \mathcal{C}_p^l(t)$
7:     $\mathcal{C}_p(t) = \mathcal{C}_p(t) \setminus \mathcal{C}_p^l(t)$
8: $\tilde{\mathcal{C}}_p(t) = \bigcup_{k=0}^{v} \tilde{\mathcal{C}}_{p,k}(t)$
9: Return $\tilde{\mathcal{C}}_p(t)$
```

2. *For each sub-curve $\tilde{\mathcal{C}}_{p,k}(t)$, get $\mathcal{C}_p^l(t)$ by truncating $\mathcal{C}_p(t)$ to $(k+1)\cdot T$, so that $\tilde{\mathcal{C}}_{p,k}(t)$ is given by $\mathcal{C}_p^l(t)$, and then remove $\mathcal{C}_p^l(t)$ from $\mathcal{C}_p(t)$. In the example truncating $\mathcal{C}_p(t)$ to 5 (Example 4.5) gives us $\tilde{\mathcal{C}}_{p,0}(t) = \{(1,4)\}$, thus $\mathcal{C}_p(t) = \{(7,9),(11,12),(13,14),(16,18),(21,24),(27,29),(32,34)\}$. Truncating $\mathcal{C}_p(t)$ by 10 and updating the curve leads to $\tilde{\mathcal{C}}_{p,1}(t) = \{(7,9)\}$ and $\mathcal{C}_p(t) = \{(11,12),(13,14),(16,18),(21,24),(27,29),(32,34)\}$. Eventually, $\tilde{\mathcal{C}}_{p,2}(t) = \{(11,12),(13,14)\}$, $\tilde{\mathcal{C}}_{p,3}(t) = \{(16,18)\}$, $\tilde{\mathcal{C}}_{p,4}(t) = \{(21,24)\}$, $\tilde{\mathcal{C}}_{p,5}(t) = \{(27,29)\}$, and $\tilde{\mathcal{C}}_{p,6}(t) = \{(32,34)\}$.*

3. *Finally $\tilde{\mathcal{C}}_p(t) = \cup_{k=0}^{6}\tilde{\mathcal{C}}_{p,k}(t)$.*

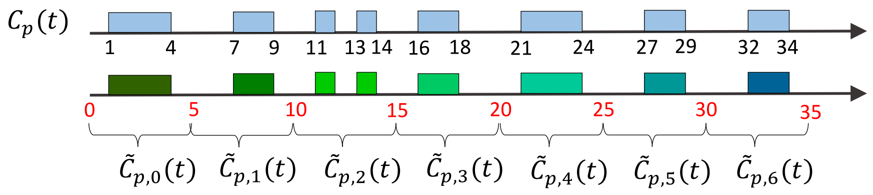


Figure 4.12 $split(\mathcal{C}_p(t), 5)$ splits $\mathcal{C}_p(t)$ into intervals of 5.

The following operation lets us transform any unsatisfied demand into a window shifted to a point in time t'. Moreover, if a curve is composed of one window, $trans(\mathcal{C}_p(t), t')$ shifts the window to t' as shown in Example 4.8.

Definition 4.10. *Transformation of a curve into a window: Given a curve $\mathcal{C}_p(t)$, transforming the curve into a window, denoted by $trans(\mathcal{C}_p(t), t')$, returns a window $\{(x,y)\}$ with start time $x = t'$ and length equal to the total capacity of the curve, i.e. $y = t' + cap(\mathcal{C}_p(t))$.*

Example 4.7. *Given $C_p(t) = \{(2,3),(5,7),(10,12)\}$ with $t = 27$ as presented in Figure 4.13, the operation $trans(\mathcal{C}_p(t), t')$, with $t' = 15$, transforms the curve into a window with start time $x = 15$ and end time $y = t' + cap(\mathcal{C}_p(t)) = 15 + (3-2) + (7-5) + (12-10) = 20$, i.e. $trans(\mathcal{C}_p(t), 15) = \{(15, 20)\}$.*

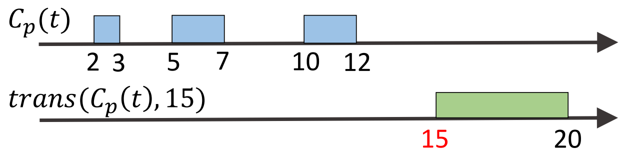


Figure 4.13 Transformation of a curve into a window.

Example 4.8. *Given $C_p(t) = \{(12, 14)\}$ with $t = 27$, and $t' = 15$, $trans(\mathcal{C}_p(t), 15) = \{(15, 17)\}$.*

Unlike, the Polling Server, the Deferrable Server preserves its capacity if no request are pending upon its invocation. This capacity is maintained until the end of the period, so that future jobs can be served within the limit of its budget. Hence, the *constraint* operation is introduced in order to place budget constraints on a curve, serving as a basis for reproducing the behavior of the Deferrable Server, and for calculating the *actual execution curve* of a server.

Definition 4.11. *Constraint on a curve: Given a curve $\mathcal{C}_p(t)$, a budget constraint B, and a point in time t', the constraint operation on the curve, $constr(\mathcal{C}_p(t), B, t')$, results in a 2-tuple $(\mathcal{C}_p^c(t), W^r)$ defined by Algorithm 7. $\mathcal{C}_p^c(t)$ is a curve whose total capacity is B, provided that the total capacity of the original curve $\mathcal{C}_p(t)$ allows it, and W^r is a window starting at t' and whose length is equal to $cap(\mathcal{C}_p(t)) - B$.*

Algorithm 7 Constraint on a curve

```
1: Input: $\mathcal{C}_p(t), B, t'$
2: $\mathcal{C}_p^c(t) = W^r = \emptyset$
3: if $\exists i$ s.t. $\sum_{j=1}^{i} W_{p,j}.l > B$ then
4:     Get the smallest index $i^*$ s.t. $\sum_{j=1}^{i^*} W_{p,j}.l > B$
5:     if $i^* = 1$ then
6:         $\mathcal{C}_p^c(t) = \{(W_{p,i^*}.s, W_{p,i^*}.s + B)\}$
7:     else
8:         $b = B - \sum_{j=1}^{k-1} W_{p,j}.l$
9:         $\mathcal{C}_p^c(t) = \bigcup_{j=1}^{i^*-1} W_{p,j} \cup \{(W_{p,i^*}.s, W_{p,i^*}.s + b)\}$
10:    $W^r = \{(t', t' + cap(\mathcal{C}_p(t)) - B)\}$
11: else
12:    $\mathcal{C}_p^c(t) = \mathcal{C}_p(t)$
13: Return $\mathcal{C}_p^c(t), W^r$
```

The algorithm dictates to obtain the first window of $\mathcal{C}_p(t)$, W_{p,i^*}, whose length added to those of its predecessors yields a value larger than B (Line 4). It then groups all the preceding windows into a new curve $\mathcal{C}_p^c(t)$, adding that portion of W_{p,i^*} needed so that $cap(\mathcal{C}_p^c(t)) = B$ (Lines 5-9). Finally, the remainder of $\mathcal{C}_p(t)$ is transformed into a window starting at time t' (Line 10).

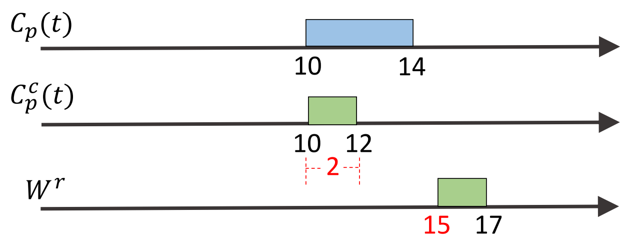


Figure 4.14 Placing a budget constraint $constr(\mathcal{C}_p(t), B = 2, t' = 15)$ on $\mathcal{C}_p(t)$ yields a curve $\mathcal{C}_p^c(t)$ with capacity 2 and a window W^r starting at $t' = 15$ with length equal to $cap(\mathcal{C}_p^c(t)) - B = 2$.

Example 4.9. *Given $\mathcal{C}_p(t) = \{(10, 14)\}$ with $t = 27$ as illustrated in Figure 4.14, $B = 2$, and $t' = 15$, in order to calculate $constr(\mathcal{C}_p(t), 2, 15)$ Algorithm 7 follows the next steps:*

1. *Find the first window, W_{p,i^*}, whose length plus those of the preceding windows result in a value larger than B. In the example, the first and only window of $\mathcal{C}_p(t)$, $W_{p,1}$, fulfills this condition, i.e. $cap(\mathcal{C}_p(t)) = W_{p,1}.l = 4 > B = 2$.*

2. *If $i^* = 1$, then $\mathcal{C}_p^c(t)$ is given by a window with start time $W_{p,i^*}.s$ and length B. Otherwise, $\mathcal{C}_p^c(t)$ is composed of the windows preceding W_{p,i^*} plus a window with start time $W_{p,i^*}.s$ and length b, representing the portion of W_{p,i^*} needed so that the total capacity of the resultant curve equals B. In the example, $i^* = 1$, and so $\mathcal{C}_p^c(t) = \{W_{p,1}.s, W_{p,1}.s + B\} = \{(10, 10 + 2)\} = \{(10, 12)\}$.*

3. *Finally, compute W^r as $\{(t', t' + cap(\mathcal{C}_p(t)) - B)\}$, i.e. $W^r = \{15, 15 + 4 - 2\} = \{15, 17\}$. Thus, $constr(\mathcal{C}_p(t), 2, 5) = (\{(10, 12)\}, \{(15, 17)\})$ as depicted in Figure 4.14.*

4.6 Demand of tasks and servers

The next section models the demand of tasks and servers laying the foundations for the response time analysis provided in this work.

Definition 4.12. *Demand of a task: The demand of a task τ_i can be modelled by a curve, $\mathcal{C}_{\tau_i}^d(t)$, representing the cumulative execution demanded by all jobs that have arrived in the time interval $[0, t)$.*

Example 4.10. *Given task τ_2 from Table 4.2, its demand up to time $t = 27$ can be modelled by $\mathcal{C}_{\tau_2}^d(t) =$* $\{(4, 6), (9, 11), (14, 16), (19, 21), (24, 26)\}$.

Definition 4.13. *Aggregated demand curve of a server: Given a server S scheduling n tasks, its aggregated demand curve, $\mathcal{C}_s^d(t)$, is given by the aggregation of the demand of its tasks, i.e. $\mathcal{C}_s^d(t) = \mathcal{C}_{\tau_1}^d(t) \oplus ... \oplus \mathcal{C}_{\tau_n}^d(t)$.*

Example 4.11. *Given the server S_2 from Table 4.2 scheduling tasks τ_3, τ_4, and τ_5, its demand up to time $t = 27$ can be modelled by $\mathcal{C}^d_{s_2}(t) = \mathcal{C}^d_{\tau_3}(t) \oplus \mathcal{C}^d_{\tau_4}(t) \oplus \mathcal{C}^d_{\tau_5}(t) = \{(2,3)\} \oplus \{(5,7)\} \oplus \{(10,14)\} = \{(2,3),(5,7),(10,14)\}$ as exposed in Figure 4.15.*

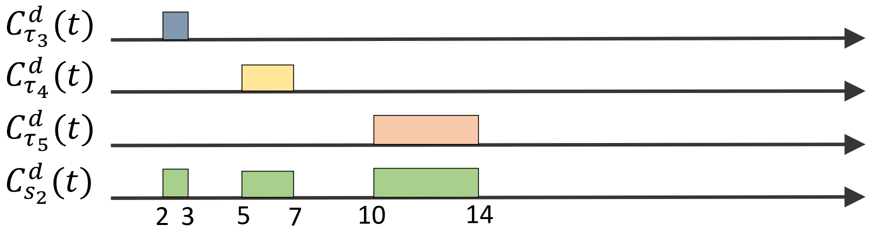

$\mathcal{C}^d_{\tau_3}(t)$
$\mathcal{C}^d_{\tau_4}(t)$
$\mathcal{C}^d_{\tau_5}(t)$
$\mathcal{C}^d_{s_2}(t)$
2
3
5
7
10
14

Figure 4.15 Demand of the tasks served by S_2 (See Table 4.2) and aggregated demand of the latter, $\mathcal{C}^d_{s_2}(t)$.

Definition 4.14. *Constrained demand curve of a Polling Server: Given a Polling Server S with aggregated demand curve $\mathcal{C}^d_s(t)$, period T_s, and budget C_s, its constrained demand curve $\overline{\mathcal{C}}^d_s(t)$, is computed as shown in Algorithm 8.*

Example 4.12. *Given a Polling Server with $C_s = 2$, $T_s = 5$, and aggregated demand $C^d_s(t) = \{(1,2),(5,7),(10,14)\}$ with $t = 27$, in order to compute its constrained demand curve $\overline{\mathcal{C}}^d_s(t)$, Algorithm 8 performs the following steps:*

1. *Split $C^d_s(t)$ by T_s as per Algorithm 6 so that $\tilde{\mathcal{C}}^d_s(t) = \cup^v_{k=0}\tilde{\mathcal{C}}^d_{s,k}(t)$. In the example, the algorithm yields $v = \lceil t/T_s \rceil - 1 = \lceil 27/5 \rceil - 1 = 5$, $\tilde{\mathcal{C}}^d_{s,0}(t) = \{(1,2)\}$, $\tilde{\mathcal{C}}^d_{s,1}(t) = \{(5,7)\}$, $\tilde{\mathcal{C}}^d_{s,2}(t) = \{(10,14)\}$, and $\tilde{\mathcal{C}}^d_{s,3}(t) = \tilde{\mathcal{C}}^d_{s,4}(t) = \tilde{\mathcal{C}}^d_{s,5}(t) = \emptyset$. Hence $\tilde{\mathcal{C}}^d_s(t) = \cup^5_{k=0}\tilde{\mathcal{C}}^d_{s,k}(t)$.*

2. *For each sub-curve $\tilde{\mathcal{C}}^d_{s,k}(t)$, check if its first window, $W^1_{s,k}$, has a start time different from $k \cdot T_s$. If so, $\overline{\mathcal{C}}^d_{s,k}(t) = \emptyset$. Otherwise, get $\overline{\mathcal{C}}^d_{s,k}(t)$ by truncating $W^1_{s,k}$ to $k \cdot T_s + C_s$, so that $\overline{\mathcal{C}}^d_{s,k}(t)$ is given by $\mathcal{C}^l_s(t)$, and update $\tilde{\mathcal{C}}^d_{s,k}(t)$ by removing its first window and adding any remainder to the sub-curve. Next, transform $\tilde{\mathcal{C}}^d_{s,k}(t)$ into a window, W^r, starting at time $(k+1) \cdot T_s$ and add it to the next sub-curve $\tilde{\mathcal{C}}^d_{s,k+1}(t)$. In the example, $W^1_{s,0}.s = 1$ is different from $0 \cdot T_s = 0$, and so $\overline{\mathcal{C}}^d_{s,0}(t) = \emptyset$. Next, $\tilde{\mathcal{C}}^d_{s,0}(t)$ is transformed into W^r by computing $trans(\tilde{\mathcal{C}}^d_{s,0}(t), T_s) = trans(\{(1,2)\}, 5) = \{(5,6)\}$ and added to $\tilde{\mathcal{C}}^d_{s,1}(t)$, i.e. $\tilde{\mathcal{C}}^d_{s,1}(t) = \{(5,7)\} \oplus \{(5,6)\} = \{(5,8)\}$. In the second iteration, $W^1_{s,1}.s = 5$ is equal to $T_s = 5$, and so $W^1_{s,1}$ is truncated to $T_s + C_s = 5 + 2 = 7$, i.e. $(\mathcal{C}^l_s(t), \mathcal{C}^r_s(t)) = trunc(W^1_{s,1}, 7) = (\{(5,7)\}, \{(7,8)\})$. Hence $\overline{\mathcal{C}}^d_{s,1}(t) = \mathcal{C}^l_s(t) = \{(5,7)\}$ and $\tilde{\mathcal{C}}^d_{s,1}(t)$ is updated to $\tilde{\mathcal{C}}^d_{s,1}(t) \setminus W^1_{s,1} \cup \mathcal{C}^r_s(t) = \{(7,8)\}$, transformed into a window starting at time $2 \cdot T_s$, i.e. $trans(\tilde{\mathcal{C}}^d_{s,1}(t), 2 \cdot T_s) = trans(\{(7,8)\}, 10) = \{(10,11)\}$, and added to $\tilde{\mathcal{C}}^d_{s,2}(t)$, so that $\tilde{\mathcal{C}}^d_{s,2}(t) = \{(10,11)\} \oplus \{(10,14)\} = \{(10,15)\}$. Eventually $\overline{\mathcal{C}}^d_{s,2}(t) = \{(10,12)\}$, $\overline{\mathcal{C}}^d_{s,3}(t) = \{(15,17)\}$, $\overline{\mathcal{C}}^d_{s,4}(t) = \{(20,21)\}$, and $\overline{\mathcal{C}}^d_{s,5}(t) = \emptyset$.*

3. Finally $\overline{\mathcal{C}}^d_s(t) = \cup_{k=0}^{5} \overline{\mathcal{C}}^d_{s,k}(t) = \{(5,7),(10,12),(15,17),(20,21)\}$.

Algorithm 8 constrained demand curve of a Polling Server

```
1: Input: $\mathcal{C}^d_s(t)$, $C_s$, $T_s$
2: Compute $\tilde{\mathcal{C}}^d_s(t) = \bigcup_{k=0}^{v} \tilde{\mathcal{C}}^d_{s,k}(t) = split(\mathcal{C}^d_s(t), T_s)$ as per Algorithm 6.
3: Let $W^1_{s,k}$ represent the first window of $\tilde{\mathcal{C}}^d_{s,k}(t)$.
4: for $k \leftarrow 0$ to $v$ do
5:     if $W^1_{s,k}.s = k \cdot T_s$ then
6:         Get $(\mathcal{C}^l_s(t), \mathcal{C}^r_s(t)) = trunc(W^1_{s,k}, k \cdot T_s + C_s)$ as shown by Algorithm 5.
7:         $\overline{\mathcal{C}}^d_{s,k}(t) = \mathcal{C}^l_s(t)$
8:         $\tilde{\mathcal{C}}^d_{s,k}(t) = \tilde{\mathcal{C}}^d_{s,k}(t) \setminus W^1_{s,k} \cup \mathcal{C}^r_s(t)$
9:     else
10:        $\overline{\mathcal{C}}^d_{s,k}(t) = \emptyset$
11:    if $k < v$ then
12:        Calculate $W^r = trans(\tilde{\mathcal{C}}^d_{s,k}(t), (k+1) \cdot T_s)$ according to Definition 4.10.
13:        Obtain $\tilde{\mathcal{C}}^d_{s,k+1}(t) = \tilde{\mathcal{C}}^d_{s,k+1}(t) \oplus W^r$ just as Algorithm 3.
14: $\overline{\mathcal{C}}^d_s(t) = \bigcup_{k=0}^{v} \overline{\mathcal{C}}^d_{s,k}(t)$
15: Return $\overline{\mathcal{C}}^d_s(t)$
```

Definition 4.15. *Constrained demand curve of an extended Polling Server: Given an extended Polling Server S with aggregated demand curve* $\mathcal{C}^d_s(t)$*, period* T_s*, and budget* C_s*, its constrained demand curve* $\overline{\mathcal{C}}^d_s(t)$*, is computed as per Algorithm 9.*

Example 4.13. *Given a Polling Server with* $C_s = 2$*,* $T_s = 5$*, and aggregated demand* $C^d_s(t) = \{(2,3),(5,7),(10,14)\}$ *with* $t = 27$*, in order to compute its constrained demand curve* $\overline{\mathcal{C}}^d_s(t)$*, Algorithm 9 performs the following steps:*

1. *Split* $C^d_s(t)$ *by* T_s *as per Algorithm 6 so that* $\tilde{\mathcal{C}}^d_s(t) = \cup_{k=0}^{v} \tilde{\mathcal{C}}^d_{s,k}(t)$*. In the example, the algorithm yields* $v = \lceil t/T_s \rceil - 1 = \lceil 27/5 \rceil - 1 = 5$*,* $\tilde{\mathcal{C}}^d_{s,0}(t) = \{(2,3)\}$*,* $\tilde{\mathcal{C}}^d_{s,1}(t) = \{(5,7)\}$*,* $\tilde{\mathcal{C}}^d_{s,2}(t) = \{(10,14)\}$*, and* $\tilde{\mathcal{C}}^d_{s,3}(t) = \tilde{\mathcal{C}}^d_{s,4}(t) = \tilde{\mathcal{C}}^d_{s,5}(t) = \emptyset$*. Hence* $\tilde{\mathcal{C}}^d_s(t) = \cup_{k=0}^{5} \tilde{\mathcal{C}}^d_{s,k}(t) = \cup_{k=0}^{3} \tilde{\mathcal{C}}_{p,k}(t)$*.*

2. *For each sub-curve* $\tilde{\mathcal{C}}^d_{s,k}(t)$*, get* $\overline{\mathcal{C}}^d_{s,k}(t)$ *by truncating* $\tilde{\mathcal{C}}^d_{s,k}(t)$ *to* $k \cdot T_s + C_s$*, so that* $\overline{\mathcal{C}}^d_{s,k}(t)$ *is given by* $\mathcal{C}^l_s(t)$*. The remainder,* $\mathcal{C}^r_s(t)$*, is transformed into a new window starting at* $(k+1) \cdot T_s$ *and added to* $\tilde{\mathcal{C}}^d_{s,k+1}(t)$*. In the example,* $trunc(\tilde{\mathcal{C}}^d_{s,0}(t), 2) = (\emptyset, \{(2,3)\})$*, and so* $\overline{\mathcal{C}}^d_{s,0}(t) = \emptyset$*. The remainder* $\mathcal{C}^r_s(t) = \{(2,3)\}$ *is transformed into a new window* $W^r = trans(\mathcal{C}^r_s(t), 5) = \{(5,6)\}$ *and added to the next sub-curve, so that* $\tilde{\mathcal{C}}^d_{s,1}(t) = \tilde{\mathcal{C}}^d_{s,1}(t) \oplus \{(5,7)\} = \{(5,8)\}$*. Eventually* $\overline{\mathcal{C}}^d_{s,1}(t) = \{(5,7)\}$*,* $\overline{\mathcal{C}}^d_{s,2}(t) = \{(10,12)\}$*,* $\overline{\mathcal{C}}^d_{s,3}(t) = \{(15,17)\}$*,* $\overline{\mathcal{C}}^d_{s,4}(t) = \{(20,21)\}$*, and* $\overline{\mathcal{C}}^d_{s,5}(t) = \emptyset$*.*

3. *Finally* $\overline{\mathcal{C}}^d_s(t) = \cup_{k=0}^{5} \overline{\mathcal{C}}^d_{s,k}(t) = \{(5,7),(10,12),(15,17),(20,21)\}$*.*

Example 4.14. *Given an extended Polling Server with $C_s = 2$, $T_s = 5$, and aggregated demand $\mathcal{C}^d_s(t) = \{(1,2),(5,7),(10,14)\}$ with $t = 27$, Algorithm 9 yields $\overline{\mathcal{C}}^d_s(t) = \cup_{k=0}^{5}\overline{\mathcal{C}}^d_{s,k}(t)$, where $\overline{\mathcal{C}}^d_{s,0}(t) = \{(1,2)\}$, $\overline{\mathcal{C}}^d_{s,1}(t) = \{(5,7)\}$, $\overline{\mathcal{C}}^d_{s,2}(t) = \{(10,12)\}$, $\overline{\mathcal{C}}^d_{s,3}(t) = \{(15,17)\}$, and $\overline{\mathcal{C}}^d_{s,4}(t) = \overline{\mathcal{C}}^d_{s,5}(t) = \emptyset$.*

Algorithm 9 Constrained demand curve of an extended Polling Server

```
1:  Input: $\mathcal{C}^d_s(t)$, $C_s$, $T_s$
2:  Compute $\tilde{\mathcal{C}}^d_s(t) = \bigcup_{k=0}^{v} \tilde{\mathcal{C}}^d_{s,k}(t) = split(\mathcal{C}^d_s(t), T_s)$ as per Algorithm 6.
3:  for $k \leftarrow 0$ to $v$ do
4:      Get $(\mathcal{C}^l_s(t), \mathcal{C}^r_s(t)) = trunc(\tilde{\mathcal{C}}^d_{s,k}(t), k \cdot T_s + C_s)$ as shown by Algorithm 5.
5:      $\overline{\mathcal{C}}^d_{s,k}(t) = \mathcal{C}^l_s(t)$
6:      if $k < v$ then
7:          Calculate $W^r = trans(\mathcal{C}^r_s(t), (k+1) \cdot T_s)$ according to Definition 4.10.
8:          Obtain $\tilde{\mathcal{C}}^d_{s,k+1}(t) = \tilde{\mathcal{C}}^d_{s,k+1}(t) \oplus W^r$ just as Algorithm 3.
9:  $\overline{\mathcal{C}}^d_s(t) = \bigcup_{k=0}^{v} \overline{\mathcal{C}}^d_{s,k}(t)$
10: Return $\overline{\mathcal{C}}^d_s(t)$
```

Figure 4.16 (bottom) shows how the aggregated demand of the server from Example 4.13, $\mathcal{C}^d_s(t)$, is truncated to C_s every period T_s, producing its constrained demand curve $\overline{\mathcal{C}}^d_s(t)$. Green arrows show, how the unsatisfied demand is shifted to the next period.

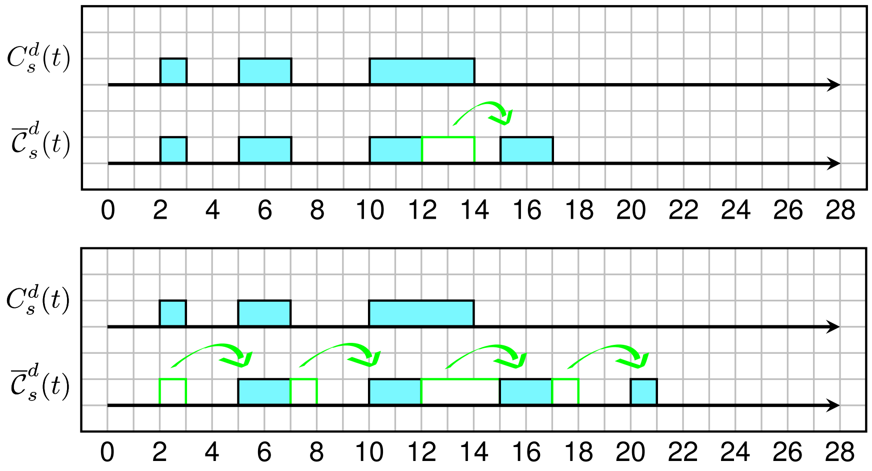


Figure 4.16 Constrained demand curve of a Deferrable Server (top) extended Polling Periodic Server (bottom)

Definition 4.16. *Constrained demand curve of a Deferrable Server: Given a Deferrable Server S with aggregated demand curve $\mathcal{C}^d_s(t)$, period T_s, and budget C_s, its constrained demand curve $\overline{\mathcal{C}}^d_s(t)$ is given by Algorithm 10.*

Algorithm 10 Constrained demand curve of a Deferrable Server

```
1: Input: $\mathcal{C}_s^d(t)$, $C_s$, $T_s$
2: Compute $\tilde{\mathcal{C}}_s^d(t) = \bigcup_{k=0}^{v} \tilde{\mathcal{C}}_{s,k}^d(t) = split(\mathcal{C}_s^d(t), T_s)$ as per Algorithm 6.
3: for $k \leftarrow 0$ to $v$ do
4:     Obtain $(\mathcal{C}_s^c(t), W^r) = constr(\tilde{\mathcal{C}}_{s,k}^d(t), C_s, (k+1)\cdot T_s)$ according to Algorithm 7.
5:     $\overline{\mathcal{C}}_{s,k}^d(t) = \mathcal{C}_s^c(t)$
6:     if $k < v$ then
7:         Get $\tilde{\mathcal{C}}_{s,k+1}^d(t) = \tilde{\mathcal{C}}_{s,k+1}^d(t) \oplus W^r$ just as Algorithm 3.
8: $\overline{\mathcal{C}}_s^d(t) = \bigcup_{k=0}^{v} \overline{\mathcal{C}}_{s,k}^d(t)$
9: Return $\overline{\mathcal{C}}_s^d(t)$
```

Example 4.15. *Given a Deferrable Server with $C_s = 2$, $T_s = 5$, and aggregated demand $C_s^d(t) = \{(2,3),(5,7),(10,14)\}$ with $t = 27$, in order to compute its constrained demand curve $\overline{\mathcal{C}}_s^d(t)$, Algorithm 10 performs the following steps:*

1. *Split $C_s^d(t)$ by T_s as per Algorithm 6 so that $\tilde{\mathcal{C}}_s^d(t) = \cup_{k=0}^{v}\tilde{\mathcal{C}}_{s,k}^d(t)$. In the example, the algorithm yields $v = \lceil t/T_s \rceil - 1 = \lceil 27/5 \rceil - 1 = 5$, $\tilde{\mathcal{C}}_{s,0}^d(t) = \{(2,3)\}$, $\tilde{\mathcal{C}}_{s,1}^d(t) = \{(5,7)\}$, $\tilde{\mathcal{C}}_{s,2}^d(t) = \{(10,14)\}$, and $\tilde{\mathcal{C}}_{s,3}^d(t) = \tilde{\mathcal{C}}_{s,4}^d(t) = \tilde{\mathcal{C}}_{s,5}^d(t) = \emptyset$. Hence $\tilde{\mathcal{C}}_s^d(t) = \cup_{k=0}^{5}\tilde{\mathcal{C}}_{s,k}^d(t) = \cup_{k=0}^{3}\tilde{\mathcal{C}}_{p,k}(t)$.*

2. *For each sub-curve $\tilde{\mathcal{C}}_{s,k}^d(t)$, get $\overline{\mathcal{C}}_{s,k}^d(t)$ by constraining $\tilde{\mathcal{C}}_{s,k}^d(t)$ by $(C_s, k\cdot T_s + C_s)$, so that $\overline{\mathcal{C}}_{s,k}^d(t)$ is given by $\mathcal{C}_s^c(t)$, and then add W^r to the next sub-curve $\tilde{\mathcal{C}}_{s,k+1}^d(t)$. In the example, $constr(\tilde{\mathcal{C}}_{s,0}^d(t), 2, 5) = (\{(2,3)\}, \emptyset)$, and so $\overline{\mathcal{C}}_{s,0}^d(t) = \{(2,3)\}$ and $\tilde{\mathcal{C}}_{s,1}^d(t) = \tilde{\mathcal{C}}_{s,1}^d(t) \cup \emptyset = \{(5,7)\}$. Next, $constr(\tilde{\mathcal{C}}_{s,1}^d(t), 2, 10) = (\{(5,7)\}, \emptyset)$, which in turn gives us $\overline{\mathcal{C}}_{s,1}^d(t) = \{(5,7)\}$ and $\tilde{\mathcal{C}}_{s,2}^d(t) = \tilde{\mathcal{C}}_{s,2}^d(t) \cup \emptyset = \{(10,14)\}$. Eventually, $\overline{\mathcal{C}}_{s,2}^d(t) = \{(10,12)\}$, $\overline{\mathcal{C}}_{s,3}^d(t) = \{(15,17)\}$, and $\overline{\mathcal{C}}_{s,4}^d(t) = \overline{\mathcal{C}}_{s,5}^d(t) = \emptyset$.*

3. *Finally $\overline{\mathcal{C}}_s^d(t) = \cup_{k=0}^{5}\overline{\mathcal{C}}_{s,k}^d(t) = \{(2,3),(5,7),(10,12),(15,17)\}$.*

Figure 4.16 (top) shows how the aggregated demand of the server from Example 4.15, $C_s^d(t)$, is constrained conforming to the Deferrable Server's budget constrain, producing its constrained demand curve $\overline{\mathcal{C}}_s^d(t)$.

Definition 4.17. *Constrained demand curve of a Sporadic Server: Given a Sporadic Server S with aggregated demand curve $\mathcal{C}_s^d(t)$, period T_s, and budget C_s, its constrained demand curve $\overline{\mathcal{C}}_s^d(t)$, is computed as dictated by Algorithm 11.*

As the budget of a Sporadic Server starts being consumed only when the very first request arrives, the algorithm begins by initializing $\mathcal{C}_p(t)$ to a supply window with start time equal to that of $W_{q,1}$, and length equal to the server's capacity (Line 2). Observe that the length of this new window is equal to C_s since the server can only serve its requests within the limits of its budget.

Algorithm 11 Constrained Demand of a Sporadic Server

```
1: Input: $\mathcal{C}_s^d(t) = \{W_{q,1}, W_{q,2}, \dots\}$, $C_s$, $T_s$
2: Let $\mathcal{C}_p(t) = \{(W_{q,1}.s, W_{q,1}.s + C_s)\}$.
3: $\overline{\mathcal{C}}_s^d(t) = \emptyset$.
4: while $\mathcal{C}_s^d(t) \neq \emptyset$ do
5:     Get the first window, $W_{p,1}$, of $\mathcal{C}_p$
6:     $(R_p, W_o, R_q) = W_{p,1} \ominus W_{q,1}$
7:     $\overline{\mathcal{C}}_s^d(t) = \overline{\mathcal{C}}_s^d(t) \oplus W_o$
8:     $\mathcal{C}_s^d(t) = \mathcal{C}_s^d(t) \setminus W_{q,1} \cup R_q$
9:     Get the first window of the updated $\mathcal{C}_s^d(t)$, $W'_{q,1}$
10:    Compute $\gamma = \{(W_o.s + T_s, W_o.e + T_s)\}$
11:    if $R_p \neq \emptyset$ then
12:        if $(R_p.s < W'_{q,1}.s)$ then
13:            $R_p = \{(W'_{q,1}.s, W'_{q,1}.s + R_p.l)\}$
14:    if $(\gamma.s < W'_{q,1}.s)$ then
15:        $\gamma = \{(W'_{q,1}.s, W'_{q,1}.s + \gamma.l)\}$
16:    $\mathcal{C}_p(t) = R_p \oplus \gamma$
17: Return $\overline{\mathcal{C}}_s^d(t)$
```

Algorithm 11 then fits $W_{q,1}$ into the first window of $\mathcal{C}_p(t)$, $W_{p,1}$, by means of the delta operation (Line 6), and adds the resultant satisfied demand, W_o, to the constrained demand curve of the server, $\overline{\mathcal{C}}_s^d(t)$ (Line 7). Next, its updates the demand by replacing $W_{q,1}$ with the resultant unsatisfied demand, R_q (Line 8), and gets the first window of the updated $\mathcal{C}_s^d(t)$, $W'_{q,1}$ (Line 9). After that the algorithm computes a window, γ, with start and end time like those of W_o but shifted T_s time units (Line 10). This last steps is inline with the Sporadic Server algorithm as its budget is replenished only by the amount of consumed capacity, W_o, and T_s time units after the server became active, i.e. at $W_o.s + T_s$.

Depending on the outcome of the delta operation, the remaining supply exists or is empty. In the former case, i.e. when $R_p \neq \emptyset$, and in the event that R_p starts earlier than $W'_{q,1}$, R_p is shifted so that the start of the supply and that of the demand coincide (Lines 11-13). Likewise, if γ precedes the demand, then γ is shifted to the start of $W'_{q,1}$ (Line 14-15). In any case, $\mathcal{C}_p(t)$ is updated by the addition of R_p and γ (Line 16). In this fashion, any unused capacity is preserved until the start of the next request. These steps are repeated until the demand is empty.

Example 4.16. *Given the server S_2 from Table 4.2 with aggregated demand $C_{s_2}^d(t) = \{(2,3),(5,7),(10,14)\}$ with $t = 27$, Algorithm 11 performs the following steps to compute the constrained demand curve $\overline{\mathcal{C}}_{s_2}^d(t)$:*

1. *Initialize $\mathcal{C}_p(t)$ to $\{(W_{q,1}.s, W_{q,1}.s + C_s)\}$, where $W_{q,1}$ is the first window of the server's aggregated demand. Since $W_{q,1} = \{(2,3)\}$, $\mathcal{C}_p(t) = \{(2, 2 + C_s)\} = \{(2,4)\}$.*

2. *Fit $W_{q,1}$ into the first window of $\mathcal{C}_p(t)$, $W_{p,1}$, by computing $W_{p,1} \ominus W_{q,1}$ (Blue dashed arrows in Figure 4.17). In our case, $W_{p,1} \ominus W_{q,1} = \{(2,4)\} \ominus \{(2,3)\} = (R_p, W_o, R_q) = (\{(3,4)\}, \{(2,3)\}, \emptyset)$.*

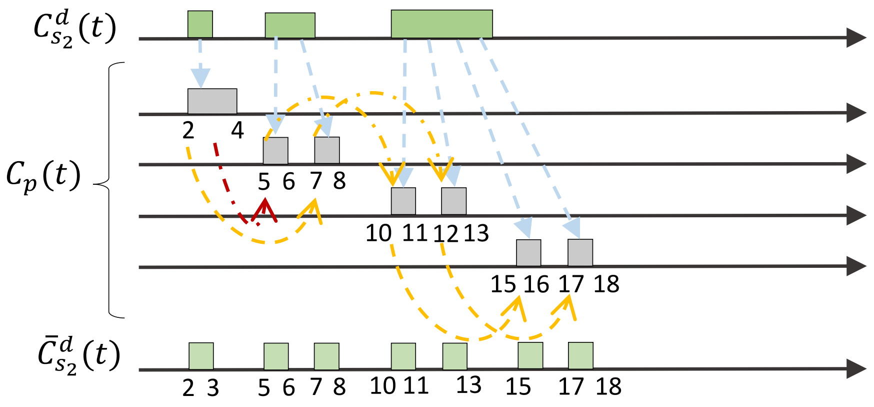


Figure 4.17 Constrained demand of a Sporadic Server. While blue arrows demonstrate how the demand is fit into the supply $\mathcal{C}_p(t)$, yellow arrows show how each budget replenishment takes place T_s times units after the server became active.

3. *Add the resultant satisfied demand, W_o, to the constrained demand curve, $\overline{\mathcal{C}}^d_s(t)$, and update $C^d_s(t)$ by replacing $W_{q,1}$ with the resultant unsatisfied demand, R_q. Thus, $\overline{\mathcal{C}}^d_s(t) = \{(2,3)\}$ and $C^d_s(t) = \{(5,7),(10,14)\}$. Observe that as the unsatisfied demand is $\emptyset$ (See previous step), $W_{q,1}$ is simply removed from $C^d_s(t)$. Moreover, for the sake of readability the index of the server is dropped.*

4. *Calculate $\gamma = \{(W_o.s + T_s, W_o.e + T_s)\}$, and obtain $W'_{q,1}$ from the new $C^d_s(t)$. In the example $\gamma = \{(2+5, 3+5)\} = \{(7,8)\}$, and $W'_{q,1} = \{(5,7)\}$.*

5. *Update R_p and γ if necessary. Since the remaining unused supply is different from $\emptyset$, and $R_p.s = 3 < W'_{q,1}.s = 5$, then $R_p = \{(W'_{q,1}.s, W'_{q,1}.s + R_p.l)\} = \{(5, 5+1)\} = \{(5,6)\}$. This is highlighted by the red dashed arrow in Figure 4.17. As $\gamma.s = 7 > W'_{q,1}.s = 5$, γ remains unchanged.*

6. *Update $\mathcal{C}_p(t)$ to $R_p \oplus \gamma$. Hence, $\mathcal{C}_p(t) = \{(5,6)\} \oplus \{(7,8)\} = \{(5,6)\} \cup \{(7,8)\}$.*

7. *Repeat the previous steps until there is no demand left. Eventually, $\overline{\mathcal{C}}^d_{s_2}(t) = \{(2,3),(5,6),(7,8),(10,11),(12,13),(15,16),(17,18)\}$ as depicted in Figure 4.17.*

Theorem 6. *The constrained demand curve of a Sporadic Server represents the exact windows of time in which the server serves its aggregated demand in isolation, i.e. without higher priority interference.*

Proof. In isolation a Sporadic Server serves its demand as soon as possible and in compliance with its replenishment rules. Thus, the theorem is proved by showing that the following loop invariant of Algorithm 11 holds. **Loop invariant**: At the start of each iteration loop, $\mathcal{C}_p(t)$ denotes the supply provided by a Sporadic Server at its next activation. **Initialization**: Previous to the first iteration of the loop, the server is idle. At its next activation, i.e. at the time of the arrival of its first request $W_{q,1}$, the budget of the server has not yet been consumed. Thus, $\mathcal{C}_p = \{(W_{q,1}.s, W_{q,1}.s + C_s)\}$. **Maintenance**: Assume that the loop invariant holds at the start of an iteration. According to Theorem 3 and Theorem 4, R_p and W_o represent the remaining supply and satisfied demand after fitting the demand window $W_{q,1}$ into $W_{p,1}$ respectively. Moreover, recall that the activation of a server occurs, when there is some pending request and there is some supply available. Furthermore, let $W'_{q,1}$ represent the start of the first window of the updated $\overline{\mathcal{C}}_s^d(t)$, i.e. after replacing $W_{q,1}$ with the unsatisfied demand R_q (Theorem 4). Hence, if $R_p \neq \emptyset$, either the next activation takes place at $R_p.s$ due to some pending request or at the arrival of the next request $W_{q,1}.s$. While in the former case R_p trivially equals the supply provided by the server at its next activation $R_p.s$, in the latter case this supply is given by a window starting at $W'_{q,1}.s$ and with a length equal to $R_p.l$. This remaining supply window coincides with the value stored on R_p at the end of the iteration. On the other hand, as the last activation occurs at $W_o.s$, the server must replenish his capacity T_s time units after $W_o.s$, and the replenishment amount must equal the used capacity, i.e. $W_o.l$. This supply can be denoted in the form of a window $\{(W_o.s + T_s, W_o.s + T_s + W_o.l)\} = \{(W_o.s + T_s, W_o.e + T_s)\}$. If the next activation takes place before (or at the same time as) $W_o.s + T_s$, then this window also represents the supply provided at the next activation. Otherwise, if the activation takes place after $W_o.s + T_s$, i.e. $W_o.s + T_s < W'_{q,1}.s$ (as $R_p.s$ cannot be greater than $W_o.s + T_s$), the replenishment supply at the next activation is given by a window starting at $W'_{q,1}.s$ and with a length equal to $W_o.l$. This supply window on account of the replenishment of the server's budget corresponds precisely to the value stored on γ at the end of the iteration. Eventually, the total supply provided by the server at the next iteration is given by the combination of any remaining unused supply and the supply due to the replenishment, i.e. $R_p \oplus \gamma$. Since $\mathcal{C}_p(t) = R_p \oplus \gamma$, the loop invariant holds true at the beginning of the next iteration. **Termination**: The loop terminates, when all the demand is served, and based on the invariant, $\overline{\mathcal{C}}_s^d(t)$ contains all the demand satisfied at the earliest (Theorem 5) which was obtained by fitting each window of the demand $\mathcal{C}_s^d(t)$ into the supply provided by a Sporadic Server, $\mathcal{C}_p(t)$, with period T_s and budget C_s at each activation. $\square$

4.7 Characteristic curves of unserved tasks and servers

In a given system, where tasks and servers are scheduled by a fixed priority preemptive mechanism, unserved tasks and servers can only execute in the absence of higher priority interference. In particular, servers need to comply with their budget constraints and replenishment algorithms according to their task demand. To that end, the next curves characterize all these requirements.

Definition 4.18. *Aggregated higher priority demand curve of unserved tasks and servers: Given an unserved task or a server, p_i, its aggregated higher priority demand curve, $C^d_{hp(p_i)}(t)$, denotes the aggregation of the* *demand of its higher priority unserved tasks and the constrained demand of its higher priority servers.*

Example 4.17. *Given the server S_2 whose higher priority server and unserved task are S_1 and τ_2 respectively (see Table 4.2), its aggregated higher priority demand curve, $C^d_{hp(s_2)}(t)$, up to $t = 27$, is given by $\overline{\mathcal{C}}^d_{s_1}(t) \oplus \mathcal{C}^d_{\tau_2}(t)$ $= \{(0,1),(6,7),(12,13),(18,19),(24,25)\} \oplus \{(4,6),(9,11),(14,16),(19,21),(24,26)\} = \{(0,1),(4,7),(9,11),(12,13),(14,16),(18,21),(24,27)\}$ as shown in Figure 4.18.*

Observe that in the previous example $\overline{\mathcal{C}}^d_{s_1}(t)$ and $\mathcal{C}^d_{\tau_1}(t)$ are the same since τ_1 is the highest priority task in the system and presents a period and worst-case execution time equal to the period and budget of its server S_1.

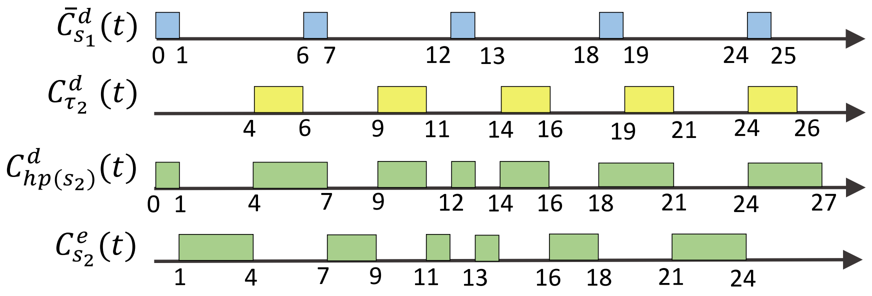


Figure 4.18 The aggregated higher priority demand of server S_2 (See Table 4.2), $C^d_{hp(s_2)}(t)$, is computed as the aggregation of the constrained demand of S_1 and the demand of τ_2. Its unconstrained execution, $C^e_{s_2}(t)$, is given by the holes left by $C^d_{hp(s_2)}(t)$.

Definition 4.19. *Unconstrained execution curve of unserved tasks and servers: Given an unserved task or server, p_i, its unconstrained execution curve, $C^e_{p_i}(t)$ is calculated as the remaining unused supply of the delta of the total supply of the system up to time t, and its aggregated higher priority demand, i.e. $\{(0,t)\} \ominus C^d_{hp(p_i)}(t)$.*

Example 4.18. *Given the server S_2 from Table 4.2 with $t = 27$ and $C^d_{hp(s_2)}(t) = \{(0,1),(4,7),(9,11),(12,13),(14,16),(18,21),(24,27)\}$ (Example 4.17), its unconstrained execution curve, $C^e_{s_2}(t)$, is calculated as the remaining unused supply of $\{(0,t)\} \ominus C^d_{hp(s_2)}(t) = \{(0,27)\} \ominus \{(0,1),(4,7),(9,11),(12,13),(14,16),(18,21),(24,27)\} = (\{(1,4),(7,9),(11,12),(13,14),(16,18),(21,24)\}, C^d_{hp(s_2)}(t), \emptyset)$, i.e. $C^e_{s_2}(t) = \{(1,4),(7,9),(11,12),(13,14),(16,18),(21,24)\}$ as illustrated in Figure 4.18.*

Theorem 7. *The unconstrained execution curve, $C^e_{p_i}(t)$, of an unserved task or server, p_i, represents the windows of time in which p_i may execute in the absence of any higher priority interference.*

Proof. According to Theorem 5, the remaining unused supply, $\mathcal{C}'_p(t)$, of $\{(0,t)\} \ominus C^d_{hp(p_i)}(t)$, denotes the remaining supply after inserting the demand $C^d_{hp(p_i)}(t)$ into the supply $\{(0,t)\}$ at the earliest, i.e. $\mathcal{C}'_p(t) = C^e_{p_i}(t)$ represents the gaps left after serving any higher priority interference of p_i. $\square$

The presented analysis, as exposed later in this work, provides a basis for the proper dimensioning of fixed priority servers. For example, in order to determine if a server S can guarantee its budget C_s every T_s time units, we split its unconstrained execution curve, $C^e_s(t)$, in intervals equal to T_s, and check if the capacity of each resultant sub-curve is at least C_s. If not, then the server cannot guarantee its budget. It is worth pointing out that approaches extending the busy window analysis, such as [17], take it for granted that a server guarantees its budget, which, as shown, is not trivial to determine.

For instance, consider the previous example, where $C^e_{s_2}(t) = \{(1,4),(7,9),(11,12),(13,14),(16,18),(21,24)\}$, and assume that $C_{s_2} = 3$ (as opposed to Table 4.2) and $T_{s_2} = 5$. The resultant sub-curve $\tilde{\mathcal{C}}^e_{s_2,1}(t)$ after performing $split(C^e_{s_2}(t), T_{s_2})$ is $\{(7, 9)\}$ with a capacity $cap(\tilde{\mathcal{C}}^e_{s_2,1}(t)) = 2 < C_{s_2} = 3$. Thus, $C_{s_2} = 3$ and $T_{s_2} = 5$ are not suitable parameters for S_2 as the server cannot guarantee its budget.

Unlike an unserved task, a server, however, cannot thoroughly use all the supply given by its unconstrained execution curve due to its budget and replenishment constraints, as well as the arrival and execution demand pattern of its workload. Thus, Algorithm 12 presents a method to compute the exact windows of time where the requests of a server are served, considering any higher priority interference.

Definition 4.20. *Actual Execution Curve of a server: Given a server S with constrained demand curve $\overline{\mathcal{C}}^d_s(t)$, unconstrained execution curve $\mathcal{C}^e_s(t)$, period T_s, and budget C_s, its actual execution curve $\overline{\mathcal{C}}^e_s(t)$, is obtained as specified by Algorithm 12.*

Algorithm 12 Actual execution curve of a server

```
1: Input: $\overline{\mathcal{C}}^d_s(t)$, $\mathcal{C}^e_s(t)$,$T_s$
2: $\bigcup_{k=0}^{v} \overline{\mathcal{C}}_{s,k}(t) = split(\overline{\mathcal{C}}^d_s(t), T_s)$ with $v = \lceil t/T_s \rceil - 1$
3: $\bigcup_{k=0}^{v} \tilde{\mathcal{C}}_{s,k}(t) = split(\mathcal{C}^e_s(t), T_s)$ with $v = \lceil t/T_s \rceil - 1$
4: $\overline{\mathcal{C}}^e_s(t) = \emptyset$
5: for $k \leftarrow 0$ to $v$ do
6:     $(\mathcal{C}'_p(t), \mathcal{C}_o(t), \mathcal{C}'_q(t)) = \tilde{\mathcal{C}}_{s,k}(t) \ominus \overline{\mathcal{C}}_{s,k}(t)$
7:     Get $(\mathcal{C}^c_o(t), W^r) = constr(\mathcal{C}_o(t), C_s, (k+1) \cdot T_s)$
8:     $\overline{\mathcal{C}}^e_s(t) = \overline{\mathcal{C}}^e_s(t) \oplus \mathcal{C}^c_o(t)$
9:     Get $W'_q = trans(\mathcal{C}'_q(t), (k+1) \cdot T_s)$
10:    if $k < v$ then
11:        $\overline{\mathcal{C}}_{s,k+1}(t) = \overline{\mathcal{C}}_{s,k+1}(t) \oplus W'_q \oplus W^r$
12: Return $\overline{\mathcal{C}}^e_s(t)$
```

The algorithm begins by splitting the constrained demand curve and unconstrained execution curve into intervals, $\overline{\mathcal{C}}_{s,k}(t)$ and $\tilde{\mathcal{C}}_{s,k}(t)$ respectively, of length T_s (Line 2-3). For each of these sub-curves, it then fits the demand into the supply, i.e. $\tilde{\mathcal{C}}_{s,k}(t) \ominus \overline{\mathcal{C}}_{s,k}(t)$ (Line 6). Next, it makes sure that the total capacity of the resultant satisfied demand, $\mathcal{C}_o(t)$, does not exceed the budget of the server, and stores any excess on

W^r (Line 7). While the constrained satisfied demand, $\mathcal{C}_o^c(t)$, is stored in the server's actual execution curve (Line 8), the unsatisfied demand transformed into a window W_q' (Line 9) as well as W^r are pushed to the next interval (Line 11).

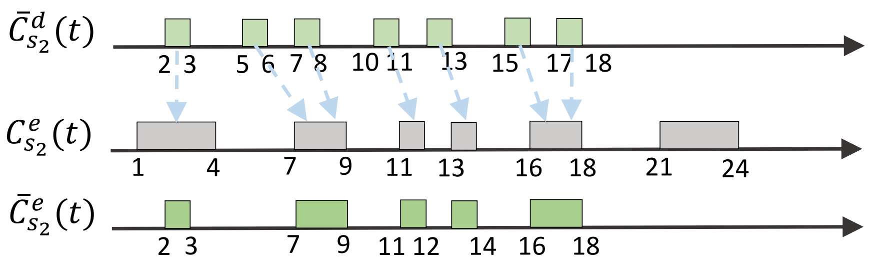


Figure 4.19 In order to obtain the actual execution curve of S_2, Algorithm 12 fits its constrained demand into its unconstrained execution curve (Blue dashed arrows) placing capacity constraints according to the Sporadic Server algorithm.

Example 4.19. *Consider the server S_2 from Table 4.2 with $t = 27$ and constrained demand curve $\overline{\mathcal{C}}_{s_2}^d(t) = \{(2,3),(5,6),(7,8),(10,11),(12,13),(15,16),(17,18)\}$ (Example 4.16), and unconstrained execution curve $\mathcal{C}_{s_2}^e(t) = \{(1,4),(7,9),(11,12),(13,14),(16,18),(21,24)\}$ (Example 4.18). To compute its actual execution curve $\overline{\mathcal{C}}_{s_2}^e(t)$, Algorithm 12 performs the following steps:*

1. *Get $\overline{\mathcal{C}}_{s,k}(t)$ by splitting $\overline{\mathcal{C}}_s^d(t)$ into intervals of T_s. Note that for the sake of readability the index of the server is dropped. In our case, split$(\overline{\mathcal{C}}_s^d(t), T_s) = \{(2,3)\} \cup \{(5,6),(7,8)\} \cup \{(10,11),(12,13)\} \cup \{(15,16),(17,18)\}$.*

2. *Obtain $\tilde{\mathcal{C}}_{s,k}(t)$ by splitting $\mathcal{C}_s^e(t)$ into intervals of T_s. In the example, split$(\mathcal{C}_s^e(t), T_s) = \{(1,4)\} \cup \{(7,9)\} \cup \{(11,12),(13,14)\} \cup \{(16,18)\} \cup \{(21,24)\}$.*

3. *Compute the delta of $\tilde{\mathcal{C}}_{s,0}(t)$ and $\overline{\mathcal{C}}_{s,0}(t)$. $(\mathcal{C}_p'(t), \mathcal{C}_o(t), \mathcal{C}_q'(t)) = \tilde{\mathcal{C}}_{s,0}(t) \ominus \overline{\mathcal{C}}_{s,0}(t) = \{(1,4)\} \ominus \{(2,3)\} = (\{(1,2),(3,4)\}, \{(2,3)\}, \emptyset)$*

4. *Calculate $(\mathcal{C}_o^c(t), W^r) = constr(\mathcal{C}_o(t), C_s, T_s)$ and add the constrained satisfied demand, $\mathcal{C}_o^c(t)$, to the actual execution curve, $\overline{\mathcal{C}}_s^e(t)$. Since the total capacity of $\mathcal{C}_o(t) < C_s$, then $\mathcal{C}_o^c(t) = \mathcal{C}_o(t) = \{(2,3)\}$ and $W^r = \emptyset$. Hence, $\overline{\mathcal{C}}_s^e(t) = \{(2,3)\}$.*

5. *Transform the unsatisfied demand into a window, W_q', starting at T_s, i.e. $W_q' = trans(\mathcal{C}_q'(t), (k+1) \cdot T_s)$. As there is no unsatisfied demand $W_q' = \emptyset$.*

6. *Add W^r and W_q' to $\overline{\mathcal{C}}_{s,1}(t)$. In our case, $\overline{\mathcal{C}}_{s,1}(t)$ remains unchanged, since both windows are empty.*

7. *Repeat the same steps for the remaining sub-curves* $\overline{\mathcal{C}}_{s,1}(t)$, $\overline{\mathcal{C}}_{s,2}(t)$, *and* $\overline{\mathcal{C}}_{s,3}(t)$. *Eventually* $\overline{\mathcal{C}}^e_s(t)=$ $\{(2,3),(7,9),(11,12),(13,14)(16,18)\}$ *as depicted in Figure 4.19.*

Observe that Algorithm 12 computes the actual execution curve of a server based upon $\overline{\mathcal{C}}^d_s(t)$, i.e. the algorithm calculates $\overline{\mathcal{C}}^e_s(t)$ for any kind of fixed priority server as long as its constrained execution curve is known, which shows that the proposed framework enjoys strong composability properties.

Theorem 8. *The actual execution curve of a server represents the exact windows of time in which the server executes its aggregated demand, taking into account any higher priority interference.*

Proof. To prove the theorem, we show that the following loop invariant of Algorithm 12 holds. **Invariant**: At the start of every iteration of the loop, while $\overline{\mathcal{C}}^e_s(t)$ is made up of all the possible satisfied demand, up to this point, constrained to the server's budget C_s, $\overline{\mathcal{C}}_{s,k}(t)$ is composed of the original demand lying within $[k \cdot T_s, (k+1).T_s)$ plus any previous unsatisfied demand. **Initialization**: Before the first iteration of the loop, there is no demand served and so $\overline{\mathcal{C}}^e_s(t)$ must be empty. Furthermore, as there cannot be any previous unsatisfied demand, by definition $\overline{\mathcal{C}}_{s,k}(t)$ consists of the demand lying within $[k \cdot T_s, (k+1) \cdot T_s)$. Then the loop invariant holds true trivially. **Maintenance**: Assuming the loop invariant holds true for the n-th iteration, then $\overline{\mathcal{C}}^e_s(t)$ contains all the possible satisfied demand up to this point, i.e. due to $\overline{\mathcal{C}}_{s,0}(t) \cup \overline{\mathcal{C}}_{s,1}(t) \cup \ldots \cup \overline{\mathcal{C}}_{s,n-1}(t)$, and constrained to the server's capacity. As per Theorem 5, the resultant $\mathcal{C}_o(t)$ and $\mathcal{C}'_q(t)$ of $\tilde{\mathcal{C}}_{s,n}(t) \ominus \overline{\mathcal{C}}_{s,n}(t)$ are the maximum possible demand that could be fit into $\tilde{\mathcal{C}}_{s,n}(t)$, and the remaining unsatisfied demand respectively. Hence, $constr(\mathcal{C}_o(t), C_s, (n+1) \cdot T_s)$ results in a curve $\mathcal{C}^c_o(t)$, representing that maximum demand that can be served within the server's budget, and a window W^r holding any remaining demand. Adding $\mathcal{C}^c_o(t)$ to $\overline{\mathcal{C}}^e_s(t)$, as well as adding W^r and $\mathcal{C}'_q(t)$ (in form of a window $W'_q = trans(\mathcal{C}'_q(t), (n+1) \cdot T_s)$) to $\overline{\mathcal{C}}_{s,n+1}(t)$ implies that the loop invariant is maintained from any n-th iteration to a $(n+1)$-th iteration. **Termination**: At the end of the loop $k = v$, and according to the invariant the initial demand, $\overline{\mathcal{C}}^d_s(t)$, is maximally satisfied without exceeding C_s every T_s time units. Additionally, since the supply is given by the unconstrained execution curve of the server, $\mathcal{C}^e_s(t)$, then the windows composing $\overline{\mathcal{C}}^e_s(t)$ represent the execution of the constrained demand within the gaps left by the execution of any higher priority task or server. On account of the fact that the constrained demand curve of a server represents the exact times slots in which the server serves its aggregated demand in isolation, then in the presence of higher priority interference, the actual execution curve as defined in Algorithm 12 denotes the exact window of time in which the server executes its aggregated demand in a fixed priority preemptive system. □

4.8 Characteristic curves of served tasks

In the same way as the actual execution curve was computed for servers, we can also obtain the actual execution curve of served and unserved tasks by fitting their higher priority demand into its unconstrained execution curve. To do so, we first introduce the notion of aggregated higher priority demand and unconstrained execution curve for a served task.

Definition 4.21. *Aggregated higher priority demand curve of served tasks: Given a served task, τ_i, its aggregated higher priority demand curve, $C^d_{hp(\tau_i)}(t)$, is the aggregation of the demand of the higher priority tasks scheduled by the same server.*

Example 4.20. *Given the task τ_5 from Table 4.2, its aggregated higher priority demand curve, $C^d_{hp(\tau_5)}(t)$, up to time $t = 27$, is given by the aggregation of the other two tasks scheduled by the same server S_2, namely τ_3 and τ_4, i.e. $C^d_{hp(\tau_5)}(t) = \mathcal{C}^d_{\tau_3}(t) \oplus \mathcal{C}^d_{\tau_4}(t) = \{(2,3),(5,7)\}$ as depicted in Figure 4.20.*

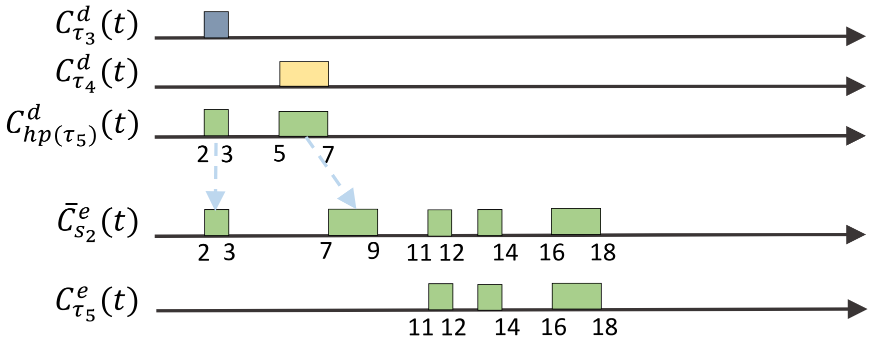


Figure 4.20 The aggregated higher priority demand curve of τ_5, $C^d_{hp(\tau_5)}(t)$, is computed as the aggregation of the demand of τ_3 and τ_4. Its unconstrained execution curve, $C^e_{\tau_5}(t)$, is given by the unused supply after fitting $C^d_{hp(\tau_5)}(t)$ into the actual execution of its server, $\overline{\mathcal{C}}^e_{s_2}(t)$.

Definition 4.22. *Unconstrained execution curve of served tasks: Given a served task, τ_i, its unconstrained execution curve, $C^e_{\tau_i}(t)$, is calculated as the unused supply of the delta of the actual execution curve of its server, $\overline{\mathcal{C}}^e_s(t)$, and the aggregated higher priority demand of the task, i.e. $\overline{\mathcal{C}}^e_s(t) \ominus C^d_{hp(\tau_i)}(t)$.*

Example 4.21. *Continuing with τ_5 from Example 4.20, its unconstrained execution curve $\mathcal{C}^e_{\tau_5}(t)$, is given by the unused supply of $\overline{\mathcal{C}}^e_{s_2}(t) \ominus C^d_{hp(\tau_5)}(t) = \{\{(2,3),(7,9),(11,12),(13,14)(16,18)\} \ominus \{(2,3),(5,7)\}$, i.e. $\mathcal{C}^e_{\tau_5}(t) = \{(11,12),(13,14)(16,18)\}$ as shown in Figure 4.20.*

Theorem 9. *The unconstrained execution curve, $C^e_{\tau_i}(t)$, of a served task, τ_i, represents the windows of time in which τ_i may execute in the absence of any higher priority interference.*

Proof. According to Theorem 5, the remaining unused supply, $\mathcal{C}'_p(t)$, of $\overline{\mathcal{C}}^e_s(t) \ominus C^d_{hp(\tau_i)}(t)$, denotes the remaining supply after inserting the task's aggregated higher priority demand, $C^d_{hp(\tau_i)}(t)$, into the supply, $\overline{\mathcal{C}}^e_s(t)$, at the earliest. As $\mathcal{C}'_p(t) = C^e_{p_i}(t)$, then the unconstrained execution curve, $C^e_{p_i}(t)$, represents the gaps left after the server executes any higher priority task co-scheduled along τ_i. □

Regardless of being served or unserved, and akin to servers, tasks also present a characteristic curve that denotes the exact windows of time in which their jobs execute, as shown next.

Definition 4.23. *Actual execution curve of a task: Given a task τ_i with demand $\mathcal{C}^d_{\tau_i}(t)$, and unconstrained execution curve $\mathcal{C}^e_{\tau_i}(t)$, its actual execution curve, $\overline{\mathcal{C}}^e_{\tau_i}(t)$, is given by the satisfied demand of $\mathcal{C}^e_{\tau_i}(t) \ominus \mathcal{C}^d_{\tau_i}(t)$.*

Example 4.22. *Carrying on with τ_5 from Example 4.21, its actual execution curve $\mathcal{C}^e_{\tau_5}(t)$, is given by the satisfied demand of $\mathcal{C}^e_{\tau_5}(t) \ominus \mathcal{C}^d_{\tau_5}(t) = \{(11,12),(13,14)(16,18)\} \ominus \{(10,14)\}$, i.e. $\mathcal{C}^e_{\tau_5}(t) = \{(11,12),(13,14)$ $(16,18)\}$ as exposed in Figure 4.21.*

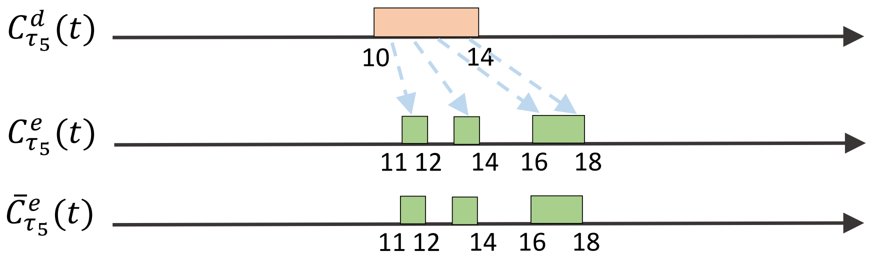


Figure 4.21 The actual execution curve of a task is given by the resultant satisfied demand after fitting its demand into its unconstrained execution curve (Blue dashed arrows).

Theorem 10. *The actual execution curve of a task represents the exact windows of time in which its demand is executed taking account of any higher priority interference.*

Proof. According to Theorem 5, the resultant $\mathcal{C}_o(t)$ of $\mathcal{C}^e_{\tau_i}(t) \ominus \mathcal{C}^d_{\tau_i}(t)$, denotes the maximum satisfied demand after serving $\mathcal{C}^d_{\tau_i}(t)$ with the supply $\mathcal{C}^e_{\tau_i}(t)$ at the earliest possible time. Moreover, as stated in Theorem 7 and Theorem 9, $\mathcal{C}^e_{\tau_i}(t)$ denotes the supply left by the execution of the task's higher priority tasks or servers. Thus, its actual execution curve, $\mathcal{C}_o(t) = \overline{\mathcal{C}}^e_{\tau_i}(t)$ represents the exact time slots, where the task's demand, $\mathcal{C}^d_{\tau_i}(t)$, is executed. □

4.9 Summary

In this chapter, basic concepts and operations were introduced in order to model tasks and servers by means of curves. In the next chapter, these characteristic curves will allow us to provide an exact response time analysis for fixed priority systems based on fixed priority servers in a multi-level scheduling setting under preemptive scheduling. Moreover, an experimental characterization of the schedulabilty improvement that can be obtained with respect to existing sufficient schedulability tests will be shown, proving the effectiveness of the proposed exact analysis.

CHAPTER 5
RESPONSE TIME ANALYSIS

Modified from a journal paper[39] published in the Journal of Systems Architecture: Embedded Software Design (JSA)

Jorge Martinez[17,18], Dakshina Dasari[19], Arne Hamann[20], Ignacio Sañudo[21] , Marko Bertogna[22]

In this chapter, based on the presented characteristic curves of tasks and servers, an exact response time analysis for tasks scheduled by fixed priority servers in a multi-level scheduling setting under preemptive scheduling is provided. Furthermore, it is shown by means of experiments that the proposed analysis outperforms existing sufficient schedulability tests. It is worth mentioning that the system model is the same as that of Chapter 4.

5.1 Response Time Analysis

The actual execution curve of a task represents the actual windows of time where its jobs get to be executed, so, in order to compute their response time, it is sufficient to step through this curve as described below.

Definition 5.24. *Demand of a job: The demand of the j-th job of task τ_i can be modelled by a window $W_{i,j} = \{(a_{i,j}, a_{i,j} + c_{i,j})\}$, where $a_{i,j}$ and $c_{i,j}$ represent the arrival time and execution demand of the job respectively. Moreover, $\overline{C}_{i,j} = \sum_{k=1}^{j} W_{i,k}.l$ denotes the cumulative demand up to this job.*

Theorem 11. *Job response time: Given a task τ_i with actual execution curve $\overline{\mathcal{C}}^{e}_{\tau_i}(t)$, and cumulative demand up to its j-th job, $\overline{C}_{i,j}$, let W^{last} be the last window in the constrained curve of $constr(\overline{\mathcal{C}}^{e}_{\tau_i}(t), \overline{C}_{i,j}, 0)$, then the response time of this job, $R_{i,j}$, is given by $W^{last}.e - a_{i,j}$.*

Proof. We know that the response time of the j-th job released by task τ_i, $R_{i,j}$, is given by the difference between its finishing time, i.e. the time at which its execution ends, denoted by $f_{i,j}$ and its arrival time $a_{i,j}$, i.e. $R_{i,j} = f_{i,j} - a_{i,j}$. As per Definition 4.11, the resultant $\mathcal{C}^c_p(t)$ of $constr(\overline{\mathcal{C}}^{e}_{\tau_i}(t), \overline{C}_{i,j}, 0)$ is a curve whose total capacity equals the cumulative demand up to the j-th job $\overline{C}_{i,j}$. If $\mathcal{C}^c_p(t) = \{W^1, W^2, \ldots, W^{last}\}$, then $W^{last}.e$ represents the point in time at which the execution of $\tau_{i,j}$ is completed. Hence, $R_{i,j} = W^{last}.e - a_{i,j}$. $\square$

[17]Graduate student at the University of Modena and Reggio Emilia
[18]Primary researcher and author
[19]Researcher at Robert Bosch GmbH
[20]Researcher at Robert Bosch GmbH
[21]Postgraduate researcher at the University of Modena and Reggio Emilia
[22]Full Professor at the Univeristy of Modena and Reggio Emilia

Example 5.23. *Proceeding with task τ_5 from Example 4.22, with cumulative demand up to its first job $\overline{C}_{5,1} = 4$ and $a_{5,1} = 10$, the response time of this first job, $R_{5,1}$, is given by $W^{last}.e - a_{5,1}$, where W^{last} is the last window in the constrained curve of $constr(\overline{\mathcal{C}}^{e}_{\tau_5}(t), \overline{C}_{5,1}, 0) = constr(\{(11,12),(13,14)(16,18)\}, 4, 0) = (\{(11,12),(13,14)(16,18)\}, \emptyset)$. From this it follows that $W^{last} = \{(16,18)\}$, and hence $R_{5,1} = 18 - 10 = 8$.*

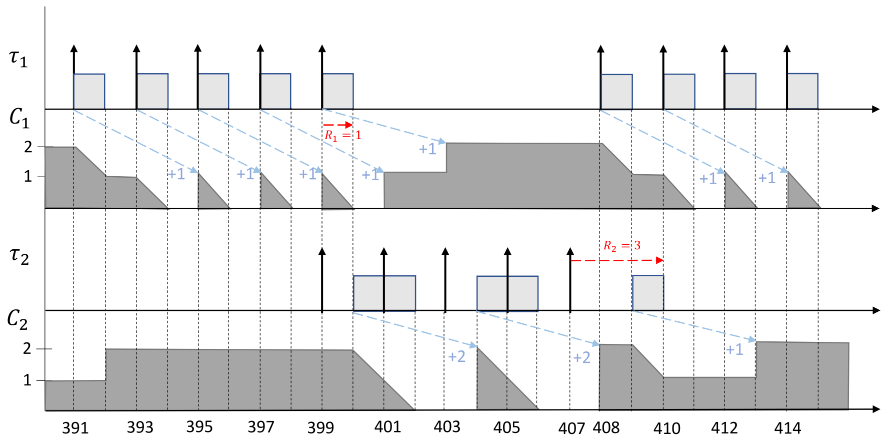


Figure 5.1 Gantt chart of the system described in Table 5.1.

5.1.1 Aperiodic jobs

Section 4.3 shows that the analysis in [17] is not exact for periodic tasks, however, in the following example, it is also shown that [17] is not exact for sporadic tasks either. To that end, consider a system composed of two Sporadic Servers HP and LP, where the higher (resp. lower) priority server HP (resp. LP) handles a sporadic task τ_1 (resp. τ_2) with parameters given in Table 5.1. Assume that while τ_1 arrives at time $17 \cdot N$, $17 \cdot N + 2$, $17 \cdot N + 4$, $17 \cdot N + 6$, and $17 \cdot N + 8$ $\forall N \in \mathbb{N}$, τ_2 arrives at time $19 \cdot N$, $19 \cdot N + 2$, $19 \cdot N + 4$, $19 \cdot N + 6$, and $19 \cdot N + 8$ $\forall N \in \mathbb{N}$ (Upward arrows denote their arrivals in Figure 5.1). Moreover, while these tasks have an execution time of 1, T_i in Table 5.1 denotes the minimum inter-arrival time for each of them.

Server	C_s	T_s
HP	2	4
LP	2	4

Task	C_i	T_i
1	1	2
2	1	2

Table 5.1 Parameters of the servers and sporadic tasks of the second counterexample.

In line with [17], to compute the worst-case response time of τ_2, R_2, the critical instant occurs when (i) τ_2 arrives just after its server's capacity has been exhausted, and (ii) the capacity of LP is replenished at the start of its next period but with a deferred execution of the server due to interference from HP. Thus, R_2 is given by $w + T_{LP} - C_{LP}$, where $w = L_2(w) + (\lceil\ L_2(w)/C_{LP} \rceil - 1)(T_{LP} - C_{LP}) + I(w) = C_2 + \lceil\ w/T_{HP} \rceil \cdot C_{HP}$.

The recurrence starts with $w = C_2 + (\lceil C_2/C_{LP} \rceil - 1)(T_{LP} - C_{LP}) = 1$ and ends with $w = 3$. As a result $R_2 = w + T_{LP} - C_{LP} = 3 + 4 - 2 = 5$. In the same fashion, it can be shown that according to [17] the worst-case response time of τ_1, R_1 is 3.

The Gantt Chart of Figure 5.1, however, shows that R_1 and R_2 (highlighted in bold red text) are actually 1 and 3 respectively, which proves that the analysis in [17] is not exact for sporadic tasks neither. Moreover, the picture shows that condition (i) of the critical instant of τ_2 does not hold, which explains the pessimism of the analysis.

Fortunately, the analysis proposed in this dissertation can be used to compute the response time of sporadic tasks as well as aperiodic jobs, as shown by the next example. Note that while for aperiodic jobs their exact arrival trace is needed, in the case of sporadic tasks a minimum inter-arrival time must be provided.

Example 5.24. *Assume that an unserved aperiodic task τ_6 is added to the system described in Table 4.2, with a priority lower than that of τ_5. Moreover, the demand of its first and second job is $W_{6,1} = \{(2,3)\}$ and $W_{6,2} = \{(10,12)\}$, which in turn leads to a demand, $\mathcal{C}^d_{\tau_6}(t)$, and an aggregated higher priority demand, $\mathcal{C}^d_{hp(\tau_6)}(t)$, equal to $\{(2,3),(10,12)\}$ and $\{(0,1),(2,3)(4,21),(24,27)\}$, respectively (refer to Figure 5.2). Hence, its unconstrained execution curve, $\mathcal{C}^e_{\tau_6}(t)$, is $\{(1,2),(3,4),(21,24)\}$. It can also be found that its actual execution curve, $\overline{\mathcal{C}}^e_{\tau_6}(t)$, is $\{(3,4),(21,23)\}$ by fitting $\mathcal{C}^d_{\tau_6}(t)$ into $\mathcal{C}^e_{\tau_6}(t)$ as illustrated by the blue dashed arrows in Figure 5.2. Thus, the response time of its second job, $R_{6,2}$, is given by $W^{last}.e - a_{6,2}$. In order to obtain the first term, we compute $constr(\overline{\mathcal{C}}^e_{\tau_6}(t), \overline{C}_{6,2}, 0) = (\{(3,4),(21,23)\}, 3, 0) = (\{(3,4),(21,23)\}, \emptyset)$. And so, $R_{6,2} = W^{last}.e - a_{6,2} = 23 - 10 = 13$.*

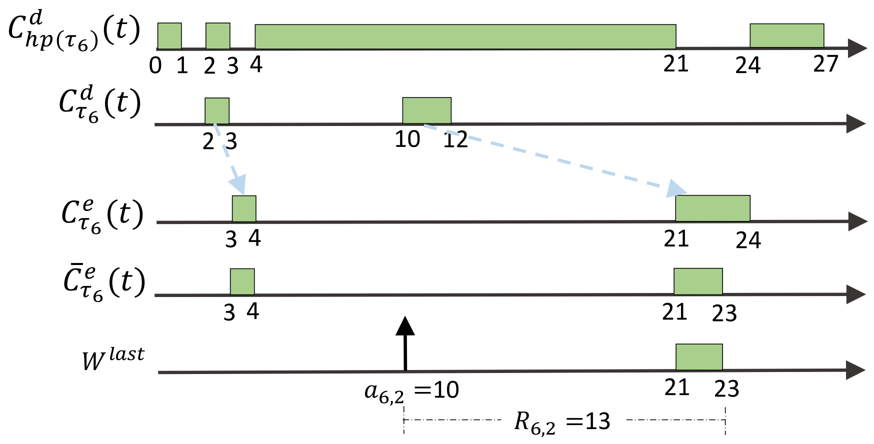


Figure 5.2 From top to bottom : Aggregated higher priority demand, task demand, unconstrained and actual execution curve of τ_6. The response time of the second aperiodic job, $R_{6,2}$, is given by $W^{last}.e - a_{6,2}$.

5.2 Evaluation

The tightness of the presented response time analysis is hereafter evaluated in an experimental setting based on LITMUS^{RT}. LITMUS^{RT} is a real-time extension of the Linux kernel allowing the implementation of different kind of servers. It is also shown that the method proposed in [17] is not exact and that considerable improvements in task schedulability can be obtained through the method presented in the previous chapter.

To that end, for each of the following experiments, the worst-case response times, R_i, obtained with both methods is compared against the actual measurements obtained in LITMUS^{RT}. Each task set was run for 3 seconds, thereby producing a significant number of jobs. For each set of experiments, we plot a *cumulative distribution function* (CDF) of the ratio between the worst-case response time ($WCRT$) and the period of tasks, i.e. the percentage of tasks whose ratio is less than or equal to R_i/T_i.

Tasks are uniformly sampled from $\{1, 2, 10, 100, 1000\}$ms, and their utilizations are generated as per the *UUnifast* algorithm [52] with a total system utilization of 0.7. These synthetic tasks are randomly assigned to servers, whose periods, T_s, are also sampled from the same uniform distribution. Server utilizations, U_s, are computed as the sum of the utilization of the served tasks, $\overline{U}_s$, times a random number, β_s, between 1.1 and 1.2. Server budgets, C_s, are therefore obtained as $C_s = \beta_s \cdot \overline{U}_s \cdot T_s$. Servers and tasks' priorities are randomly assigned.

The following set of experiments aims at showing the tightness of the proposed response time analysis for tasks scheduled by fixed priority servers. For each subset of experiments, unless otherwise stated, 500 task sets were randomly generated, where the number of servers, served tasks, and unserved tasks (if applicable) in each task set varies from 1 to 5, 2 to 5, and 2 to 3, respectively.

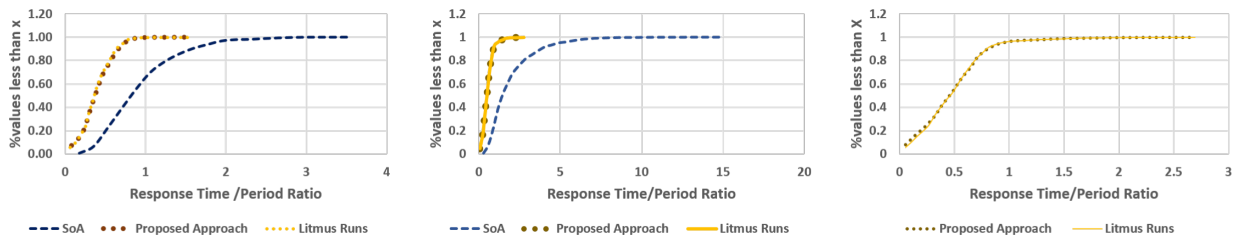


Figure 5.3 Single DS (left) multiple DSs (middle) multiple deferrable servers with offsets (right)

5.2.1 Single Deferrable Server

500 task sets were considered where each task set had 5 tasks randomly generated and allocated to a single deferrable server. The tasks had zero offsets and the total load of each task set was 60%. It is clearly seen in Figure 5.3(a) that [17] estimates the response times of around 40% of all tasks to be greater than their

periods, while the proposed approach calculates values that are very close to those measured in LITMUS^{RT}. In general, the response times computed by [17] are much higher.

5.2.2 Multiple Deferrable Servers

500 task sets were considered where each task set had 7 tasks randomly spread across 2 DSs. The tasks had no offsets and the total load of each task set was 70%. It can be seen in Figure 5.3(b) that the proposed approach clearly outperforms [17]. The latter estimates the response times of around 62% of all tasks to be larger than their periods, whereas LITMUS^{RT} and the proposed approach show that this is only the case for around 5.7% of all tasks. The experiment also demonstrates that the pessimism of existing SoA approaches keeps increasing and does not scale to multiple servers.

5.2.3 Multiple Deferrable Servers with offsets

500 task sets were considered, each with a load of 70%, where each task set had 5 tasks randomly spread across 2 DSs. Every task had an offset that was randomly chosen between 0 and 0.4 times its period. Since [17] cannot deal with offsets, the analysis was not part of the comparison. It can be seen in Figure 5.3(c) that the proposed analysis is exact.

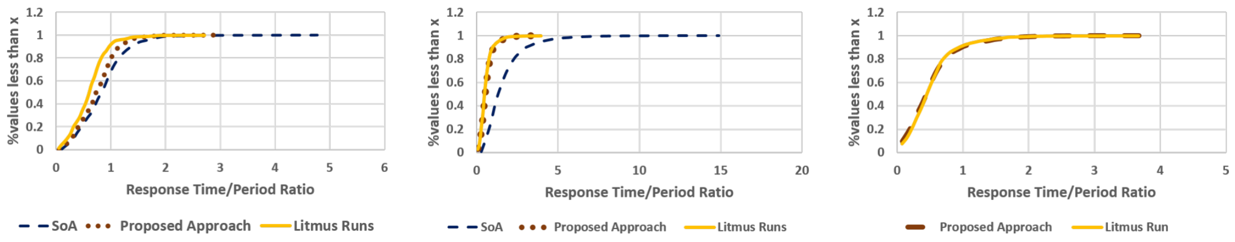


Figure 5.4 Multiple extended PPSs (left) mixed servers (middle) mixed servers with offsets (right)

5.2.4 Multiple extended Polling Periodic Servers

Figure 5.4(a) illustrates the results for 500 task sets with a total load of 60% where 7 tasks were distributed across 3 servers. It can clearly be seen that while the proposed approach provides bounds close to that of LITMUS^{RT}, [17] calculates more pessimistic response times. Note that since [17] does not deal with extended Polling Periodic Servers, its results refer to classic Polling Periodic Servers.

5.2.5 Mixed servers

In this experiment, a mix of DSs and extended PPSs are considered. Figure 5.4(b) shows the results across 500 task sets where each task set had 10 tasks randomly assigned to one of the three present servers. The type of each server was also randomly chosen. Again the proposed approach calculates response times that are very close to the actually observed executions in LITMUS^{RT}, whereas [17] is far more conservative. T[17] reports that around 73% of the samples have a response time greater than that the task period, while only 9% of the samples observed on LITMUS^{RT} exhibit such a response time.

5.2.6 Mixed servers with offsets

A mix of Deferrable and Polling Periodic Servers is considered. Figure 5.4(c) shows the results across 500 task sets, where each task set had 8 tasks randomly assigned to one of the three servers, whose type was also randomly chosen. Additionally, each task had an offset which was randomly chosen between 0 and 0.4 times its period. Figure 5.4(c) shows results that prove that the proposed analysis is exact.

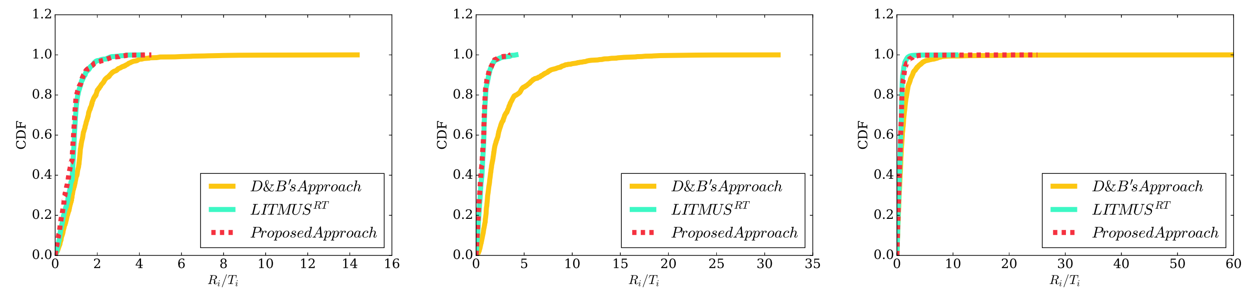


Figure 5.5 Served tasks without offsets (left) with offsets (middle) with unserved tasks (right)

5.2.7 Multiple Sporadic Servers

Figure 5.5 (left) shows the results of the first subset of experiments consisting of served tasks mapped onto Sporadic Servers. The figure shows that while the approach in [17] estimates that 40% of tasks have a $WCRT$ greater than their period, the proposed method and the LITMUS^{RT} simulation indicate that the actual percentage is 60%.

5.2.8 Multiple Sporadic Servers with offsets

The second subset of experiments evaluates the impact of offsets to schedulability, randomly assigning offsets to tasks in a range of $0-30\%$ times their period. Even though the analysis in [17] does not deal with offsets, we use it to evaluate the pessimism of a zero-offset analysis. Results are shown in Figure 5.5 (middle), where 40% of tasks have a ratio less than 1 according to our analysis and in line with the LITMUS^{RT} simulation, decreasing to 15% with the existing pessimistic approach. Furthermore, the latter analysis expects that 20% of tasks present a ratio greater than 5 but less than 32, although the highest ratio observed in the experiments is 4.

5.3 Multiple Sporadic Servers and unserved tasks

The last group of this set of experiments aims at comparing results when unserved tasks are taken into consideration and are co-scheduled with Sporadic Servers serving tasks with no offsets. In [17], the authors do not explicitly cope with unserved tasks. For the sake of comparison, we assign each unserved task to a server, whose budget, period, and priority are the same as those of the original task. Figure 5.5 (right) shows that, akin to previous subsets of experiments, their analysis predicts tasks with a much larger ratio (in the range of 25 to 58) than those obtained through our analysis or LITMUS^{RT}. In addition to that, their analysis forecasts that 80% of tasks present a $WCRT$ smaller than their period, whereas our proposed method and LITMUS^{RT} show that this percentage is close to 98%.

5.4 Summary

In this chapter, an exact response time analysis for tasks scheduled by fixed priority servers, namely Polling Periodic, extended Polling Periodic, Deferrable and Sporadic Server, was provided. While this analysis can be applied to either periodic or aperiodic tasks, in the following chapters tasks are assumed to be periodic, since the automotive applications requiring temporal isolation are considered to be mostly periodic. Furthermore, experiments on top of LITMUS^{RT} validated the tightness of the aforementioned analysis, proving that the proposed method outperforms existing ones.

Even though the overheads induced by the implementation of a given server algorithm can have a negative impact on the schedulability of a system, their dimensions are neglected by existing analyses. Thus, in the following chapter, the above-mentioned analysis will be extended in order to take overheads into account. Based on this extension, a practical server parametrization heuristic technique that preserves the least possible utilization as well as enhances the aggregated WCRT of the tasks in a hierarchical scheduling setting will be introduced.

CHAPTER 6
OVERHEAD-AWARE ANALYSIS

As mentioned in this previous chapter, while existing works consider overheads as negligible, their effect can have significant, and even adverse impact on the schedulability of a system. Thus, in this chapter, an overhead-aware method to assess the schedulability of real-time applications in a hierarchical fixed priority preemptive setting is presented. Moreover, actual overhead measurements are provided and their influence on system schedulability is exposed. It is also shown how existing sufficient, but not necessary, studies [17],[18], are inadequate for server parameter selection in hierarchical settings, and based on the proposed overhead model, a practical parameter selection heuristic that results in schedulable systems with low utilization and small aggregated WCRTs is derived. While the system model is akin to the one used in chapter 4 and 5, due to the practicability of this chapter and according to [4], tasks are considered to be periodic, served and present no offsets. Moreover, due to the limitation of the PPS exposed in Chapters 1 and 4, bandwidth-preserving fixed priority servers, such as the Deferrable and Sporadic Servers, are the main focus of this chapter.

6.1 Motivation

Even though the overheads induced by the implementation of a given server algorithm can have a negative impact on the schedulability of a system, previous analyses, either neglect their dimensions [11], [18], or provide simplistic approaches to account for them [17]. Moreover, regarding the optimal server parameter selection, studies using methods based on response time analysis that are only necessary but not sufficient, such as [17], can lead to wrong conclusions as shown in the next example. Consider the Deferrable Servers s_1 and s_2 (Table 7.1 [Right]), scheduling the tasks shown in Table 7.1 [Left]. Moreover, assume s_1 has the highest priority.

Task	C_i	T_i	D_i	*Server*
τ_1	10	20	20	s_1
τ_2	3	24	24	s_2

Server	C_s	T_s
s_1	11	20
s_2	4	24

Table 6.1 System parameters

In [17] the authors propose that the WCRT of τ_2, R_2, is to be obtained at its critical instant. According to [17], this scenario occurs when τ_2 and its server, s_2, are released at the same time, but the execution of the server is delayed due to interference from s_1. Additionally, similar to the assumptions made in [17], let us assume a server overhead of 1 time unit, whenever the server has to execute any of its tasks. Based on this scenario, R_2 is given by w, where w is obtained by solving $w = (C_{\tau_2} + 1) + \lceil (w + T_{s_1} - C_{s_1})/T_{s_1} \rceil \cdot C_{s_1} = 4 + \lceil (w + 9)/20 \rceil \cdot (11)$, and so $R_2 = w = 26$. A similar criteria is used to obtain the WCRT of τ_1, $R_1 = 11$. Hence the analysis claims that the system is not schedulable.

On the other hand, in [18] Shin and Lee introduce a compositional method to abstract an application composed of periodic tasks as a single periodic resource model $\Gamma(\Pi, \Theta)$ that describes a partitioned resource guaranteeing an allocation (budget) of Θ every Π time units. Moreover, if the resource period Π is given, the authors present a method to derive a lower bound on the budget Θ so that the timing requirements of the tasks assigned to this resource model are satisfied. For instance, if an application is composed of a task τ_2 and its corresponding resource model $\Gamma(\Pi_2, \Theta_2)$ presents a period of 24, i.e., $\Pi_2 = 24$, according to [18] the minimum budget, Θ_2^+, is computed by solving $\Theta^+ = (-(T_2 - 2 \cdot \Pi_2) + \sqrt{(T_2 - 2 \cdot \Pi_2)^2 + 8 \cdot \Pi_2 \cdot C_2})/4$. Hence, $\Theta_2^+ = 14.49$. Similarly, a resource model $\Gamma(\Pi_1, \Theta_1)$ that abstracts the timing requirements of τ_1 with $\Pi_1 = 20$ yields a minimum budget, Θ_1^+, of 16.18. These results indicate that in order for the servers to fulfill the timing requirements of their tasks, given $T_{s_1} = 20$ and $T_{s_2} = 24$, their budgets have to be at least $C_1 = 16.18$ and $C_2 = 14.49$. With these parameters, however, the total system utilization is 1.41, which implies that the analysis fails to guarantee the coexistence of the servers.

Now assume that given the task set of (Table 7.1 [Left])), the problem is to find the optimal set of server parameters that leads to the minimum processor utilization and does not jeopardize the schedulability of the system. In such scenario, an exhaustive parameter search in conjunction with a simulation of the actual execution of the tasks reveals that the (budget, period) parameter pairs leading to the lowest utilization for s_1 and s_2 are $(11, 20)$ and $(4, 24)$, resulting in a schedulable system with $R_1 = 11$ and $R_2 = 15$. Nevertheless, with these parameters [17] yields that the system is not schedulable, and [18] cannot guarantee the fulfillment of task deadlines.

This example and the experiments conducted in Section 6.5 illustrate that existing studies of the optimal server parameter selection for Hierarchical Fixed Priority Preemptive Systems (HFPPSs) are inconclusive and that such an investigation can only be built on an exact analysis that accounts for the induced overheads.

6.2 Overhead Model

To integrate overhead accounting into the presented schedulability analysis, overheads are modeled by inflating the demand, i.e. the execution requirement, of a task. Notice that this work deals with *context-switch* and *scheduling overhead.* While the former refers to the overhead due to task switching, the latter is the overhead associated to task selection. Overheads due to the execution of an interrupt (release and handling), as well as cache-related overheads are not considered in this thesis. Henceforth context-switch and scheduling overheads are referred to as overheads.

In a HFPPS, overheads manifest themselves when a job is preempted by a higher priority job. In such a case, the costs of a preemption are charged to the preempting higher priority job [53]. In this way, the execution requirement of each job is inflated at its release and on its completion by adding the maximum worst-case overhead, i.e. the maximum duration of the combination of a context-switch and scheduling overhead, in each case. This is a safe approximation as any task, except for the lowest-priority task in the system, could potentially cause a preemption to occur in a multilevel scheduling setting.

Another source of overhead is the preemption involved when a job uses up its server's budget, in which case the preempted job pays for the overhead, suggesting that the job has to be preempted before exhaustion of the budget [54]. Furthermore, recall that this job resumes its execution upon replenishment of its server's budget, and hence the induced overhead has to be considered too. In the next subsection, before modelling the induced overhead demand of a task τ_i as a window set $O_{\tau_i}(t)$, the release and budget exhaustion points of its server are grouped in two sets, M_s and L_s respectively.

6.2.1 Release and Budget Exhaustion Points

Intuitively, before analyzing the schedulability of a system taking account of the additional load due to the overheads, first the points in time when the scheduler is invoked should be calculated. For instance, consider the DS s ($C_s = 10$, $T_s = 20$) shown in Figure 6.1, with highest priority in the system, serving its only tasks τ_1 ($C_1 = 14$, $T_1 = 50$). In the figure, it can be seen that the budget is exhausted at $t = 10$ and $t = 60$, and that the first and second job of τ_1, whose arrivals are represented by upward arrows, resume their execution after s is reinvoked at $t = 20$ and $t = 60$ respectively.

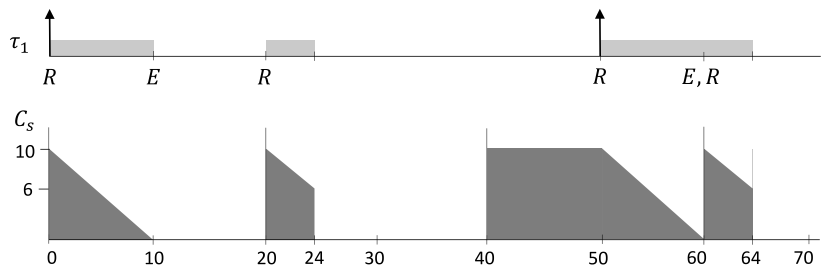


Figure 6.1 A DS $s(C_s = 10, T_s = 20)$ serving its task $\tau_1(C_1 = 14, T_1 = 50)$. Release and budget exhaustion points of the server are marked as R and E.

Algorithm 13 presents a method to calculate the points in time, when a server s is invoked and also when its budget depletes to zero. The algorithm starts by splitting the AEC, $\overline{\mathcal{C}}^e_s(t)$, into subcurves, so that each window lies within $[k \cdot T_s, (k+1) \cdot T_s\rangle$. Then for each subcurve, it searches for the server's release as well as budget exhaustion points and stores them in two sets, M_s and L_s respectively. Example 6.25 demonstrates, how Algorithm 13 derives those sets for the system shown in Figure 6.1.

Algorithm 13 Server's release and budget exhaustion points

```
1: Input: $\overline{\mathcal{C}}^e_s(t)$, $T_s$, $C_s$
2: $M_s = \emptyset$, $L_s = \emptyset$, $v = \lceil t/T_s \rceil - 1$
3: Compute $\tilde{\mathcal{C}}^e_s(t) = \bigcup_{k=0}^{v} \tilde{\mathcal{C}}_{s,k}(t) = split(\overline{\mathcal{C}}^e_s(t), T_s)$
4: for $k \leftarrow 0$ to $v$ do
5:     Get the first window in $\tilde{\mathcal{C}}_{s,k}(t)$, $W^{first}$
6:     $M_s = M_s \cup W^{first}.s$
7:     if $cap(\tilde{\mathcal{C}}_{s,k}(t)) = C_s$ then
8:         Get the last window in $\tilde{\mathcal{C}}_{s,k}(t)$, $W^{last}$
9:         $L_s = L_s \cup W^{last}.e$
10: Return $M_s$, $L_s$
```

Example 6.25. *Given a DS s ($T_s = 20$, $C_s = 10$), with Actual Execution Curve $\overline{\mathcal{C}}^e_s(t = 100) = \{(0, 10),$ $(20, 24), (50, 64)\}$, in order to compute the points in time when the server is released and when its budget is depleted to zero, Algorithm 13 performs the following steps:*

1. *Split the server's AEC, $\overline{\mathcal{C}}^e_s$, in intervals equal to the period of the server, T_s, thereby producing a set $\tilde{\mathcal{C}}^e_s(t) = \bigcup_{k=0}^{v} \tilde{\mathcal{C}}_{s,k}(t) = split(\mathcal{C}^e_s(t), T_s)$, with $v = \lceil t/T_s \rceil - 1$. In the example, $v = \lceil 100/20 \rceil - 1 = 4$, and $\tilde{\mathcal{C}}^e_s(t) = \{(0, 10)\} \cup \{(20, 24)\} \cup \{(50, 60)\} \cup \{(60, 64)\}$.*

2. *For each curve $\tilde{\mathcal{C}}_{s,k}(t)$, get its first window, W^{first}, and add the start time of the window, $W^{first}.s$, to M_s. In our case, all curves are composed of one window, and so their start times are added to M_s. Thus, $M_s = \{0, 20, 50, 60\}$.*

3. *For each curve $\tilde{\mathcal{C}}_{s,k}(t)$, provided that its total capacity, $cap(\tilde{\mathcal{C}}_{s,k}(t))$, is equal to that of the server, add the end time of its last window, $W^{last}.e$, to the set containing the server's exhaustion points, L_s. In the example, $\tilde{\mathcal{C}}_{s,0}(t) = \{(0, 10)\}$ and $\tilde{\mathcal{C}}_{s,2}(t) = \{(50, 60)\}$ present a total capacity equal to $C_s = 10$, and so the end times of each window are added to L_s. Thus, $L_s = \{10, 60\}$.*

Theorem 12. *Algorithm 13 provides the preemption points due to release and budget exhaustion of a server up to time t.*

Proof. The theorem is proven by showing that the following loop invariant holds. **Loop invariant**: At the beginning of each iteration with index $j \in [0, v]$, M_s and L_s contain the server's release and budget exhaustion points up to time $j \cdot T_s$ respectively. **Initialization**: At the start of the first loop the two sets should contain the release and budget exhaustion points of the server up to $0 \cdot T_s = 0$, i.e. $M_s = L_s = \emptyset$ and this is what the sets have to be set to. **Maintenance**: Assume that the loop invariant holds at the start of iteration j. In the body of the loop and for the interval $[k \cdot T_s, (k + 1) \cdot T_s\rangle$, we add the point in time where the server is released, i.e. the first window W^{first} in $\tilde{\mathcal{C}}_{s,k}(t)$, to M_s, and the point in time where the server's budget is depleted, if any, i.e. the last window W^{last} in $\tilde{\mathcal{C}}_{s,k}(t)$, provided that $cap(\tilde{\mathcal{C}}_{s,k}(t)) = C_s$. Thus, at the start of iteration $(j+1)$, M_s and L_s present the server's release and budget exhaustion points up to $(j+1) \cdot T_s$ respectively, which is what needed to be proven. **Termination**: When the for-loop terminates, according to the loop invariant M_s and L_s contain the release and budget exhaustion points of the server up to $(v+1)T_s = (\lceil t/T_s \rceil - 1 + 1)T_s = (\lceil t/T_s \rceil)T_s \geq t$. However, since $\overline{\mathcal{C}}^e_s(t)$ denotes the demand served by the server up to time t, then $\tilde{\mathcal{C}}_{s,v}(t)$ represents the served demand within $[v \cdot T_s, t]$. Hence M_s and L_s contain the release and budget exhaustion points of the server up to time t. □

6.2.2 Overhead Demand Windows

Once the release and budget exhaustion points of a server associated with a task τ_i are obtained, the extra load originated from the overheads can be calculated by finding the scheduler invocation points during the execution of its jobs. Let C^o be the maximum worst-case overhead, i.e. the sum of the maximum context-switch and scheduling overhead, then Algorithm 14 proposes a procedure for computing the overhead window set, $O_{\tau_i}(t)$, that represents the extra demand that derives from the overhead related to the execution of the jobs released by τ_i.

Algorithm 14 Total overhead

```
1: Input: $\overline{\mathcal{C}}^e_{\tau_i}(t)$, $M_s$, $L_s$, $C^o$, job demand set $\{(a_{i,j}, a_{i,j} + c_{i,j})\}$
2: $f_{i,0} = 0$, $O_{\tau_i}(t) = \emptyset$
3: for every job $J_{i,j}$ do
4:     Given $a_{i,j}$ and $c_{i,j}$, get the WCRT, $R_{i,j}$
5:     Compute the finishing time $f_{i,j} = a_{i,j} + R_{i,j}$
6:     $O_{\tau_i}(t) = O_{\tau_i}(t) \cup \{(f_{i,j} - C^o, f_{i,j})\}$
7:     $O_{\tau_i}(t) = O_{\tau_i}(t) \cup \{(a_{i,j}, a_{i,j} + C^o)\}$
8: for every element $t'$ in $M_s$ do
9:     if $\exists W' \in \overline{\mathcal{C}}^e_{\tau_i}(t)$ s.t. $W'.s \leq t' < W'.e$ then
10:        $O_{\tau_i}(t) = O_{\tau_i}(t) \cup \{t', t' + C^o\}$
11: for every element $t''$ in $L_s$ do
12:     if $\exists W'' \in \overline{\mathcal{C}}^e_{\tau_i}(t)$ s.t. $W''.s < t'' \leq W''.e$ then
13:        $O_{\tau_i}(t) = O_{\tau_i}(t) \cup \{t'' - C^o, t''\}$
14: Sort the windows of $O_{\tau_i}(t)$ according to their start times
15: Return $O_{\tau_i}(t)$
```

The algorithm starts by calculating the response time, $R_{i,j}$, of each job $J_{i,j}$ and computing its finishing time, $f_{i,j}$, as the sum of its arrival time and response time, i.e. $f_{i,j} = a_{i,j} + R_{i,j}$. Next, a window $\{(a_{i,j}, a_{i,j} + C^o)\}$ and another window $\{(f_{i,j} - C^o, f_{i,j})\}$, representing the overhead due to the release and completion of the job respectively, are added to $O_{\tau_i}(t)$. Finally, the windows representing the scheduler demand as a result of the resumption of a job after a server release as well as due to the server's budget exhaustion during the execution of τ_i are added to $O_{\tau_i}(t)$. Example 6.26 demonstrates how Algorithm 14 groups the overhead of

τ_1 (see Figure 6.1) in $O_{\tau_1}(t)$.

Example 6.26. *Given the AEC of task* τ_1, $\overline{\mathcal{C}}^e_{\tau_1}(t = 100) = \{(0, 10), (20, 24), (50, 64)\}$, *job window demand set in* $[0, 100]$, $\{(a_{1,1}, a_{1,1} + c_{1,1}), (a_{1,2}, a_{1,2} + c_{1,2})\} = \{(0, 14), (50, 64)\}$, $C^o = 1$, $M_s = \{0, 20, 50, 60\}$, *and* $L_s = \{10, 60\}$, *in order to compute the scheduler overheads Algorithm 14 follows the next steps:*

1. *For each job,* $J_{i,j}$, *obtain its response time,* $R_{i,j}$. *By following the steps described in the previous chapter, we get* $R_{1,1} = 24$, *and* $R_{1,2} = 14$ *as illustrated by Figure 6.1.*

2. *Get the finishing time of each job,* $f_{i,j}$, *as the sum of its arrival time and its response time, and add a window* $\{(f_{i,j} - C^o, f_{i,j})\}$ *to the overhead window set* $O_{\tau_i}(t)$. *In our case,* $f_{1,1} = a_{1,1} + R_{1,1} = 0 + 24 = 24$, *and* $f_{1,2} = a_{1,2} + R_{1,2} = 50 + 14 = 64$. *Thence,* $\{(24 - 1, 24)\} = \{(23, 24)\}$ *and* $\{(63, 64)\}$ *are added to* $O_{\tau_i}(t)$.

3. *Next, add a window* $\{(a_{i,j}, a_{i,j} + C^o)\}$ *to* $O_{\tau_i}(t)$. *In the example,* $\{(a_{1,1}, a_{1,1} + C^o)\} = \{(0, 1)\}$ *and* $\{(a_{1,2}, a_{1,2} + C^o)\} = \{(50, 51)\}$, *and so* $O_{\tau_i}(t) = \{(23, 24), (63, 64), (0, 1), (50, 51)\}$.

4. *If* t' *represents an element of* M_s, *add a window* $\{(t', t' + C^o)\}$ *to* $O_{\tau_i}(t)$ *provided that* t' *lies within the execution of* τ_i. *In our case* $\{(0, 1)\}$, $\{(20, 21)\}$, $\{(50, 51)\}$, *and* $\{(60, 61)\}$ *are added to* $O_{\tau_i}(t)$, *i.e.* $O_{\tau_i}(t) = \{(23, 24), (63, 64), (0, 1), (50, 51), (20, 21), (60, 61)\}$.

5. *If* t'' *represents an element of* L_s, *add a window* $\{(t'' - C^o, t'')\}$ *to* $O_{\tau_i}(t)$ *provided that* t'' *lies within the execution of* τ_i. *In our case* $\{(9, 10)\}$ *and* $\{(59, 60)\}$ *are added to* $O_{\tau_i}(t)$. *Thus,* $O_{\tau_i}(t) = \{(23, 24), (63, 64), (0, 1), (50, 51), (20, 21), (60, 61), (9, 10), (59, 60)\}$.

6. *Finally, sort the windows of* $O_{\tau_i}(t)$ *according to their start times. In this way,* $O_{\tau_i}(t) = \{(0, 1), (9, 10), (20, 21), (23, 24), (50, 51), (59, 60), (60, 61), (63, 64)\}$.

Theorem 13. *Given a task* τ_i *scheduled by a server* s, *Algorithm 14 returns a set* $O_{\tau_i}(t)$ *containing the additional overhead demand originated due to the execution of its jobs up to time* t.

Proof. Let us prove the theorem by showing that $O_{\tau_i}(t)$ contains the extra overhead due to the release and budget exhaustion of s, as well as that due to the release and completion of the first $j - th$ jobs of τ_i, where j represents the number of jobs executed up to time t. From Theorem 12, it is known that M_s (resp. L_s) represents the release (resp. budget exhaustion) points of s up to time t. Hence, for any point in time $t' \in M_s$ (resp. $t'' \in L_s$), if there exists one window W' (resp. W'') $\in \overline{\mathcal{C}}^e_{\tau_i}(t)$ so that $W'.s \leq t' < W'.e$ (resp. $W''.s < t' \leq W''.e$), then t' (resp. t'') is a release (resp. budget exhaustion) point that coincides with the execution of a job of τ_i. Thus, the extra demand originated from the release (resp. budget exhaustion) of s, denoted by a window of the form $\{t', t' + C^o\}$ (resp. $\{t'' - C^o, t''\}$), is added to $O_{\tau_i}(t)$, which proves the first part of the theorem. Next, let us continue with the proof by showing that the following loop invariant holds. **Loop invariant:** At the start of each iteration with index j of the first for-loop, $O_{\tau_i}(t)$ contains the extra demand due to the release and completion of all the jobs preceding the $j-th$ job of τ_i, $J_{i,j}$. **Initialization:** At the start of the first loop, τ_i has not released any jobs and so there is no extra demand, i.e. $O_{\tau_i}(t) = \emptyset$. **Maintenance:** Assume that the loop invariant holds at the start of iteration j. In the body of the loop a window in the form of $\{(a_{i,j}, a_{i,j} + C^o)\}$ (resp. $\{(f_{i,j} - C^o, f_{i,j})\}$) is added to $O_{\tau_i}(t)$, which represents the extra demand due to the release (resp. completion) of $J_{i,j}$. Thus, at the start of iteration $(j+1)$, $O_{\tau_i}(t)$ is composed of the extra demand due to the release and completion of the jobs preceding $J_{i,j+1}$, which is what needed to be proven. **Termination**: When the for-loop terminates, according to the loop invariant, $O_{\tau_i}(t)$ contains the extra demand due to the release and completion of all the jobs of τ_i executed in t, which concludes the proof of the theorem. □

6.3 Overhead-Aware Analysis

In any real system, job execution is slowed down by various overheads. In the real-time literature such overheads are usually considered to be negligible, and yet they can have a significant effect on the performance of a scheduling algorithm. Thus, Algorithm 15 shows a method to determine the WCRT of tasks in a HFPPS taking their overhead into consideration.

Algorithm 15 Response time computation with overheads

```
1: Input: $C^o$, tasks and servers' parameters
2: Compute the demand of each task, $\mathcal{C}^d_{\tau_i}(t)$, in a hyperperiod
3: for each server s (in priority order) do
4:     for each task $\tau_i$ of s (in priority order) do
5:         Initialize the temporary curve $\overline{\mathcal{C}}^{e*}_s(t) = \emptyset$
6:         Calculate $\overline{\mathcal{C}}^e_s(t)$
7:         while $\overline{\mathcal{C}}^{e*}_s(t) \neq \overline{\mathcal{C}}^e_s(t)$ do
8:             $\overline{\mathcal{C}}^{e*}_s(t) = \overline{\mathcal{C}}^e_s(t)$
9:             Compute $M_s$ and $L_s$ as per Algorithm 13
10:            Get the Actual Execution Curve of $\tau_i$, $\overline{\mathcal{C}}^e_{\tau_i}(t)$
11:            Obtain $O_{\tau_i}(t)$ as per Algorithm 14
12:            Recalculate $\overline{\mathcal{C}}^e_s(t)$ with $\mathcal{C}^d_{\tau_i}(t) \oplus O_{\tau_i}(t)$
13:        for every job $J_{i,j}$ do
14:            Get $R_{i,j}$
```

The algorithm begins by computing the demand in a hyperperiod of the tasks composing the system, thereby calculating the AEC of the highest priority server. Next, it determines the AEC of the highest priority task in the server and the overhead related to its execution by means of Algorithm 14. This additional overhead demand is added to its initial demand so that a new AEC of its server is determined.

This process is repeated until the recalculation of the server's AEC results in the same curve. After that the algorithm proceeds to repeat the same calculations for all the tasks of the server, as well as all the servers in the systems on a priority basis. Observe that the algorithm performs its steps in priority order as the response time analysis of a task is only possible after convergence of the AECs of its higher priority tasks. Example 6.27 shows how Algorithm 15 obtains the response time of the jobs released by the tasks in the system described in Table 6.2.

Server	C_s	T_s
S_1	10	20
S_2	20	50

Task	C_i	T_i	*Server*
τ_c	10	50	S_1
τ_b	20	100	S_2
τ_a	20	250	S_2

Table 6.2 System parameters

Example 6.27. *Consider the DSs, S_1 and S_2 shown in Table 6.2 (Left) serving the tasks in Table 6.2 (Right), and assume that $C^o = 1$. Notice that in the example, priorities of tasks and servers are assigned according to the Rate-Monotonic policy. In order to compute the WCRT of the tasks in the system, Algorithm 15 performs the next steps:*

1. *Get the hyperperiod of the system as $LCM(T_a, T_b, T_c, T_{s_1}, T_{s_2}) = LCM(250, 100, 50, , 20, 50) = 500$. Hence, $t = 500$.*

2. *Obtain the demand of each task for a hyperperiod. In the example, $\mathcal{C}^d_{\tau_a}(t) = \{(0, 20), (250, 270)\}$, $\mathcal{C}^d_{\tau_b}(t) = \{(0, 20), (100, 120), (200, 220), (300, 320), (400, 420)\}$, and $\mathcal{C}^d_{\tau_c}(t) = \{(0, 10), (50, 60), (100, 110), (150, 160), (200, 210), (250, 260), (300, 310), (350, 360), (400, 410), (450, 460)\}$.*

3. *Get the AEC of the server with highest priority. In the example, s_1 is that server and so $\overline{\mathcal{C}}^e_{s_1}(t) = \{ (0, 10), (50, 60), (100, 110), (150, 160), (200, 210), (250, 260), (300, 310), (350, 360), (400, 410), (450, 460)\}$.*

4. *Store the AEC of this server, $\overline{\mathcal{C}}^e_{s_1}(t)$, in a temporary curve, $\overline{\mathcal{C}}^{e*}_{s_1}(t)$.*

5. *Get the release and exhaustion point sets of the server. Algorithm 13 yields $M_{s_1} = \{0, 50, 100, 150, 200, 250, 300, 350, 400, 450\}$, and $L_{s_1} = \{10, 60, 110, 160, 210, 260, 310, 360, 410, 460\}$.*

6. *Calculate the AEC, of the task with the highest priority in the server. In the example that task is τ_c and so $\overline{\mathcal{C}}^e_{\tau_c}(t) = \{(0, 10), (50, 60), (100, 110), (150, 160), (200, 210), (250, 260), (300, 310), (350, 360), (400, 410), (450, 460)\}$.*

7. *Compute the overhead window set, $O_{\tau_i}(t)$, of the task under analysis according to Algorithm 14 and add (Definition 4.6) $O_{\tau_i}(t)$ to the original task demand. In the example, $O_{\tau_c}(t) = \{(0, 1), (9, 10), (50, 51),$*

$(59, 60), (100, 101), (109, 110), (150, 151), (159, 160), (200, 201), (209, 210), (250, 251), (259, 260), (300, 301), (309, 310), (350, 351), (359, 360), (400, 401), (409, 410), (450, 451), (459, 460)\}$, *and so* $\mathcal{C}^d_{\tau_c}(t) \oplus O_{\tau_c}(t) = \{(0, 12), (50, 62), (100, 112), (150, 162), (200, 212), (250, 262), (300, 312), (350, 362), (400, 412), (450, 462)\}$.

8. *With* $\mathcal{C}^d_{\tau_c}(t) \oplus O_{\tau_c}(t)$, *recalculate the server's AEC. In our case, the new* $\mathcal{C}^d_{\tau_c}(t)$ *yields* $\overline{\mathcal{C}}^e_{s_1}(t) = \{(0, 10), (20, 22), (50, 62), (100, 110), (120, 122), (150, 162), (200, 210), (220, 222), (250, 262), (300, 310), (320, 322), (350, 362), (400, 410), (420, 422), (450, 462)\}$.

9. *As long as the current server's AEC,* $\overline{\mathcal{C}}^e_s(t)$, *differs from the previous one stored in the temporary curve,* $\overline{\mathcal{C}}^{e*}_s(t)$, *overwrite the latter with* $\overline{\mathcal{C}}^e_s(t)$ *and compute a new* $\overline{\mathcal{C}}^e_s(t)$ *by repeating steps 4 - 8. In the example, as it can be seen that the current AEC,* $\overline{\mathcal{C}}^e_{s_1}(t)$, *and the previous one,* $\overline{\mathcal{C}}^{e*}_{s_1}(t)$, *are different, a new server's AEC is calculated. This new curve* $\overline{\mathcal{C}}^e_{s_1}(t) = \{(0, 10), (20, 24), (50, 64), (100, 110), (120, 124), (150, 164), (200, 210), (220, 224), (250, 264), (300, 310), (320, 324), (350, 364), (400, 410), (420, 424), (450, 464)\}$ *also differs from the previous server's AEC and hence* $\overline{\mathcal{C}}^e_{s_1}(t)$ *is recalculated. Since this recalculation yields the same curve, i.e* $\overline{\mathcal{C}}^{e*}_s(t) = \overline{\mathcal{C}}^e_s(t)$, *the iteration is over.*

10. *For each job,* $J_{i,j}$, *obtain its response time,* $R_{i,j}$. *In the example* $R_{c,1} = R_{c,3} = R_{c,5} = R_{c,7} = R_{c,9} = 24$, *and* $R_{c,2} = R_{c,4} = R_{c,6} = R_{c,8} = R_{c,10} = 14$.

11. *Repeat steps 3-10, for each task and each server in the system in order of priority. In our case,* s_1 *presents no more tasks and hence we continue to analyze* s_2, *starting with* τ_b, *which results in* $R_{b,1} = R_{b,2} = R_{b,3} = R_{b,4} = R_{b,5} = 68$. *Next, let us consider the next task in* s_2, τ_a, *and so* $R_{a,1} = 175$ *and* $R_{a,2} = 125$.

Once, the response times are known, the WCRT of each task can be determined by taking the largest response time obtained by the previous algorithm. Thus, $R_a = 175$, $R_b = 68$, and $R_c = 24$, and so the system is schedulable. Figure 6.2 shows the actual execution of the jobs released by the tasks in the example, after their demand was inflated on account of the overheads as shown by the dark rectangles. The WCRT of each task can also be seen in the inset.

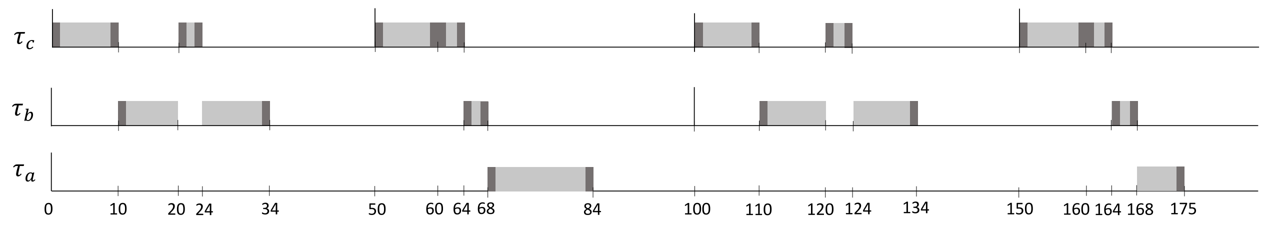


Figure 6.2 Gantt chart of the task set of Table 6.2. Dark grey segments represent the overheads

Theorem 14. *The WCRT of a task, taking overheads into consideration, is given by the largest response time obtained by means of Algorithm 15.*

Proof. Since Algorithm 14 is structured as nested loops, we prove the theorem by showing that the following loop invariants hold. **While-loop invariant:** *At the start of each iteration, $\overline{\mathcal{C}}_s^e(t)$ represents the AEC of a server s, when the original demand of τ_i is inflated due to the overheads induced by its execution and the previous AEC, $\overline{\mathcal{C}}_s^{e*}(t)$.* The invariant is trivially true at the start of the first loop as there are no induced overheads. Assume the invariant is true at the beginning of an iteration, in the body of the loop we obtain the release and budget exhaustion points of the given $\overline{\mathcal{C}}_s^e(t)$ (Theorem 12) in a hyperperiod. With these points and the AEC of τ_1, we get the additional overhead demand $O_{\tau_i}(t)$ induced by the jobs executed up to the hyperperiod (Theorem 13). As the AEC of the server is stored in $\overline{\mathcal{C}}_s^{e*}(t)$, a new AEC $\overline{\mathcal{C}}_s^e(t)$, computed after adding $O_{\tau_i}(t)$ to the original task demand, represents the AEC of the server due to the release and completion of τ_i as well as the release and budget exhaustion given by $\overline{\mathcal{C}}_s^{e*}(t)$, which is what needed to be proven. Eventually, the while-loop terminates, when $\overline{\mathcal{C}}_s^e(t)$ converges, which aids proving the next invariant. **Inner for-loop invariant:** *At the start of each iteration, the AEC of the server, $\overline{\mathcal{C}}_s^e(t)$, denotes the AEC that considers the overheads due to the execution of the tasks with priorities higher than that of the task under analysis τ_i.* The invariant is trivially true at the start of the first loop as there are no higher priority tasks. Assume that the invariant is true at the beginning of an iteration, the *while-loop invariant* assures us that at the next iteration $\overline{\mathcal{C}}_s^e(t)$ takes the overhead due to the execution of τ_i into consideration, which shows that for the next task the overheads due to its higher priority tasks are considered. Upon termination of the for-loop, the response time of the jobs released by the tasks scheduled by the server s include their induced overheads. **Outer for-loop invariant:** *At the start of each iteration, the AEC of the server s, $\overline{\mathcal{C}}_s^e(t)$, denotes the AEC that considers the overheads due to the execution of the tasks served by its higher priority servers.* The invariant is trivially true at the start of the first loop as there are no higher priority servers. Assume that the invariant is true at the beginning of an iteration, once the inner for-loop is terminated, the previous invariant guarantees that the AEC of the server s includes the overheads induced by the execution of all its tasks. Thus, at the next iteration, the AEC of the server under analysis includes the overhead of all the tasks scheduled by its higher priority servers. When the outer for-loop terminates, the response time of the jobs, released by a task τ_i and allocated to a server s, take account of the overhead induced by the task itself and all the higher priority tasks served by s, as well as all the tasks scheduled by all higher priority servers. Hence, the WCRT of τ_i is given by the largest job response time, which concludes the proof of the theorem. □

6.4 Server Parameter Selection

Automotive applications are particularly concerned with optimizing end-to-end propagation latencies of input events that trigger a chain of computations, leading to a final actuation or control action [55]. As this optimization involves reducing the aggregated worst-case response time of the tasks, i.e. the sum of their WCRTs, in a given system [56], the goal is to provide an optimal set of server parameters that leads to a schedulable system with minimal processor utilization as well as small aggregated WCRT.

Based on the overhead model presented in this chapter, the following conclusions can be drawn.

Theorem 15. *A lower bound on the cost charged to each job of a task due to the induced overheads is $2C^o$, where C^o is the maximum worst-case overhead.*

Proof. The execution requirement of any job is inflated by $2C^o$ due to its own release and completion. Any extra preemption due to reinvocation or exhaustion of the budget of its associated server further increases the demand of the job. □

Theorem 16. *A lower bound on the utilization of a server s, U_s, is given by $\sum(C_i + 2C^o)/T_i$ $\forall$ task τ_i scheduled by s.*

Proof. Given a task τ_j, from Theorem 15 we can conclude that after inflating the demand of each of its jobs by at least $2C^o$, $U_j \geq (C_j + 2C^o)/T_j$. Since the utilization of a server s, U_s, should be at least equal to the aggregated utilization of its tasks, then $U_s \geq \sum U_i \geq \sum(C_i + 2C^o)/T_i$ $\forall$ task τ_i in s. □

Theorem 17. *If for each server s in a given system, its period, T_s, is the LCM of its associated tasks, i.e. $T_s = LCM\{T_i\}$ $\forall$ task τ_i in s, and its budget, C_s, is equal to $T_s \cdot \sum(C_i + 2C^o)/T_i$ $\forall$ task τ_i in s, and the system is schedulable, then the server parametrization is optimal in terms of utilization.*

Proof. If the period of a server s, T_s, is the LCM of its served tasks, its budget, C_s, is $T_s \cdot \sum(C_i + 2C^o)/T_i$ $\forall$ task τ_i in s, and all its tasks meet their deadlines, then according to Theorem 16 the utilization of the server is the least possible. Thus, provided that all tasks in the system meet their deadlines, when all servers follow the aforementioned parametrization, this method leads to a schedulable system with minimal processor utilization.

□

By deploying the above-mentioned server parameter selection, the execution demand of each job of a given task is only inflated by $2C^o$, since there is no overhead induced due to an extra release or budget exhaustion of its server. As shown in the following section, this parametrization indeed produces schedulable systems with minimum utilization and small aggregated WCRT.

6.5 Evaluation

Having established an overhead model for HFPPSs, and its corresponding response time analysis, an experimental study of the server parameter selection is hereafter provided. For each of the next experiments, where synthetic tasks are generated, since each task set was run for 30 seconds, task and server periods are constrained to ensure that the hyperperiod does not exceed this time.

6.5.1 Analysis methods without overheads

In order to assess how well existing methods ([18], [17]) and the presented analysis deal with the problem of finding the optimal set of server parameters, an exhaustive search based on these three analysis is carried out. To that end, consider the system described in Table 6.3, where the DS s_1 has a higher priority than that of the DS s_2. For the sake of comparison, we conduct a brute-force search, where server periods are to be chosen in the range of 2 to the LCM of its constituent tasks, i.e. T_{s_1} and T_{s_2} lie within $[2, 10]$ and $[2, 50]$ respectively. Hence, given a server period T_s, its corresponding server capacity is found in $[1, T_s - 1]$, and if T_s^{max} represents the maximum value of T_s, then the size of the search space is $\prod((T_s^{max} \cdot (T_s^{max} - 1))/2)$ $\forall$ s in the system.

Task	C_i	T_i	*Task Priority*	*Server*
τ_1	1	10	1	S_1
τ_2	1	10	2	S_1
τ_3	2	10	3	S_2
τ_4	2	25	4	S_2

Table 6.3 System parameters

In the example, the size of the search space is $((10 \cdot 9)/2) \cdot ((50 \cdot 49)/2) = 55125$. This number of possible solutions can still be further reduced by considering that the server utilization should be at least equal to the sum of those of its constituent tasks, i.e. $U_{s_1} \geq 1/10 + 1/10 = 0.2$ and $U_{s_2} \geq 2/10 + 2/25 = 0.28$. Additionally, the system utilization should be at most 1, i.e. $U_{s_1} + U_{s_2} \leq 1$. With this constraint, the number of combinations is reduced to 10386.

In Figure 6.3 each dot represents the aggregated WCRT, with the horizontal and vertical position showing the period and budget of s_2. Moreover, the aggregation is encoded by color, where a smaller value is reflected in a lighter color. As the aggregated WCRT depends not only on the parameters of the lowest priority server but also on those of s_1, when the same pair of parameters of s_2 yields different results, their encoded colors are superimposed. The inset only displays pairs that produce a schedulable system. Furthermore, the figure shows a comparison between results obtained by means of the analysis presented in Chapter 5 (Left) against those through the method in [17] (Right).

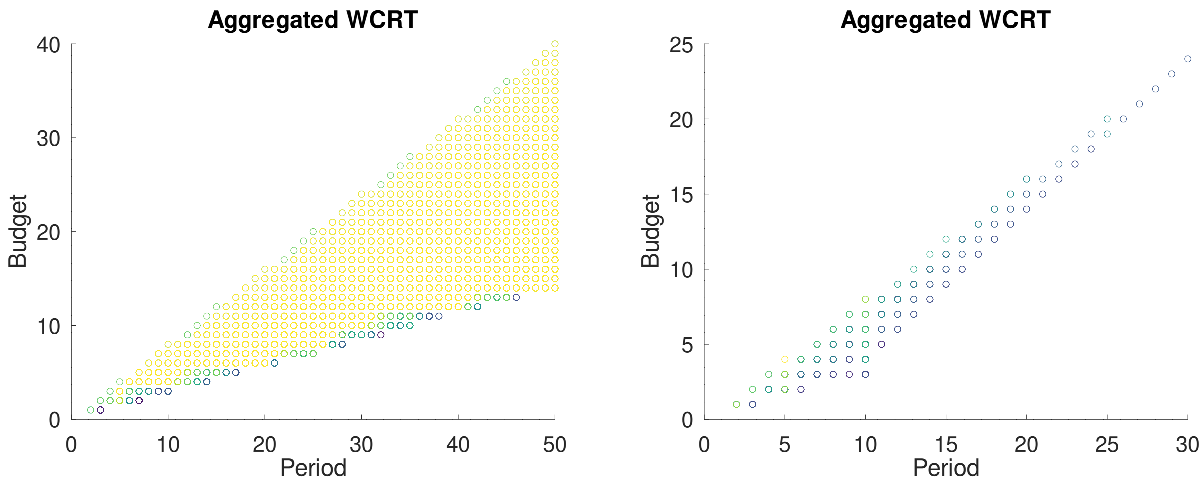


Figure 6.3 Exhaustive search comparison for s_2 by the method presented in Chapter 5 (Left) [17] (Right).

As expected, the scatter plots show a positive correlation between the budget and the period for both analysis, as increasing the period results in a proportional increment in the budget. Moreover, since the analysis proposed in [17] is based on a critical instant that does not necessarily occur, this pessimism manifests itself in the lower number of schedulable systems as shown by the number of dots in the figure, as well as the fact that all obtained parameters are a subset of the ones produced by the analysis presented in Chapter 5. Indeed, while use of the proposed method produces 10227 schedulable systems, deployment of [17] only makes for 364, i.e. 3.56%, of the actual number.

Furthermore, let the tuple (C_s, T_s) represent the parameters of a server s, then the optimal set of parameters given by the method exposed in Chapter 5 is $(2, 10)$ and $(14, 50)$ for s_1 and s_2 (as depicted in the inset), respectively, with a CPU utilization of 0.48 and an aggregation of 13. The aggregated WCRT and the utilization are the smallest possible numbers in the experiment. Nevertheless, with these parameters [17] claims that the system is not schedulable. It even suggests $(1, 5)$ and $(3, 10)$ as optimal parameters for s_1 and s_2 (refer to the figure), respectively, with a system utilization of 0.5 and aggregation of 33, provided that minimizing the utilization prevails over a minimization of the aggregated WCRT. Otherwise, it proposes $(1, 5)$ and $(4, 5)$ as optimal parameters for the highest and lowest (as shown in the picture) priority servers, respectively, with a system utilization of 1 and aggregation of 17. For the set of parameters of the first and second case, the analysis proposed in Chapter 5 provides an aggregated WCRT of 24 and 15, respectively, confirming the pessimism of [17].

With regard to [18], for every server s and for every possible value of its period T_s, the minimum resource allocation $\Theta^+_{s,i}$ of a task τ_i in s is derived as done in the beginning of this chapter, so that the budget of the server is computed as $C_s = \Theta_s^+$, where Θ_s^+ is the maximum $\Theta_{s,i}^+$ in the server. Note that the resulting capacity is a real number, and so we round it up. Additionally, when calculating the budget of a server, this method assumes the maximum interference from higher priority entities, and presumes that the budget of the server is guaranteed, thus the capacity of the server can be computed without knowing the parameters of others servers in the system.

In the example, this results in 12 and 21 admissible pairs for s_1 and s_2 respectively. However, not all the possible combinations are permitted since many combinations result in a utilization higher than 1. Eventually, this approach results in 15 parameter sets, all of them leading to schedulable systems, accounting for 0.15% of the sets found by following the method presented in Chapter 5. While the minimum utilization out of this 15 sets is 0.83 with an aggregated WCRT of 16, the least possible aggregated WCRT found is 15.

In conclusion, it is clear that the analysis presented in [17] and [18] are not suited for a thorough study of the server parameter selection problem for HFPPSs. Nonetheless, while the biggest advantage of the former is its simplicity to compute upper bounds on the response time of tasks, the greatest advantage of the latter is that regardless of the type of server being used, it provides a way to individually calculate the capacity of a server that fulfills the timing requirements of all its tasks, provided that its period is known.

6.5.2 Effect of overheads

As existing methods [17], [18] do not consider the effects of overheads in their analysis, the preceding comparison was made ignoring their effect. Thus, in order to present a more realistic study, the next experiment aims at showing the effects of an overhead-aware exact analysis. For this reason, consider the system of Table 6.3 once more and assume that the maximum worst-case overhead is 0.1, i.e. $C^o = 0.1$. Conducting the same exhaustive search described in the previous experiment produces the results shown in Figure 6.4.

First of all, Figure 6.4 (Left) makes evident that the number of feasible parameter sets decreases. In fact, it drops to 5999, i.e. the effect of overheads leads to a 58.66% reduction in the number of schedulable systems in comparison to that obtained by an overhead-agnostic schedulability analysis. It can also be seen that the amount of parameter sets resulting in a large aggregated WCRT increases as indicated by the darker color of the dots. Recall that the aggregated WCRT is color encoded and different values for the same pair of parameters of s_2 are superimposed, which can be better appreciated in the corresponding 3D scatter plot (Figure 6.4 (Right)).

Table 6.4 displays the top-5 parameter sets that lead to the minimal aggregated WCRT of 13 and 15 for the overhead-agnostic (Left) and overhead-aware (Right) analysis, respectively. In both cases it is evident that a server period equal to the LCM of its task periods is the optimum for a low system utilization as well

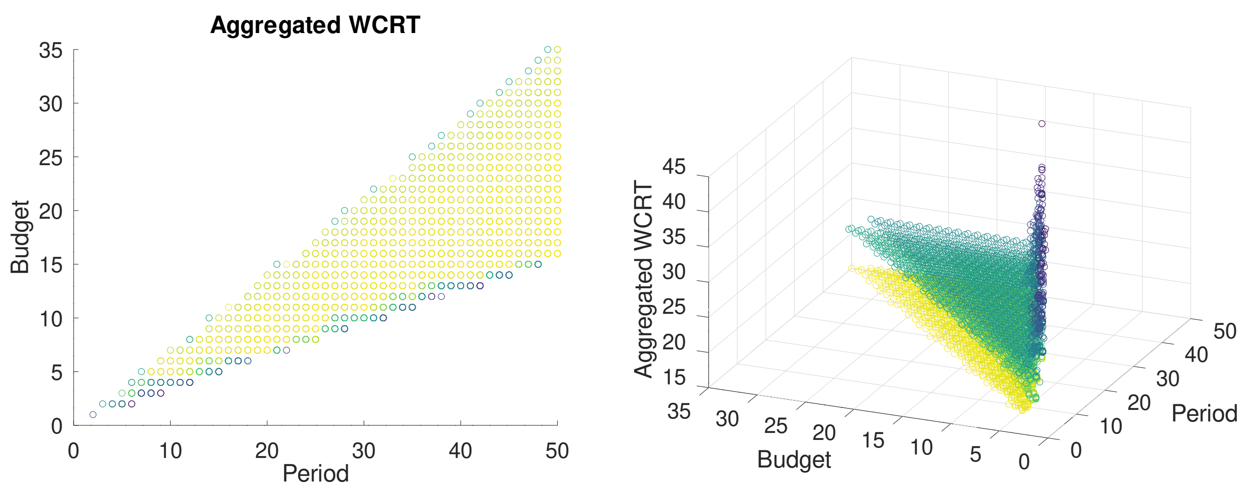


Figure 6.4 Exhaustive search for s_2 with overheads

as a low aggregated WCRT. As highlighted by the charts, in order to satisfy the extra overhead demand, a server's budget must increase, and as expected, this additional demand leads to a rise of the aggregated WCRT.

Based on the above results, it is apparent that the larger the overheads are, the larger the WCRTs become. Hence, for the purpose of giving realistic overhead dimensions, the next experiment aims at measuring the overheads of a real platform.

C_1	T_1	C_2	T_2	$\overline{U}$
2	10	14	50	0.48
2	10	14	49	0.486
2	10	14	48	0.492
2	10	14	47	0.498
2	10	6	20	0.5

C_1	T_1	C_2	T_2	$\overline{U}$
3	10	16	50	0.62
3	10	17	50	0.64
3	10	7	20	0.64
3	10	14	40	0.65
3	10	9	25	0.66

Table 6.4 Top-5 for overhead-agnostic (left) and -aware (right) analysis

6.5.3 Measurement of overheads

As the magnitude of overheads is impossible to anticipate in the absence of an actual, working implementation, the overheads associated to two different kinds of fixed priority servers, namely the DS and SS, are evaluated on top of a quad-core Intel i5-7200U processor @ 2.5GHz using 8GB of RAM running LITMUS^{RT} [57]. LITMUS^{RT} is a real-time extension of the Linux kernel with a focus on multiprocessor real-time scheduling and synchronization that also provides implementation of different servers.

In order to measure the overheads related to a given real-time scheduler, LITMUS^{RT} provides a low-overhead tracing toolkit called *Feather-Trace* that is based on static instrumentation of the kernel[23]. If n represents the number of tasks composing a system, with the objective to measure maximum and average overheads under different settings, sets ranging from $n = 3$ to $n = 10$, where tasks are uniformly sampled from $\{100, 150, 200, 250, 300\}$ms, and randomly assigned to servers are traced. Task utilizations are generated as per the *UUnifast* algorithm [52] with a total system utilization of 0.65. Server and task priorities are randomly assigned. The number of servers ranges from 1 to 7 and the number of task sets[24] for each n is at least 100. Given a server s, its period, T_s, is computed as the LCM of its associated tasks, and its budget, C_s, is calculated as $C_s = 1.2 \cdot T_s\overline{U}_s$, where 1.2 accounts for the overheads.

Tracing tasks sets of different sizes and with different tasks parameters results in traces with a variable number of overhead samples. Thus, in order to obtain an unbiased estimator for the worst-case overhead, all traces corresponding to a given n under a given server are collected and shuffled, prior to being truncated to the length of the shortest observed trace of its kind [57]. In this way, 32839 and 33349 samples are obtained for scheduling and context-switch overheads, respectively.

Figure 6.5 shows worst-case context-switch (Left) and scheduling overhead (Right) as a function of n. Three trends are apparent. First, worst-case overheads tend to be higher under the SS than under the DS, with some exceptions, which might be due to the variance inherent in empirical measurement. This is inline with the fact that the SS has a slightly more complex implementation than that of the DS due to the requirement to keep track of a number of different replenishment times and capacities. Second, under both servers, scheduling overheads are higher than context-switch overheads. While the former is related to the time to execute the scheduling code to determine the next task to run, the latter accounts for the time to switch tasks. The time associated to each overhead depends on the underlying LITMUS^{RT} implementation.

Third, worst-case scheduling overhead under either scheduler does not appear to be strongly correlated to the task size, which owes to the fact that fixed priority scheduling is implemented by using bitfield-based ready queues that result in a runtime complexity independent of the number of tasks [57]. The last two trends can also be observed, when average overhead is measured as depicted in Figure 6.6. The inset depicts a slightly decrease in the average scheduling overhead with increasing task count, which can be explained by the increase cache hit rate due to a more frequent invocation of the scheduler, which in turn lowers the average cost of the invocations.

Surprisingly, Figure 6.6 shows that from an overhead point of view, the performance of DS and that of the SS are comparable, since in the worst-case ($n = 4$), their scheduling and context-switch overheads differ

[23]For further details please refer to `https://github.com/LITMUS-RT/feather-trace-tools/blob/master/doc/howto-trace-and-process-overheads.md`

[24]An exact copy of the generated task sets is assigned to each type of server

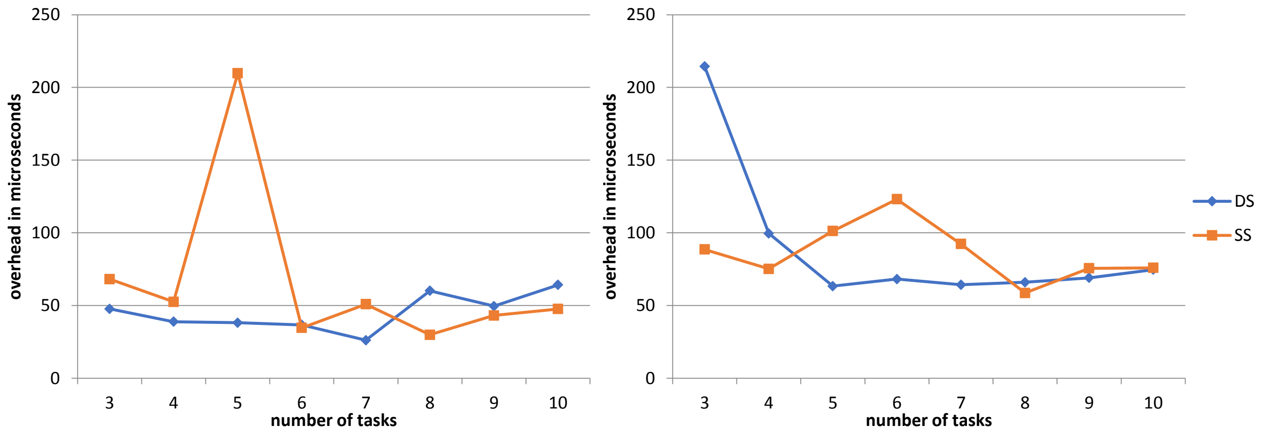


Figure 6.5 Maximum observed context-switch (Left) scheduling (Right) overhead

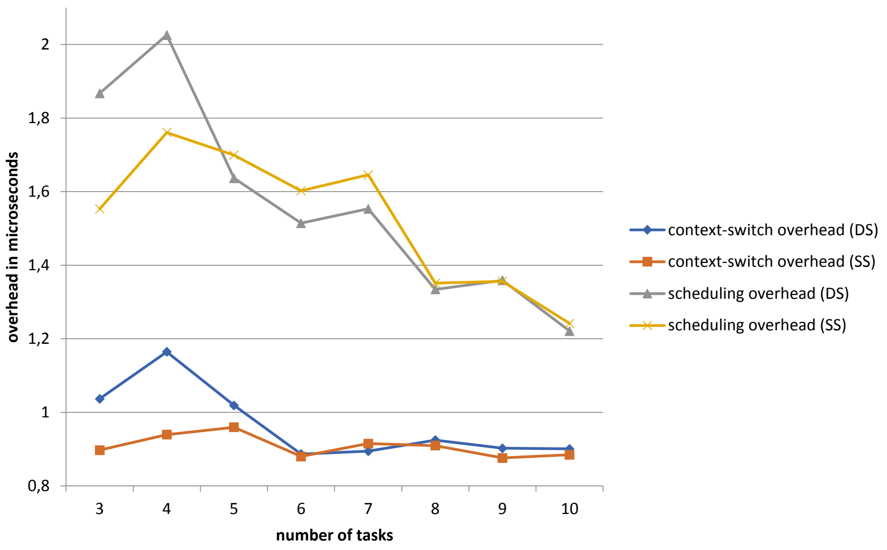


Figure 6.6 Average context-switch and scheduling overhead

in 0.26 and 0.25 microseconds, respectively. Finally, this experiment is concluded by pointing out that the dimension of the overheads cannot be underestimated, for example, in the scheduling overhead case, the minimum observed worst-case overhead for the DS and SS is 68.16 ($n = 5$) and 58.63 ($n = 8$) microseconds respectively. Given that there are many real-life applications composed of tasks with WCETs whose order of magnitude are microseconds, an overhead-aware analysis is of paramount importance.

6.5.4 Server parameter selection

As shown in the previous experiments, the server parameter selection proposed in Section 7.3 can indeed produce optimal results. To show that this effect can also be seen in a real system, task sets based on an industrial case study consisting of an automotive engine control system [2] are synthetically produced.

Hence, task sets ranging from 3 to 9 tasks are generated, where tasks' periods are randomly chosen with uniform distribution from {1, 2, 5, 10, 20, 50, 100, 200, 1000}ms, and randomly assigned to servers. The number of servers ranges from 1 to 6 and the number of task sets is 70.

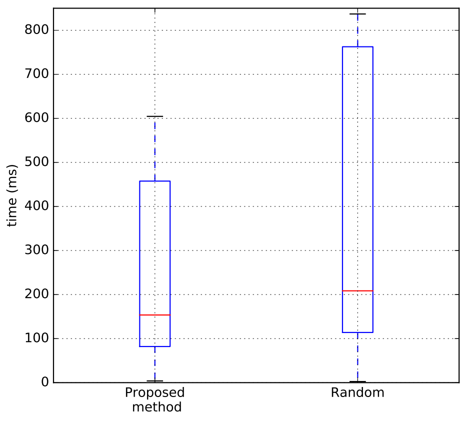
In this experiment, two kinds of parametrization can be distinguished. The first method is derived from Section 7.3, where the period, T_s, and budget, C_s, of a server s, are computed as $T_s = LCM\{T_i\}$, and $C_s = T_s \cdot \sum(C_i + 2 \cdot C^o)/T_i \ \forall$ task τ_i in the server. The maximum worst-case overhead, C^o, is obtained from the results of the previous experiment. The second parametrization involves randomly choosing the server's period and calculating its budget as $C_s = \beta \cdot T_s \cdot \overline{U}_s$, where β is randomly chosen within $[1.2, 1.3]$. On the aforementioned platform, this β parameter proves to be overcompensating for the overhead involved. This process is repeated 100 times per task set and the *best* parameters are chosen.

Figure 6.7 (Left) shows the box plot of the aggregated WCRT in each set for both parametrizations. The inset depicts that the inter-quartile range, defined by the first quartile, Q_1, and the third quartile, Q_3, of the first method (Q_1 = 82.03, Q_3 = 457.58) is lower than that of the other (Q_1 = 113.91, Q_3 = 762.58). Moreover, the median (Q_2 = 208.46) and maximum value (max = 836.99) of the random parametrization are much higher than those of its counterpart (Q_2 = 153.61, max = 604.49). This indicates that the proposed parametrization indeed yields better results than a trial-and-error approach. Observe that this technique produces optimal results provided that the system is schedulable. In this experiment the suggested parametrization method resulted in 89% of the generated task sets being schedulable.

Furthermore, even if optimizing the aggregated WCRT does not necessarily imply that each individual WCRT is minimal, the proposed parametrization can also address this problem and achieve good results as illustrated by Figure 6.7 (Right). While the maximum value of the random parametrization is 808.69, that of the other approach is 602.32. Splitting the jobs released by higher priority tasks can lead to higher interference in lower priority ones, which in turn results in larger response times, since the number of runtime overheads increases as well. The figure also shows that the median of the random approach, 3.13, is smaller than that of the other method, 4.32. This indicates that the proposed technique cannot always find the lowest WCRT of each task.

6.5.5 Comparison of fixed priority servers

Of particular interest to the automotive domain is the use of servers based on fixed priority scheduling, due to the simpler implementation and smaller scheduling overhead [11]. Akin to the study conducted in a previous subsection where the DS and SS are compared in terms of overhead, the following experiment aims at examining if the simpler implementation of the DS is reflected in a similar or even better performance than that of its counterpart, when the parameter selection presented in this chapter is applied to both servers.



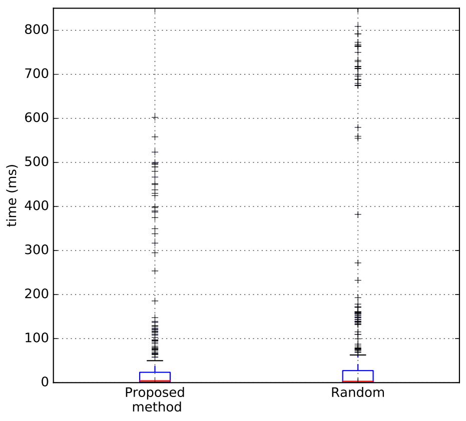
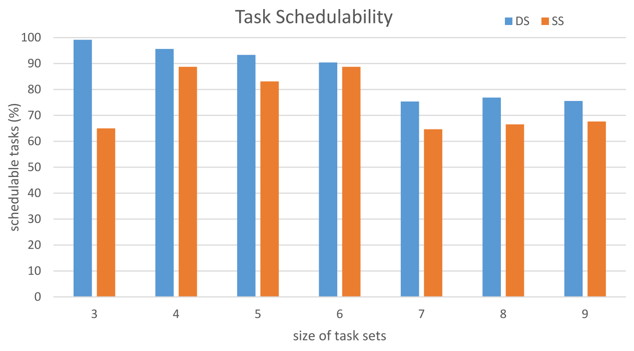


Figure 6.7 Aggregated (Left) / Individual (Right) WCRT

For this experiment, groups of 300 randomly generated task sets24^0 are used with the same setup as in the previous experimentation. Fig. Figure 6.8 (Left) depicts the percentage of schedulable tasks for both servers under the aforementioned setting. Whereas for $n = 6$, the DS can schedule 1.67%, more tasks than its counterpart, for $n = 3$ the DS guarantees that 34.17% more tasks meet their deadlines. Furthermore, while the minimum percentage of schedulable tasks is 75.36% (DS) and 64.64% (SS) for $n = 7$, the maximum percentage is 99.17 (DS) and 88.75 (SS) for $n = 3$ and $n = 4, 6$ respectively.

This trend can also be noticed in Fig. Figure 6.8 (Right), where the distribution of each individual normalized WCRT, i.e. the ratio of a task's WCRT to its period, for both servers is shown. Although the first and third quartile, as well as the median of the box plots for the DS are lower (except for $n = 9$ where Q_3 is higher), the type of server resulting in shorter response times depends heavily on the given task set as shown by the fact that for $n = 3, 4, 8$ the SS presents the lower minimum and maximum value. While a high priority DS can cause an interference of two times its budget on lower priority servers by preserving its capacity until near the end of its period, aka *double* or *back-to-back hit* phenomenon, a SS replenishes its budget one period after the server activation, and only by the amount of capacity that has been consumed in that time interval. In the same settings, any of these algorithmic peculiarities can lead to larger response times.

In the light of the points mentioned above, it can be concluded that at the very least the DS performs equally well as the SS from an overhead as well as from a schedulability point of view. Nonetheless, the best choice varies on a case-by-case basis since the execution demand pattern of the tasks scheduled by a server contribute to the rise of adverse side effects innate in the nature of each server algorithm. Although actual implementations of both servers can be found in hypervisors such as RT-XEN [19], the SS is the only one



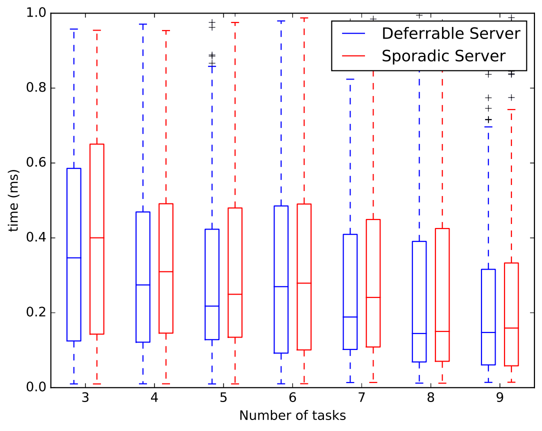


Figure 6.8 Schedulability (Right) and Normalized WCRT Comparison between the Deferrable and Sporadic Server

specified in the IEEE Portable Operating System Interface (POSIX) standard [20]. Thus, as also suggested in [21] for handling aperiodic requests, a standardization of the DS is highly desirable.

6.6 Summary

In this chapter, an overhead-aware schedulability study of real-time applications in a hierarchical fixed priority preemptive setting was presented and the inadequacy of other methods to search for optimal server parameters was highlighted. Based on the presented analysis, a parameter selection heuristic technique was derived. This technique yields schedulable systems with low utilization and small aggregated WCRTs, responding to an actual need in the automotive industry. Moreover, with real overhead dimensions, it was concluded that the DS and SS are comparable in terms of overhead.

In the next chapter, due to the less complexity of the DS in terms of implementation in comparison to that of the SS, a method to implement this kind of server on top of a ubiquitous AUTOSAR-compliant OS will be presented. Moreover, a heuristic to select its parameters will be also exposed.

CHAPTER 7
INTRODUCING A DEFERRABLE SERVER INTO AUTOSAR

Modified from a paper[58] presented at the 2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)

Jorge Martinez[25,26], Ignacio Sañudo[27]

The compositional timing guarantees provided by server-based systems are of particular interest to the automotive domain, where different software components may be concurrently executed on the same platform. However, the AUTOSAR standard does not propose any API to schedule aperiodic tasks or implement compositional scheduling by means of servers. Despite existing practices in the automotive and avionic domain, where *Time Division Multiple Access* (TDMA) approaches are adopted to provide temporal isolation among different applications, there has been a growing interest in real-time servers [11] [39] as an alternative method to obtain even better results. Of particular interest to the automotive and avionic domains is the use of servers based on fixed priority scheduling, due to the simpler implementation and smaller scheduling overhead. Thus, in this chapter, a method to implement a DS on top of ETAS RTA-OS, a ubiquitous AUTOSAR-compliant OS, will be presented. Moreover, a heuristic to select its parameters will be also exposed, and the effectiveness of this parametrization will be proven by applying the technique to an industrial case study consisting of an automotive engine control system.

7.1 Use Case

In fixed priority preemptive systems, a task scheduled in background, i.e. a task with no other task instances ready to execute, is called *background* task and presents the lowest priority. In this regard, the major problem with background scheduling is that, for high foreground, i.e. higher-priority task loads, the response time of background requests can be too long for certain kind of applications.

In the automotive domain, while scheduled in the slack time left by higher-priority tasks, background tasks present timing constraints, which, if not properly fulfilled, can cause serious problems in the system, jeopardizing the guarantee performed for the critical tasks and causing an abrupt performance degradation. To meet those timing constraints, the priority of the background task can be raised, however, this solution is not completely safe, as the typical execution time of a background task is larger than the period of many foreground tasks, which in turn can lead to deadline misses.

[25]Graduate student at the Univeristy of Modena and Reggio Emilia
[26]Primary researcher and author
[27]Postgraduate researcher at the Univeristy of Modena and Reggio Emilia

A much more flexible solution involves the use of a fixed-priority server with a higher priority than that of the original background task. Thus, if the task of interest is scheduled by this server, the latter prevents the former from consuming more than its assigned CPU bandwidth, and so protects the other tasks in the system providing in this way *temporal isolation*.

Task	C_i	T_i	Π_i
τ_1	1	6	1
τ_2	1	8	2
τ_3	1	24	3
τ_4	40	60	4

Table 7.1 System parameters

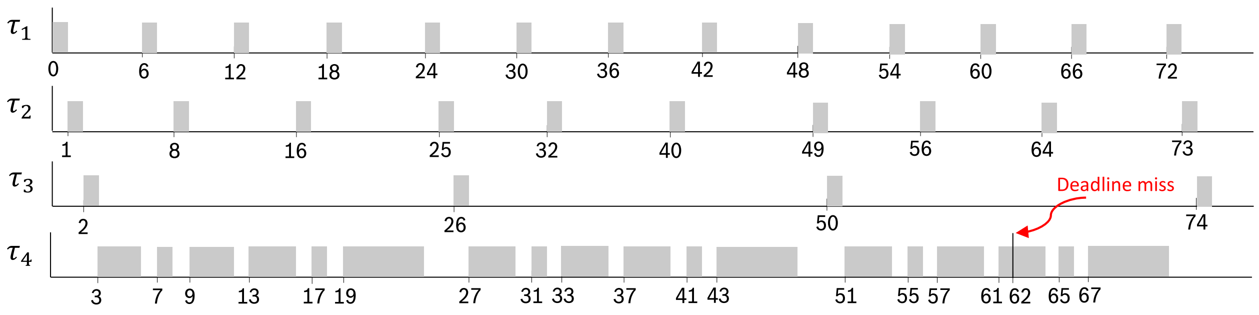


Figure 7.1 Gantt chart of the system in Table 7.1

Consider the system described in Table 7.1, where τ_4 denotes the background in the system. As shown in the Gantt chart depicted in Figure 7.1, all tasks in the system meet their deadlines except for τ_4. Moreover, it can be seen that deliberately increasing the priority of τ_4 to meet its deadline affects the schedulability of the other tasks. However, if τ_4 is scheduled by a server s with a budget $C_s = 4$, period $T_s = 6$, and priority $\Pi_s = 3$ such that $\Pi_1 = 1, \Pi_2 = 2, \Pi_3 = 4$, all tasks meet their deadlines as shown in Figure 7.2 ($R_1 = 1, R_2 = 2, R_3 = 24, R_4 = 60$).

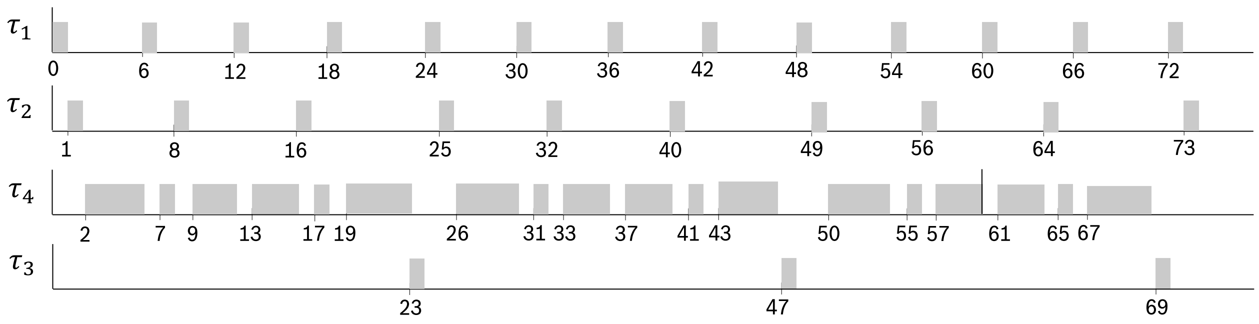
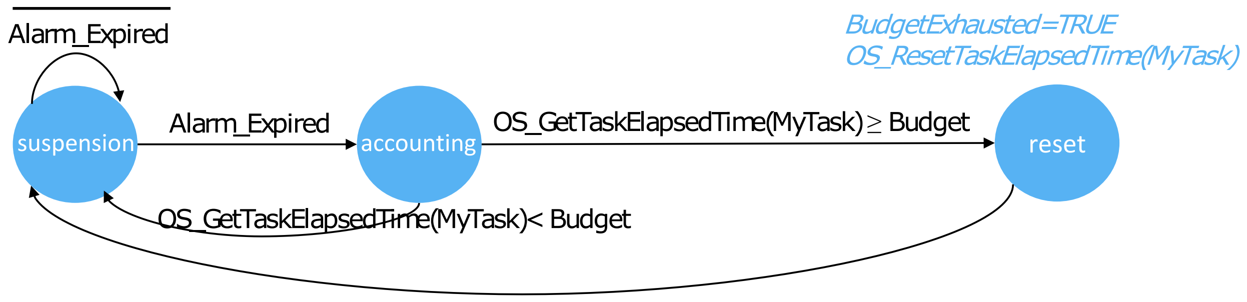


Figure 7.2 τ_4 served by a Deferrable Server s ($C_s = 4, T_s = 6$)

This example illustrates that by introducing a Deferrable Server, a task can still meet its deadline, that otherwise would have been missed if scheduled in the background, without jeopardizing the schedulability of any foreground task.

7.2 Deferrable Server Implementation

As previously mentioned, the AUTOSAR standard does not propose any API to assign a fraction of the CPU bandwidth to a given task by means of a server. Thus, based on the API calls presented in Section 2, a method to implement a DS is presented so that a specified amount of CPU time C_s in every interval T_s can be reserved for a given task. To that end, a *Finite State Machine* (FSM) is first presented that provides a systematic way to introduce a task, by the name of *Budget Accounting Task*, that keeps track of the budget consumed by another task served by a DS. Then, a second FSM is presented that shows how the latter task can be scheduled by a DS.



$\overline{\text{Alarm\_Expired}}$
suspension
Alarm_Expired
accounting
OS_GetTaskElapsedTime(MyTask)≥ Budget
reset
BudgetExhausted=TRUE
OS_ResetTaskElapsedTime(MyTask)
OS_GetTaskElapsedTime(MyTask)< Budget

Figure 7.3 Budget Accounting Task FSM

7.2.1 Budget Accounting (BA) Task

As depicted in Figure 7.3, the FSM is composed of three states: *suspension, accounting*, and *reset*. The default state of the task is *suspended* and is moved into the *accounting* state upon expiration of its associated alarm. Recall that when an alarm expires, it can activate a task, raise an event, etc. As long as the execution time of the task of interest (*MyTask* in the example) given by the output of *OS_GetTaskElapsedTime(MyTask)* is smaller than the assigned budget C_s (given by the scalar *Budget* representing a number of ticks), the task returns to the suspended state. Otherwise, it enters the *reset* state, where the global boolean variable *BudgetExhausted* is set to *TRUE* and the cumulative execution time of *MyTask* is reset, i.e. *OS_ResetTaskElapsedTime(MyTask)*. The snippet Listing 7.3 shows a possible implementation of the FSM of a task namely *BATask* that takes care of the budget accounting of *MyTask*.

Observe that in this implementation, *BATask* cancels its own alarm once the budget assigned to *MyTask* is depleted, since there is no longer need to track the budget of *MyTask*. Moreover, the priority of *BATask* should be set higher than that of the served task, so that *BATask* can preempt *MyTask* every time *BATask* is dispatched. In this way calling *Os_GetTaskElapsedTime(MyTask)* accurately accounts for the cumulative execution time of *MyTask*.

Listing 7.3: Budget Accounting Task Implementation

```
TASK(BATask) {
  ET = Os_GetTaskElapsedTime(MyTask);
  if(ET >= Budget){
    BudgetExhausted = TRUE;
    //Reset the cumulative execution time to 0
    Os_ResetTaskElapsedTime(MyTask);
    //Deactivate the assigned alarm
    CancelAlarm(Alarm_BATask);
  }
  TerminateTask();
}
```

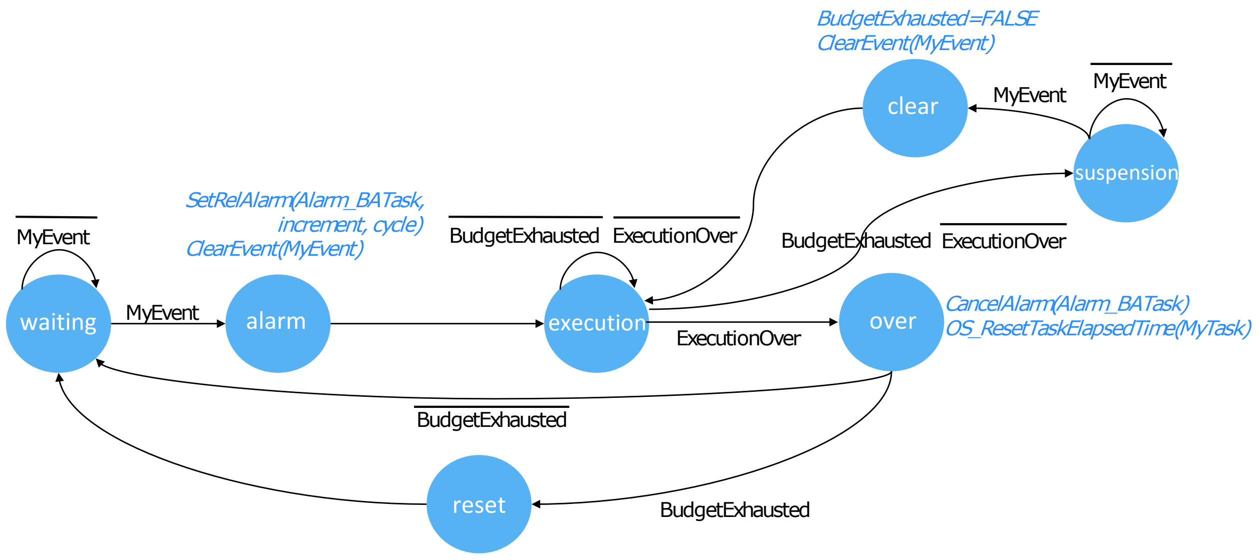

BudgetExhausted=FALSE
ClearEvent(MyEvent)
clear
MyEvent
$\overline{\text{MyEvent}}$
suspension
SetRelAlarm(Alarm_BATask, increment, cycle)
ClearEvent(MyEvent)
$\overline{\text{BudgetExhausted}}$
$\overline{\text{ExecutionOver}}$
BudgetExhausted
$\overline{\text{ExecutionOver}}$
$\overline{\text{MyEvent}}$
waiting
MyEvent
alarm
execution
ExecutionOver
over
CancelAlarm(Alarm_BATask)
OS_ResetTaskElapsedTime(MyTask)
$\overline{\text{BudgetExhausted}}$
reset
BudgetExhausted

Figure 7.4 A task served by a Deferrable Server

7.2.2 An Extended Task as a Deferrable Server

As shown in the previous section, tasks are similar to C functions that implement some form of system functionality when they are called by the OS. In this section the design of an extended task as a deferrable server is presented, so that this task can execute *at most* C_s units of time every T_s. The FSM of this extended task is depicted in Figure 7.4. In the inset it can be seen that initially the served task finds itself in the *waiting* state, where it waits on the event *MyEvent*. This event is to be set by an assigned alarm set up during the configuration of the OS and whose cycle time corresponds to the period of the desired DS, i.e. T_s.

Once this event is set, the task enters the *alarm* state, where it sets the alarm *Alarm_BATask* by means of *SetRelAlarm(Alarm_BATask, increment, cycle)*, which upon expiration activates the BA task that will keep track of the cumulative execution of the served extended task. The parameters *increment* and *cycle* are defined at design time depending on the granularity of the invocation of the BA task. After *MyEvent* is cleared by *ClearEvent(MyEvent)*, the served task is moved into the *execution* state, where the actual code is executed polling the value of the global variable *BudgetExhausted*. Recall that this variable is to be set by the BA task.

If the execution of the code is over (denoted in the inset by *ExecutionOver*), then the served task enters the *over* state, where its cumulative execution is reset through *OS_ResetTaskElapsedTime(MyTask)* and the alarm *Alarm_BATask* is cancelled via *CancelAlarm(Alarm_BATask)*, as there is no longer need for keeping track of its execution. If right before the task enters this state, its CPU budget is depleted, the task is moved into the *reset* state, after leaving the *over* state, where the boolean variable *BudgetExhausted* is set to *FALSE*. Once *BudgetExhausted = FALSE*, the task returns to the *waiting* state.

However, in the *execution* state, if the CPU quota of the served task is depleted before the execution of its code is over, the task enters the *suspension* state, where it waits on *MyEvent*. Notice that the task cannot go back to the *waiting state* since there is still code to be executed and returning to *waiting* will usually imply skipping that part of the code. Once *MyEvent* is set, the task is moved into the *clear* state, where it clears the event via *ClearEvent(MyEvent)* and sets *BudgetExhausted* to *FALSE*. The task then goes straight into the *execution* state and continues the execution of its code. Listing 7.4 shows a possible implementation of the FSM for a dummy task *MyTask* that increments a variable *mycounter* up to a value given by *max_number* before resetting the counter.

Listing 7.4: Deferrable Server Implementation

```
TASK(MyTask){
  while (TRUE){
   WaitEvent(MyEvent);
   SetRelAlarm(Alarm_BATask, increment, cycle);
   ClearEvent(MyEvent);
   while (!BudgetExhausted){
      mycounter++;
      if(mycounter == max_number){
            mycounter = 0;
            //Deactivating the BA task
            Os_ResetTaskElapsedTime(MyTask);
            CancelAlarm(Alarm_BATask);
            break;
          }
        }
        if(BudgetExhausted) //Resetting the budget
          BudgetExhausted = FALSE;
      }
    }
```

Observe that upon depletion of its CPU quota, the implementation of the FSM for this particular task can be done without the *suspension* and *clear* state, and instead return to the *waiting* state. The rationale behind this idea is that when the inner while-loop is broken and the tasks suspends itself, the most recent value of *mycounter* is kept, in this way, when the task resumes its execution, it continues where it left.

7.3 Server Parametrization

In the context of hierarchical fixed priority preemptive scheduling, it has been shown in [59] that an increase in the remaining utilization of a system can be achieved by choosing server periods that are exact divisors of their task periods. Based on this idea, we propose a method to derive the parameters of a DS s serving a task τ_i, that when scheduled in the background would miss its deadline. This server parametrization technique (Algorithm 16), when possible, returns a set of parameters that make the task of interest schedulable without jeopardizing the schedulability of the system. Note that in the algorithm HP (resp. LP) represents the highest (resp. lowest) possible priority assigned to the server.

Algorithm 16 Server Parameter Selection

```
1: Input: $T_i$, $C_i$, $HP$, $LP$
2: Get the set of common divisors, $\Gamma$, of $T_i$ and $C_i$
3: for each $\pi \in [HP, LP]$ do
4:     Set $\Pi_s = \pi$
5:     for each $k \in \Gamma$ in ascending order do
6:         Set $T_s = T_i/k$ and $C_s = C_i/k$
7:         Check if the system is schedulable via [11]
8:         if schedulable then return $T_s$, $C_s$, $\Pi_s$
```

Algorithm 16 starts by calculating the set Γ of common divisors of T_i and C_i. Then, it allocates the server to the highest possible priority, i.e. $\Pi_s = HP$, and for each number k in Γ, the algorithm proceeds to set the parameters of s equal to those of τ_i divided by k, i.e. $T_s = T_i/k$ and $C_s = C_i/k$. If for a given k

the system is schedulable, then Algorithm 16 terminates and returns C_s, T_s, and Π_s as feasible parameters. Otherwise, this process is repeated by decreasing the priority of the server. If allocating the server to priority LP does not make the system schedulable, then the algorithm cannot find feasible parameters.

Example 7.28. *Given the system described in Table 7.1, $HP = 1$ and $LP = 3$, in order to calculate feasible parameters for a server s scheduling τ_4 without jeopardizing the schedulability of the system, Algorithm 16 follows the next steps:*

1. *Obtain the set of common divisors, Γ, of $T_4 = 60$ and $C_4 = 40$. $\Gamma = \{1, 2, 4, 5, 10, 20\}$.*

2. *Set $\Pi_3 < \Pi_2 < \Pi_1 < \Pi_s = HP$.*

3. *Starting with $k = 1$, set $T_s = T_4/k$ and $C_s = C_4/k$, and check the schedulability of the system. In the example, [11] yields the system unschedulable.*

4. *Repeat the process, in ascending, order with each $k \in \Gamma$, until a set of feasible parameters is found. In our case, $k = 10$ yields a feasible system ($R_1 = 5, R_2 = 6, R_3 = 24, R_4 = 58$), and so the algorithm terminates returning $T_s = 6$, $C_s = 4$ and $\Pi_s = 1$.*

Note that the algorithm tests each number $k \in \Gamma$ in ascending order as smaller values of k are preferred. Remember that splitting the background task in k chunks involves some extra overhead due to each invocation of its associated server. Moreover, the algorithm can be further optimized by only testing the first n numbers in Γ, where n is a number defined at design time. It is at this development stage that the priorities HP and LP are to be chosen as well. For instance, if $HP = LP = 3$, Algorithm 16 returns as feasible parameters $T_s = 12$ and $C_s = 8$. Notice that Π_s should be higher than the priority of its assigned task as scheduling the server in the background brings no benefits.

7.4 Evaluation

To prove the effectiveness of the implementation and algorithm, an automotive application representing an engine control system is considered. As previously explained, the application is composed of multiple tasks partitioned onto four cores. The center of attention is one particular core whose tasks have the following periods: {2 , 5, 20, 50, 100, 200, 1000} ms. In order to replicate the settings on top of an Infineon's TC297 processor @ 80MHz, synthetic tasks are generated that are scheduled by ETA's RTA-OS V5.7.0. Moreover, HighTec GCC Compiler v4.9.2.0 is used to produce code for the target. Additionally a synthetic task (*Task_Bg*) is created that is to be scheduled in the background, whose period and WCET are 50ms and 10ms respectively. Figure 7.5 shows the traces stored on the Onchip Trace Buffer of the microcontroller and obtained by means of Lauterbach's TRACE32 debugger.

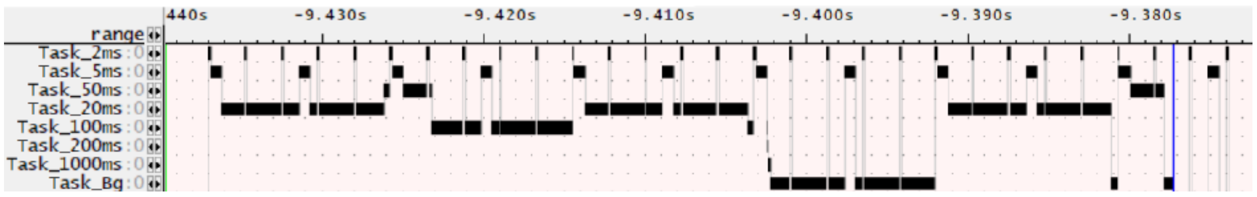


Figure 7.5 The background task misses its deadline as its first instance takes 58,9 ms (blue line)

As shown in Figure 7.5, Task_Bg is unschedulable. Moreover, allocating Task_Bg to a higher priority leads to at least of one of the tasks in the system to miss its deadline. Thus, the task is placed inside a server s, whose parameters are chosen by means of Algorithm 16. HP (resp. LP) is such that the server priority can be lower (resp. higher) than that of the task with a 5ms (resp. 1000ms) period. Figure 7.6 depicts the corresponding trace information. Algorithm 16 reveals that the system is schedulable with HP and $k = 2$ $(T_s = 25, C_s = 5)$ as depicted in Figure 7.6. For this experiment, we chose a budget accounting granularity of $100\mu s$. Moreover, measurements conducted with the TRACE32 debugger showed that the overhead induced due to each invocation of the BA task was 11,494 μs on average. The budget of the server was inflated in order to take the induced overhead into consideration.

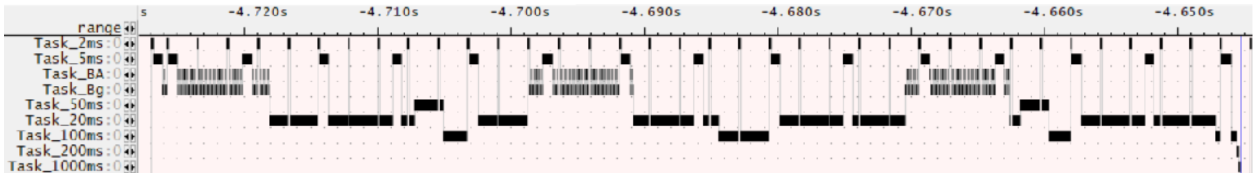


Figure 7.6 Allocating the task to a server $(T_s = 25, C_s = 5)$ results in a schedulable system

7.5 Summary

In this chapter, an implementation of a Deferrable Server on top of RTA-OS was presented so that an originally unschedulable task can meet its deadline. A parametrization technique to properly dimension the server without jeopardizing the schedulability of the system was also provided. Although the main focus were background tasks, the method can be applied to any foreground task.

CHAPTER 8
CONCLUSION

With respect to the Logical Execution Time, the main objectives of the research presented in this dissertation were to determine how to shorten the end-to-end latency of an effect chain composed of tasks following the LET semantics (Question Q1 in Chapter 1), as well as how to improve its determinism by reducing the jitter of a given effect chain (Question Q2). Regarding fixed priority servers, the focus of this work was to provide an exact Response Time Analysis for jobs released by tasks scheduled by fixed priority servers (Question Q3), study the server parametrization problem (Question Q4), compare the Sporadic Server against another fixed priority server (Question Q5), and assess the possibility of implementing one type of fixed priority server on top of an AUTOSAR-compliant OS (Question Q6).

To answer these questions, we provided a deep study of the LET communication semantic, as well as a framework that lead to an exact RTA of jobs in a fixed priority hierarchical setting. In the following, we first summarize our results (Section 8.1), and then discuss open questions and future work (Section 8.2).

8.1 Summary of results

Our research makes novel contributions in two areas of the real-time theory, namely end-to-end latency, and fixed priority real-time servers. In the following, we briefly recapitulate the key points of Chapters 3-7.

8.1.1 Logical Execution Time

In this work, due to a great concern over the optimization end-to-end propagation latencies of ECs in automotive applications, an analytical characterization of the end-to-end latency of effect chains composed of periodic tasks communicating using the LET model was presented. A closed formula expression was provided to compute reading and publishing points where the actual communication between tasks takes place. Based on these points, the end-to-end latency may be computed considering the basic paths of an effect chain within a hyperperiod of the communicating tasks. The analysis was then extended to consider task offsets. An offset assignment method was suggested to further improve the determinism of the end-to-end latency, reducing control jitter. The effectiveness of the LET model in achieving a more deterministic end-to-end communication delay was shown by means of an industrial case study from the automotive domain. Moreover, a set of experiments was presented that showed that the jitter of the end-to-end latency with the LET model is in average within 1% for representative task sets, analytically confirming the control determinism of the LET model. However, non harmonic effect chains may have significantly higher jitters. In these cases, a considerable jitter reduction can be obtained using the proposed offset assignment heuristics.

8.1.2 Fixed Priority Servers

As server-based systems provide the much needed temporal and spatial isolation, by virtue of their design, and hence are of particular interest to the automotive domain, this work also presented a formal characterization of an exact response-time analysis for fixed priority systems based on fixed priority servers (PP, extended PP, DS, and SS) in a multi-level scheduling setting under preemptive scheduling. An experimental characterization on top of $LITMUS^{RT}$ of the schedulabilty improvement that can be obtained with respect to existing sufficient schedulability tests [17] was also exposed, which proved the effectiveness of the proposed exact analysis.

As the effect of the overheads induced by the actual implementation of a given fixed-priority server algorithm can have significant, and even adverse impact on the schedulability of a system, an extension of the RTA that considered the effect of overheads was provided. This extension allowed an investigation into server parameter selection, which in turn, highlighted the inadequacy of other methods to search for optimal server parameters. Based on the same overhead-aware analysis, a parameter selection heuristic technique that yields schedulable systems with low utilization and small aggregated WCRTs was derived. Moreover, with real overhead dimensions obtained from the actual implementation of DSs and SSs on $LITMUS^{RT}$, it was observed that the DS and SS are comparable in terms of overhead and schedulabilty.

Therefore, due to its less complex implementation the DS was chosen as the favoured server to be implemented on top of RTA-OS, an AUTOSAR-compliant OS, so that an originally unschedulable task can meet its deadline, responding to an automotive industry use case. Additionally, a parametrization technique to properly dimension the server without jeopardizing the schedulability of the system was also provided. Although the main focus of the aforementioned implementation was to serve background tasks, the method can be applied to any foreground task.

8.2 Open Questions and Future Work

This dissertation opens a new line of research on end-to-end latency: while there are studies that analytically and experimentally compare end-to-end delays for the LET model against the implicit and explicit communication counterparts, such as [6], there is currently no research that does the same when offsets are assigned to tasks. While the LET model allows significantly reducing the variability in the end-to-end communication delays, the absolute latencies tend to be higher than those with other communication paradigms where tasks publish their computed result at an earlier time. Thus, a thorough comparing study is in order to understand pros and cons of each communication model, even when offsets are present, paving the way towards a generalized method that allows obtaining smaller end-to-end latencies within a reduced jitter. Moreover, as mentioned in Section 3.6, offset assignment can shorten the age latency of a particular effect chain, as it might also potentially lengthen the end-to-end latency of another chain. Thus, a holistic method to cope with the latency minimization problem of multiple effect chains having one or more tasks in common is highly desirable.

This work also opens up new research directions in hierarchical fixed priority preemptive scheduling, as it lays the foundations for new parametrization algorithms based on the presented analysis, as well as extensions in order to consider interrupt-related overheads and cope with dynamic priority servers. Furthermore, while we highlighted the benefits of using a Deferrable Server to raise the priority of a task originally scheduled in the background, and presented a method to construct this server on top of RTA-OS, without modifications of the kernel, our current implementation supports one task per server only. We believe that the automotive industry can benefit from a complete implementation of a two-level hierarchical fixed priority scheduler, similar to [10] but at OS-level.

REFERENCES

[1] Nicolas Navet and Francoise Simonot-Lion. *Automotive Embedded Systems Handbook*. CRC Press, Inc., USA, 1st edition, 2008. ISBN 084938026X.

[2] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

[3] Alan Burns and Andrew J. Wellings. *Real-time systems and programming languages: Ada, real-time Java and C/real-time POSIX*. Addison-Wesley, 2009.

[4] A. Hamann, D. Ziegenbein, S. Kramer, and M. Lukasiewycz. 2017 formals methods and timing verification (fmtv) challenge. pages 1–1, 2017. URL https://waters2017.inria.fr/challenge/.

[5] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. Communication Centric Design in Complex Automotive Embedded Systems. In Marko Bertogna, editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:20, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-037-8. doi: 10.4230/LIPIcs.ECRTS.2017.10. URL http://drops.dagstuhl.de/opus/volltexte/2017/7162.

[6] J. Martinez, I. Sañudo, and M. Bertogna. End-to-end latency characterization of task communication models for automotive systems. In *Real-Time Syst*, 2020. URL https://doi.org/10.1007/s11241-020-09350-3.

[7] Nico Feiertag, Kai Richter, Johan Nordlander, and Jan Jonsson. A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In *IEEE Real-Time Systems Symposium: 30/11/2009-03/12/2009*. IEEE Communications Society, 2009.

[8] Rémy Wyss, Frédéric Boniol, Claire Pagetti, and Julien Forget. End-to-end latency computation in a multi-periodic design. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1682–1687, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1656-9.

[9] T. A. Henzinger, C. M. Kirsch, M. A. A. Sanvido, and W. Pree. From control models to real-time code using giotto. *IEEE Control Systems*, 23(1):50–64, Feb 2003. ISSN 1066-033X.

[10] D. Dasari, M. Pressler, A. Hamann, D. Ziegenbein, and P. Austin. Applying reservation-based scheduling to a µc-based hypervisor: An industrial case study. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 987–990, 2020. doi: 10.23919/DATE48585.2020.9116385.

[11] Arne Hamann, Dakshina Dasari, Jorge Martinez, and Dirk Ziegenbein. Response time analysis for fixed priority servers. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, RTNS '18, pages 254–264, Chasseneuil-du-Poitou, France, 2018. ACM. ISBN 978-1-4503-6463-8. doi: 10.1145/3273905.3273927. URL http://doi.acm.org/10.1145/3273905.3273927.

[12] V. Pradeep Kumar and A. S. Pillai. Dynamic scheduling algorithm for a utomotive safety critical systems. In *2020 Fourth International Conference on Computing Methodologies and Communication (ICCMC)*, pages 815–820, 2020. doi: 10.1109/ICCMC48092.2020.ICCMC-000151.

[13] Brinkley Sprunt. *Aperiodic Task Scheduling for Real-time Systems.* PhD thesis, Pittsburgh, PA, USA, 1990. AAI9107570.

[14] John P. Lehoczky, Lui Sha, and Jay K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Computers*, 44:73–91, 1987.

[15] Ken Tindell. Adding time-offsets to schedulability analysis. 2007.

[16] N.C. Audsley. On priority assignment in fixed priority scheduling. *Information Processing Letters*, 79 (1):39 – 44, 2001. ISSN 0020-0190.

[17] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, RTSS '05, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2490-7. doi: 10.1109/RTSS.2005.25. URL https://doi.org/10.1109/RTSS.2005.25.

[18] Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, RTSS '03, pages 2–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2044-8. URL http://dl.acm.org/citation.cfm?id=956418.956612.

[19] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. Rt-xen: Towards real-time hypervisor scheduling in xen. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 39–48, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0714-7. doi: 10.1145/2038642.2038651. URL http://doi.acm.org/10.1145/2038642.2038651.

[20] Mark Stanovich, Theodore P. Baker, An-I Wang, and Michael Gonzalez Harbour. Defects of the posix sporadic server and how to correct them. In *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '10, pages 35–45, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4001-6. doi: 10.1109/RTAS.2010.34. URL https://doi.org/10.1109/RTAS.2010.34.

[21] Guillem Bernat and Alan Burns. New results on fixed priority aperiodic servers. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, RTSS '99, pages 68–, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0475-2. URL http://dl.acm.org/citation.cfm?id=827271.829112.

[22] Mike Holenderski, Reinder J. Bril, and Johan J. Lukkien. An efficient hierarchical scheduling framework for the automotive domain. In *Real-Time Systems, Architecture, Scheduling, and Application*, chapter 4. 2012. doi: 10.5772/38266. URL https://doi.org/10.5772/38266.

[23] Tarun Gupta, Erik J. Luit, Martijn M. H. P. van den Heuvel, and Reinder J. Bril. Experience report: Towards extending an OSEK-compliant RTOS with mixed criticality support. *e-Informatica*, 12:305–320, 2018.

[24] Automotive open system architecture (autosar). URL https://www.autosar.org/.

[25] RTA-OS User Guide. Software guide, ETAS GmbH, 2016.

[26] J. Martinez, I. Sañudo, and M. Bertogna. Analytical characterization of end-to-end communication delays with logical execution time. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2244–2254, 2018.

[27] T. A. Henzinger, B. Horowitz, and M. Kirsch. Embedded control systems development with giotto. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, LCTES '01, pages 64–72, New York, NY, USA, 2001. ACM. ISBN 1-58113-425-8. doi: 10.1145/384197.384208. URL http://doi.acm.org/10.1145/384197.384208.

[28] C.M. Kirsch and A. Sokolova. The logical execution time paradigm. In *Advances in Real-Time Systems*, pages 103–120, 2012. URL /pubpdf/ARTS-chapter.pdf.

[29] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. Synthesizing job-level dependencies for automotive multi-rate effect chains. In *The 22th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, August 2016. URL http://www.es.mdh.se/publications/4368-.

[30] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. End-to-end timing analysis of cause-effect chains in automotive embedded systems. *Journal of Systems Architecture*, 80 (Supplement C):104 – 113, 2017. ISSN 1383-7621. doi: https://doi.org/10.1016/j.sysarc.2017.09.004. URL http://www.sciencedirect.com/science/article/pii/S1383762117300681.

[31] J. C. Palencia and M. Gonzalez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, pages 26–37, Dec 1998.

[32] O. Redell and M. Torngren. Calculating exact worst case response times for static priority scheduled tasks with offsets and jitter. In *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 164–172, 2002.

[33] J. C. Palencia, M. G. Harbour, J. J. Gutiérrez, and J. M. Rivas. Response-time analysis in hierarchically-scheduled time-partitioned distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(7):2017–2030, July 2017. ISSN 1045-9219.

[34] Joël Goossens. Scheduling of offset free systems. *Real-Time Syst.*, 24(2):239–258, March 2003. ISSN 0922-6443.

[35] Mathieu Grenier, Lionel Havet, and Nicolas Navet. Pushing the limits of CAN - scheduling frames with offsets provides a major performance boost. In *4th European Congress on Embedded Real Time Software (ERTS 2008)*, Toulouse, France, 2008.

[36] A. Monot, N. Navet, B. Bavoux, and F. Simonot-Lion. Multisource software on multicore automotive ecus; combining runnable sequencing with task scheduling. *IEEE Transactions on Industrial Electronics*, 59(10):3934–3942, Oct 2012. ISSN 0278-0046.

[37] Mitra Nasri, Robert I. Davis, and year=2017 Björn B. Brandenburg. Fifo with offsets: High schedulability with low overheads.

[38] T. Kloda, B. d'Ausbourg, and L. Santinelli. Edf schedulability test for the e-tdl time-triggered framework. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, May 2016.

[39] Jorge Martinez, Dakshina Dasari, Arne Hamann, Ignacio Sañudo, and Marko Bertogna. Exact response time analysis of fixed priority systems based on sporadic servers. *Journal of Systems Architecture*, page 101836, 2020. ISSN 1383-7621. doi: https://doi.org/10.1016/j.sysarc.2020.101836. URL http://www.sciencedirect.com/science/article/pii/S1383762120301284.

[40] Goossens C. MacqD. Performance analysis of various scheduling algorithms for real-time systems composed of aperiodic an. 1999.

[41] Dario Faggioli, Marko Bertogna, and Fabio Checconi. Sporadic server revisited. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 340–345, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-639-7. doi: 10.1145/1774088.1774160. URL http://doi.acm.org/10.1145/1774088.1774160.

[42] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004. ISBN 0387231374.

[43] Luca Abeni and Giorgio Buttazzo. Resource reservation in dynamic real-time systems. *Real-Time Systems*, 27(2):123–167, Jul 2004. ISSN 1573-1383. doi: 10.1023/B:TIME.0000027934.77900.22. URL https://doi.org/10.1023/B:TIME.0000027934.77900.22.

[44] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS '98, pages 4–, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-9212-X. URL http://dl.acm.org/citation.cfm?id=827270.829047.

[45] P. Kumar, J. Chen, L. Thiele, A. Schranzhofer, and G. C. Buttazzo. Real-time analysis of servers for general job arrivals. In *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications(RTCSA)*, volume 01, pages 251–258, Aug. 2011. doi: 10.1109/RTCSA.2011.80. URL doi.ieeecomputersociety.org/10.1109/RTCSA.2011.80.

[46] Tei-Wei Kuo and Ching-Hui Li. A fixed-priority-driven open environment for real-time applications. In *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No.99CB37054)*, pages 256–267, Dec 1999. doi: 10.1109/REAL.1999.818851.

[47] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, RTSS '97, pages 308–, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8268-X. URL http://dl.acm.org/citation.cfm?id=827269.828992.

[48] Saowanee Saewong, Ragunathan (Raj) Rajkumar, John P. Lehoczky, and Mark H. Klein. Analysis of hierar hical fixed-priority scheduling. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, ECRTS '02, pages 173–, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1665-3. URL http://dl.acm.org/citation.cfm?id=787256.787352.

[49] Giuseppe Lipari and Enrico Bini. Resource partitioning among real-time applications. In *IN PROC. OF EUROMICRO CONFERENCE ON REAL-TIME SYSTEMS*, pages 151–158. IEEE Computer Society, 2003.

[50] Luis Almeida and Paulo Pedreiras. Scheduling within temporal partitions: Response-time analysis and server design. In *Proceedings of the 4th ACM International Conference on Embedded Software*, EMSOFT '04, pages 95–103, New York, NY, USA, 2004. ACM. ISBN 1-58113-860-1. doi: 10.1145/1017753.1017772. URL http://doi.acm.org/10.1145/1017753.1017772.

[51] P. Balbastre, I. Ripoll, and A. Crespo. Exact response time analysis of hierarchical fixed-priority scheduling. In *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 315–320, Aug 2009. doi: 10.1109/RTCSA.2009.40.

[52] Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1):129–154, May 2005. ISSN 1573-1383. doi: 10.1007/s11241-005-0507-9. URL https://doi.org/10.1007/s11241-005-0507-9.

[53] Jane W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000. ISBN 0130996513.

[54] S. S. Craciunas, C. M. Kirsch, and A. Sokolova. Response time versus utilization in scheduler overhead accounting. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 291–300, April 2010. doi: 10.1109/RTAS.2010.14.

[55] Jorge Martinez, Ignacio Sanudo, and Marko Bertogna. Analytical characterization of end-to-end communication delays with logical execution time. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, PP:1–1, 07 2018. doi: 10.1109/TCAD.2018.2857398.

[56] Marco Di Natale, Wei Zheng, Claudio Pinello, Paolo Giusto, and Alberto Sangiovanni-Vincentelli. Optimizing end-to-end latencies by adaptation of the activation events in distributed automotive systems. pages 293–302, 04 2007. doi: 10.1109/RTAS.2007.24.

[57] Bjorn B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-time Operating Systems*. PhD thesis, Chapel Hill, NC, USA, 2011. AAI3502550.

[58] J. L. Martinez Garcia and I. S. Olmedo. Introducing a deferrable server into autosar. In *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–6, 2020. doi: 10.1109/RTCSA50079.2020.9203616.

[59] R.I. Davis and A. Burns. An investigation into server parameter selection for hierarchical fixed priority pre-emptive systems. In *Proccedings of Real-Time and Network Systems, RTNS*, 2008.