Integrating enhanced slot-shifting in μC/OS-II

Ramachandran, Gowri Sankar

Award date:
2011

Awarding institution:
Mälardalen University

Integrating enhanced slot-shifting in μC/OS-II

Gowri Sankar Ramachandran
grn09001@student.mdh.se

School of Innovation, Design and Engineering
Malardalen University
Sweden

In cooperation with:
Chair of System Architecture and Networking
Department of Mathematics and Computer Science
Eindhoven University of Technology
The Netherlands

Supervisor:
Dr. Damir Isovic
damir.isovic@mdh.se

Remote Supervisor:
Prof. Dr. Johan J. Lukkien
j.j.lukkien@tue.nl

Tutor:
Ir. Martijn van den Heuvel
m.m.h.p.v.d.heuvel@tue.nl

August 25, 2011

Acknowledgement

This work would not have been possible without the support of kind people around me. I have worked with great people during this work. I have learned a lot from each and every individual. It is a pleasure to convey my gratitude to all those who helped and influenced me during this work.

I am grateful to my supervisor at MDH, Dr. Damir Isovic for sending me to TU/e and for supervising my work from Sweden. Thanks for all your support, encouragement and the useful discussions during my entire study period in MDH (Sweden) and as well as in TU/e (Netherlands).

A special thanks to my supervisor at TU/e, Prof. Dr. Johan J. Lukkien for giving me this opportunity to work in TU/e. Without your valuable advises, encouragements and the useful discussions, this work could not have been achieved.

I thank my daily supervisor at TU/e, Ir. Martijn van den Heuvel, for all his support, encouragement and valuable suggestions. A special gratitude for taking your time to review my thesis. I wish you all the very best for your doctoral studies.

I would also extend my gratitude to Dr. Reinder J. Bril for his valuable suggestions at the initial stages of this work. I am grateful to Ir. Mike Holenderski for helping me out with Grasp visualization tool and for his useful inputs during the various stages of this work. I thank everyone in SAN group for creating a good working environment and an enjoyable time.

Last but not least, a special thanks to my family and friends for supporting me during my entire study period.

Abstract

The growing complexity of embedded applications poses new challenges in the application development phase. The time-critical nature of the embedded applications leads to the use of an Real-Time Operating System (RTOS). Typically in a RTOS, applications are divided into a small number of concurrent functional units called tasks. The execution of tasks is accomplished through the scheduler of a RTOS. The job of the scheduler is to pick a task for execution using a scheduling mechanism. The existing scheduling mechanisms handles tasks with only a specific set of constraints such as period and deadline.

However, applications may have a set of more complex constraints such as precedence relations between tasks and non-uniform arrival patterns of tasks. Although time-triggered schedulers can solve all these constraints off-line and scheduling decisions are also made off-line, the disadvantage of this approach is that it requires a complete knowledge of tasks and their constraints. Event-triggered schedulers partially require knowledge of tasks and their constraints, but these can handle dynamically arriving tasks with run-time mechanisms and the scheduling decisions are made online. The combination of time-triggered and event-triggered scheduler is suitable for dealing with tasks with complex constraints. Complex task constraints can be resolved during the off-line preparation phase. During run-time, those resources that are unused by the off-line guaranteed tasks can be used for the execution of dynamically arriving tasks.

The slot-shifting [11, 15] scheduling technique handles mixed task sets with complex constraints by exploiting the advantages of the time-triggered and the event-triggered scheduling paradigms. In the year 1995, Fohler [11] proposed slot-shifting scheduler to deal with periodic and sporadic tasks. Subsequently, in the year 2009, Isovic and Fohler [15] extended slot-shifting scheduling algorithm to deal with periodic, aperiodic and sporadic tasks. In this work, we consider the implementation of the enhanced slot-shifting scheduling algorithm presented in [15].

We integrated slot-shifting scheduling in a off-the-self RTOS, μC/OS-II [18]. The run-time mechanisms are derived from case-studies and subsequently a design of the run-time mechanisms targeted at small micro-kernels is presented, i.e. interval tracking, task management, WCET monitoring and (online) admission control. The static properties of a slot-shifted scheduler are extracted from [15] and used in the design phase to regulate the run-time behavior of slot-shifting.

The slot-shifting scheduling algorithm uses Earliest-Deadline-First (EDF) [21] scheduling of tasks. We implemented an EDF scheduler in μC/OS-II, on top of the default fixed-priority scheduler of μC/OS-II RTOS. The performance of the EDF scheduler is compared with the default scheduler of μC/OS-II, Fixed-Priority Scheduler (FPS). The event-handling overhead

of the EDF scheduler is approximately 15% higher than the FPS, assuming the distributed arrivals of tasks (one task arrival per slot). Next, we added support for background scheduling in interval-based schedulers, resulting in a full run-time system with support for slot-shifting.

The slot-shifting scheduling algorithm is evaluated and its results are discussed. We used OpenRISC 1000 hardware platform to test our implementation. Furthermore, the slot-shifting scheduling algorithm is evaluated with the FPS scheduler [10] and the results are compared with the slot-shifting with the EDF scheduler [11, 15]. The result indicate that the run-time overhead of slot-shifting is approximately 26% high compared to a standard fixed-priority scheduler, assuming a distributed arrivals of tasks. The memory overhead of the slot-shifting scheduler increases with the number of jobs in the hyper-period of periodic tasks.

The slot-shifting scheduling algorithm is suitable for embedded systems with the complex task constraints. With an improvement in a guarantee test for aperiodic tasks, the predictability and performance of the slot-shifting scheduler can be enhanced. However, we consider such improvements as a future work. Finally, the work is concluded with suggestions for future improvements and extensions.

Table of Contents

Glossary

Notation	Description	Page List
ANSI	American National Standards Institute	30
API	Application Programming Interface	30
CAN	Controller Area Network	16
COTS	Commercial Off-The-Shelf	12
EDF	Earliest-Deadline-First	2
FPS	Fixed-Priority Scheduler	2
GNU	GNU's Not Unix	32
ICB	Interval Control Block	33
ISR	Interrupt Service Routine	31
RAM	Random Access Memory	68
RELTEQ	RELative Timed-Event Queues	15
RISC	Reduced Instruction Set Computing	32
RM	Rate-Monotonic	61
ROM	Read-Only Memory	68
RTOS	Real-Time Operating System	2
TCB	Task Control Block	28
WCET	Worst-Case Execution Time	21

Chapter 1

Introduction

The time-sensitive nature of embedded applications requires a RTOS to guarantee a timely predictable execution. RTOSes are widely used in many application domains, for example: avionics, automotive, consumer applications etc. These applications are divided into concurrent functional units, called tasks. The purpose of the real-time operating system is to ensure both the logical and temporal correctness of operation between concurrent tasks. In this thesis, we focus on the scheduling of tasks using the slot-shifting scheduling algorithm in μC/OS-II RTOS, a real-time operating system for embedded systems. μC/OS-II RTOS is used as a research vehicle in System Architecture and Networking(SAN) group of Eindhoven University of Technology.

In this section, we start with a brief introduction to the slot-shifting scheduling algorithm and we motivate its role in context of the real-time operating systems. Next, we describe the problem of integrating slot-shifting in a commercial RTOS and we present a sequence of steps to be carried out to achieve this goal. Finally, we present the contributions of this work and conclude this chapter with an overview of the thesis.

1.1 Context and background

In real-time operating systems, the selection of a task for execution is carried out by a scheduling strategy. Most of the real-time scheduling strategies are targeted at a specific set of task constraints. Time triggered scheduling provides a method in which activities are initiated at predefined points in time. All the complex constraints are typically assumed to be resolved off-line and during the off-line scheduling phase a schedule is produced in the form of a dispatching table. During run-time, the schedule represented in the table is followed. Based on the system time and the information available at the look-up table, tasks are dispatched for execution during run-time. This form of scheduling is also referred to as the static scheduling, the off-line scheduling or the table-driven scheduling. Time-triggered scheduling is predictable with low run-time overheads, since all the system activities are planned upfront. The downside of this scheduling algorithm is its low resource utilization due to the over-dimensioning of resources to accommodate for worst-case arrivals of sporadic activities. Furthermore, including of a new task after deployment of the system requires a new design of the dispatching table.

In case of the event triggered scheduling, activities are initiated by the occurrence of a particular event. The scheduling decisions are made during run-time. Scheduling is performed based on the priority of a task or the deadline of a task. This type of scheduling is also referred to as online scheduling or dynamic scheduling. Event-triggered scheduling is flexible, but it requires a strict task model with known periodic workloads mapped onto tasks which are relatively independent. Based on the task attributes, priorities are assigned to tasks, e.g. the priority of the task is chosen based on its period with the Rate-monotonic-scheduling [19], or its deadline with Deadline-monotonic-scheduling [20]. Although the down-side of this scheduling algorithm is its run-time overhead compared to table-driven scheduling as the schedule is generated dynamically during run-time, fixed priority scheduling of tasks is the de-facto standard in industry.

Slot shifting proposes a method to schedule mixed task sets by combining time-triggered scheduling and event-triggered scheduling [15, 11]. In other words, the slot-shifting scheduling strategy exploits the advantages of time-triggered and event-triggered scheduling. After preparing a task set during the off-line scheduling phase for inclusion in a priority-based setting, we can distinguish tasks as periodic, aperiodic and sporadic.

Typically, these tasks are classified into three categories based on their arrival patterns. A task which arrives with strictly regular time periods are called periodic tasks. Figure 1.1 shows the periodic task with the time-line. In this example, the periodic task is having a period of 50 time units. Hence, it arrives once every 50 time units.

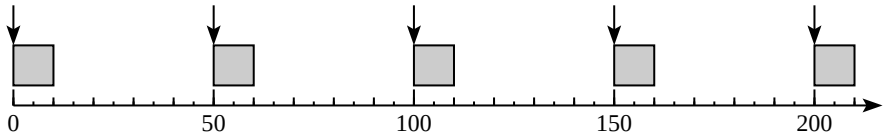

Figure 1.1: Periodic task.

A task which arrives with irregular time intervals is defined as an aperiodic task. An aperiodic task can be activated by a external event. Such tasks are activated at random instances. An example of an aperiodic task is depicted in Figure 1.2. The arrival pattern of the aperiodic task is irregular.

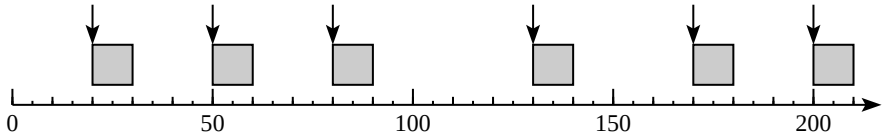

Figure 1.2: Aperiodic task.

Unlike periodic tasks, a sporadic task can arrive at any moment in time, but consecutive instances are separated by a known minimum inter-arrival time. Figure 1.3 presents a sporadic task with a minimum inter-arrival time of 50 time units. There should be at least 50 time unit interval between each sporadic request. It has been shown in literature that in the worst

case these tasks exhibit a periodic behavior [6].

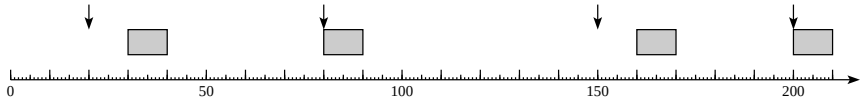

Figure 1.3: Sporadic task.

In complex distributed systems, these tasks are distributed over multiple nodes. Due to the dependencies between tasks running on various nodes, dynamic (online) scheduling of tasks becomes challenging. The slot-shifting technique addresses scheduling of tasks in a distributed setting. In such a distributed setting, the dependencies between tasks running on multiple nodes are resolved off-line in a distributed setting. The focus of this work is the run-time support for a single-node, which is eventually part of a distributed system.

1.2 Motivation

As we discussed in the previous section, embedded applications constitute a mixed set of tasks with various constraints. On the one hand, standard timing constraints such as periods and deadlines must be preserved as part of the functional requirements. On the other hand, non-temporal constraints such as reliability, performance are important factors in the realization of such systems. Handling of such complex task sets therefore becomes challenging on platforms with limited run-time support.

Existing scheduling techniques are targeted towards specific application constraints. For instance, time triggered schedulers are suitable for scheduling (strictly) periodic tasks, e.g audio and video sampling, periodic monitoring of the temperature etc. Since the schedule is generated off-line, only tasks with known attributes such as period, deadline and execution time can be scheduled. However, in many embedded systems, one or more of these attributes may be unknown upfront. The event-triggered scheduling algorithms are capable of handling tasks with known attributes, along with the dynamically arriving tasks, after an online admission test. The dynamic events can also be handled in time triggered schedulers by the polling approach, at the expense of inefficient use of system resources, since polling requires periodic checking for the arrival of events with at least double the speed of the sampling frequency. Event-triggered scheduling with interrupts is therefore more efficient as compared to the time-triggered scheduling with polling for handling dynamic events.

The introduction of the slot-shifting scheduling strategy in a real-time operating system allows efficient handling of mixed task sets, by exploiting the benefits of both the time triggered scheduling and the event triggered scheduling techniques. Periodic and sporadic tasks are guaranteed off-line and the slack time in the system can be used to admit dynamic load during run-time.

The slot-shifting scheduling strategy makes use of the EDF scheduling algorithm. EDF is known for the better resource utilization compared to fixed priority scheduling algorithms. The use of EDF scheduling in slot-shifting therefore enables a better resource utilization

compared to the fixed-priority scheduling.

1.3 Problem description

The complex nature of real-time embedded system leads to applications with complex task constraints. In this assignment, we investigated the slot-shifting scheduling technique to deal with mixed task sets. Most of the Commercial Off-The-Shelf (COTS) RTOSes only have limited support for the scheduling of mixed task sets. COTS RTOSes support either the time triggered scheduling or the event-triggered scheduling. Real-time operating systems such as OSEK-OS [22] and Rubus OS [5] support both the time-triggered scheduling and the event-triggered scheduling techniques. However, both do not allow to reallocate off-line allocated slots to enable a more efficient handling of dynamic events. The slot-shifting scheduling combines the time-triggered scheduling and the event-triggered scheduling. In addition, it requires dedicated run-time mechanisms to dynamically reallocate off-line scheduled slots. To our knowledge, there are no COTS RTOSes available in the market with the support for the slot-shifting scheduling algorithm. Our goal is to design and implement the suitable run-time mechanisms, for scheduling of tasks using slot-shifting, and investigate the feasibility of this approach by evaluating its run-time overheads.

1.4 Approach

The first step is to compare the enhanced slot-shifting scheduling strategy proposed by Isovic and Fohler [15], with other scheduling approaches. The aim is to implement the slot-shifting scheduling strategy in the μC/OS-II RTOS. Secondly, we identify the run-time mechanisms that are essential for the integration of slot-shifting. Thirdly, we carry out the design of the run-time mechanisms before starting with implementation. Fourthly, the slot-shifting scheduling strategy is implemented in the μC/OS-II. Finally, the performance of the slot-shifting scheduling technique is evaluated.

1.5 Contributions

The contributions of this thesis are as follows:

- We identify the slot-shifting run-time entities for
  1. The design and implementation of an interval tracking mechanism.
  2. The design and implementation of mixed task handling mechanism.
  3. The design and implementation of the EDF scheduler within μC/OS-II.
- The support for borrowing and lending of spare capacities is implemented to deal with the early arriving periodic tasks.
- We extend the mechanisms for dynamic resource reclaiming:

1. Isovic and Fohler [15] present a resource-reclaiming mechanism for aperiodic tasks, which was implemented in a simulator. We implemented their resource-reclaiming for aperiodic tasks in a real RTOS rather than in a simulator.
2. The resource-reclaiming mechanism is extended to deal with the early completion of other (periodic and sporadic) tasks.
3. The design and implementation of the dynamic resource-reclaiming mechanism.

- We evaluated
  1. The performance of the EDF scheduler in terms of event handling overhead and its performance is compared with a FPS.
  2. The performance of the slot-shifting scheduler in terms of event handling and memory overhead.
- We extended the Grasp [13] visualization tool with the support for the slot-shifting. This tool is used to visualize events from instrumented code running on a micro-controller.

1.6 Overview

In this chapter, the context, the motivation and the goal of our work is briefly described. The rest of this thesis is organized as follows:

Chapter 2 provides an high-level overview of time-triggered and event-triggered schedulers. We give an overview of existing operating systems with the support for the time-triggered schedulers and event-triggered schedulers. We also discuss operating systems with the support for a combination of time-triggered and event-triggered scheduling.

Chapter 3 presents the task and the time model used in this work. We derive the static system properties from the slot-shifting technique proposed in [15].

Chapter 4 discusses the slot-shifting scheduling technique in detail. The principle of slot-shifting is presented with an example task set. The required run-time mechanisms are identified for the integration of slot-shifting in COTS RTOS.

Chapter 5 provides information about μC/OS-II RTOS and the OpenRISC platform. We briefly introduce the tooling and some of the extensions to μC/OS-II developed in the SAN group.

Chapter 6 presents the design considerations for the run-time mechanisms discussed in Chapter 4. We propose various design alternatives for the integration of slot-shifting in a COTS RTOS.

Chapter 7 discusses the results of the slot-shifting scheduler. Slot-shifting uses EDF for the scheduling of tasks. We present an evaluation based on measured results for our implementation of EDF and a de-facto FPS evaluation is presented. The implementation complexity of the slot-shifting is also discussed in detail.

Chapter 8 concludes this thesis and outlines directions for future work.

Chapter 2

Related Work

The integration of slot-shifting in the μC/OS-II requires support for both the time-triggered scheduler and the event-triggered scheduler. The comparison between time-triggered and event-triggered schedulers are presented in Section 2.1. Section 2.2 presents RTOSes with support for time-triggered scheduling. RTOSes with support for the event-triggered scheduling are discussed in Section 2.3. In the slot-shifting algorithm, a time-triggered scheduler and a event-triggered scheduler are combined to result in a hybrid scheduler. Section 2.4 presents the RTOSes with the support for a hybrid scheduler. Section 2.5 discusses slot-shifting with FPS and EDF.

2.1 Time-triggered versus event-triggered scheduling

The factor influencing the decision on the time-triggered or event-triggered scheduling can be viewed in terms of the non-functional factors such as analyzability, predictability, testability, extensibility, fault tolerance and resource utilization. These factors are discussed extensively in [16] and further analyzed in [27].

2.1.1 Time triggered systems

Time-triggered systems are those that react to passage of time, i.e., all activities are initiated at predetermined points in time. Real-time systems of this kind are time triggered in the sense that their overall behavior is globally controlled by a recurring clock tick [14].

The time-triggered scheduler can be viewed as follows with the factors described above:

- **Analyzability:** Off-line analysis results in a statically computed schedule based on the given temporal constraints.
- **Predictability:** Follows the statically computed schedule, so the current state of the system can be predicted at the given time.
- **Testability:** Assuming the worst-case behavior of tasks, formal techniques can be used to ensure correct behavior.

- **Extensibility:** Addition of a new task requires complete recalculation of the static schedule.
- **Fault tolerance:** State synchronization with redundant nodes is easily achievable.
- **Resource utilization:** With aperiodic and sporadic events, the polling for events utilizes additional resource.

2.1.2 Event-triggered systems

In event-trigged systems all activities are carried out in response to relevant events external to the system, e.g., a sensor generates an interrupt which triggers a certain task. Temporal control is enforced from the environment onto the system in an unpredictable manner (interrupts) [14].

The event-triggered scheduler can be viewed as follows:

- **Analyzability:** Response time analysis is performed to ensure proper behavior during run-time.
- **Predictability:** Lack predictability as the scheduling decisions are made during run-time, and the occurrence of sporadic and aperiodic events affects the predictability.
- **Testability:** Assuming the worst-case behavior of tasks, formal techniques can be used to ensure correct behavior.
- **Extensibility:** Response time analysis must be performed.
- **Fault tolerance:** Hard to synchronize with redundant nodes and may require additional communication between nodes.
- **Resource utilization:** Sporadic and aperiodic events can be handled with better resource utilization.

The choice of the scheduler for the application can be decided based on the factors presented above. Scheler and Schroeder-Preikschat [27] discuss more about time-triggered and event-triggered schedulers and they present methods to migrate between each other. However, in this work, we focus on the combination of the time-triggered and the event-triggered scheduler to schedule mixed task sets.

2.2 Time-triggered schedulers

Real-time operating systems such as MARS [9], OSEKTime [23] supports the time-triggered scheduling. All these operating systems support the time-triggered scheduling by following a schedule presented in the form of a table. During run-time, the dispatcher is invoked to execute the tasks.

In this work, we are looking at two alternative approaches for the implementation of a time-triggered scheduler. One approach is to use the RELative Timed-Event Queues (RELTEQ)

[12] extension to schedule the time-triggered tasks. An other approach is to integrate a time-triggered scheduler with a dispatcher and a representation of an off-line schedule.

2.3 Event-triggered schedulers

In a event-triggered system, multiple interrupts are handled during run-time [24]. In these kind of systems, interrupts can come from a timer or an other hardware device, e.g. receiving messages from a Controller Area Network (CAN) and other external peripherals. A system with an event-triggered scheduler can handle dynamically arriving events. These kind of systems typically implement a priority-based scheduler, i.e. FPS or EDF.

The VxWorks [1] is one of the real-time operating systems with the support for event-triggered scheduling. This type of operating systems typically makes its scheduling decisions during run-time based on task priorities. Most COTS RTOSes support FPS. The integration of slot-shifting requires an EDF scheduler, however. In our work, we focus on the EDF scheduler for scheduling of tasks based on their deadline.

2.4 Hybrid schedulers

Hybrid schedulers are the combination of time-triggered and event-triggered schedulers. These schedulers are known for their better resource (processor) utilization. Typically, a time-triggered scheduler is responsible for handling the off-line guaranteed tasks. The job of an event-driven scheduler is to allocate the idle slots of the time-triggered scheduler to schedule dynamically arriving tasks.

Real-time operating systems such as OSEKos [22],Rubus OS [5] support this hybrid scheduling approach. The integration of slot-shifting in the μC/OS-II requires hybrid scheduling mechanism. The difference between traditional hybrid scheduling and slot-shifting is that slot-shifting allows to shift the execution of the time-triggered tasks within an interval starting from the activation of a task until its deadline. In other words, slot-shifting allows guaranteed tasks to be shifted in a pre-defined off-line calculated interval whereas more traditional hybrid approaches do not allow a change in slot allocations of any off-line guaranteed tasks. The support for slot-shifting is not present in any of the COTS RTOS. In this work, the slot-shifting scheduling strategy is integrated in μC/OS-II.

2.5 Slot-shifting: EDF versus FPS

In [10], the idea of combining the off-line scheduling method with the fixed priority scheduling (FPS) is presented. Non-periodic events are handled with the help of servers. The advantage of servers is that it prevents temporal faults in a-periodic tasks to propagate to off-line guaranteed tasks. In combination with slot-shifting, we leave such mechanisms for overload prevention as a future work. In our work, the combination of off-line and EDF scheduling is used to schedule mixed task sets without the use of servers.

2.6 Summary

The RTOSes with the support for the time-triggered, the event-triggered and the hybrid scheduling were presented, but the integration of slot-shifting requires the combination of the time-triggered and the event-triggered scheduler. In other words, an enhanced version of the hybrid scheduler is required to perform the slot-shifting. To our knowledge, RTOSes available in market today does not support the slot-shifting scheduling technique. In this work, we integrate the slot-shifting scheduling technique in μC/OS-II RTOS.

Chapter 3

System model

In this chapter, the system model of the slot-shifting is presented. In Section 3.1, we present the assumptions and boundary conditions of this work. Section 3.2 present the time model. Section 3.3 presents the task model. Section 3.4 presents the static system properties of slot-shifted schedules. Section 3.5 describes the interval invariants.

3.1 Assumptions and boundary conditions

In line with [15], we have the following assumptions:

A1. All tasks in the system have a deadline.

A2. Tasks are independent.

A3. Arrival times of aperiodic requests are unknown.

3.2 Time model

The granularity of the timer tick is assumed to be the *slot length*. Slots are numbered from 0 to ∞. Task characteristics are captured by the intervals. The deadline of a task marks the end of an interval. The start of an interval is either the arrival-time of a task or the end of the previous interval. Task parameters must be multiples of the slot length. The time model we use here is the same as the one proposed in [15].

3.3 Task model

The periodic task set, τ_p, contains a set of periodic tasks P_1, P_2, P_3 ...,P_{n_p}, where n_p is the number of periodic tasks. A periodic task P_i is characterized by a quartet $< T_i^p$, C_i^p, R_i^p, $D_i^p >$, where T_i^p denotes its period, C_i^p denotes its worst-case execution time , R_i^p denotes its release time and D_i^p is its relative deadline. The deadline of a task, D_i^p, is less than or

equal to the period, T_i^p [1]. The k^{th} invocation of the periodic task P_i is denoted as $P_{i,k}$ and is characterized by its earliest start time $est_{i,k}^p$, finishing time $ft_{i,k}^p$ and absolute deadline $d_{i,k}^p$, where $d_{i,k}^p = R_i^p$+(k-1)T_i^p+D_i^p, $k \in \mathbb{N}^+$.

The aperiodic task set, τ_a, contains a set of aperiodic tasks A_1, A_2, A_3 ..., A_{n_a}, where n_a is the number of aperiodic tasks. The aperiodic task A_i is characterized by a pair $< C_i^a, D_i^a >$, where C_i^a is its worst case execution time and D_i^a is its deadline. On arrival of an aperiodic task, it will have a known arrival time ar_i^a, and the absolute deadline $d_i^a = ar_i^a + D_i^a$. The soft aperiodic task is considered as a special type of aperiodic task with an infinite deadline, see Section 6.3.3.

The sporadic task set, τ_s, contains a set of sporadic tasks S_1, S_2, S_3 ..., S_{n_s}, where n_s is the number of sporadic tasks. The sporadic task S_i is characterized by a tuple $< C_i^s, D_i^s, \lambda_i^s >$, where C_i^s is its worst-case execution time, D_i^s is its relative deadline and λ_i^s is its minimum inter-arrival time. At each sporadic invocation, it will have a known arrival time $ar_{i,k}^s$ and an absolute deadline $d_{i,k}^s = ar^s + D_i^s$. The arrival time of $(k+1)^{th}$ invocation, $ar_{i,k+1}^s$, is $ar_{i,k+1}^s \geq ar_{i,k}^s + \lambda_i^s$.

The joint set of aperiodic and sporadic tasks, i.e. $\tau_s \cup \tau_a$, are called dynamically arriving tasks. The total number of tasks in the system is defined by $n = n_p + n_a + n_s$. The task model presented in this section is similar to the one proposed in [15].

3.4 Static system properties

Since all periodic task executions are captured in an off-line generated table which periodically repeats, all notions of time are modulo the hyper period of the periodic tasks. The hyper-period of intervals, $H_{\mathcal{I}}$, is the *least common multiple* (LCM) of the periods of the periodic task set, τ_p, i.e.

$$H_{\mathcal{I}} = LCM(T_i^p), where\ i = 1, 2, .., n_p. \tag{3.1}$$

A system $\mathcal{S}ys$ contains a set $\mathcal{I}$ of N intervals I_0, I_1, I_2, ..., I_{N-1}. Each interval, $I_m \in \mathcal{I}$, has a start time (st_m), an end time (e_m), a spare capacity (sc_m) and a set of periodic tasks with guaranteed time-slots (τ_m). It is represented as,

$$\mathcal{I} = \{I_m(st_m, e_m, sc_m, \tau_m), m = 0, 1, 2, ..., N-1\}. \tag{3.2}$$

The start time, st_0, of the first interval, I_0, is always zero, while the end of the preceding interval is considered as the start-time of the intervals:

$$st_m = \begin{cases} 0 & \text{if } m = 0 \\ e_{m-1} & \text{if } m > 0. \end{cases} \tag{3.3}$$

For all intervals, I_m, there exists a periodic task P_i, for which the end of an interval, e_m equals the deadline, $d_{i,k}^p$ of a job of a periodic task, $P_{i,k}$, . The deadline of this job, $P_{i,k}$, marks the

[1] This eases the bookkeeping for monitoring of execution times. In general, we only support one activation of a task at a time, so in case of larger deadlines (deadlines larger than periods) we assume that jobs are mapped onto different tasks.

end of an interval, i.e.

$$\forall I_m : \exists\ P_{i,k}\ \in\ \tau_m :\ d^p_{i,k} = e_m. \tag{3.4}$$

Definition An interval is uniquely defined by the start time and the end time denoted by Equation (3.3) and (3.4) respectively. Furthermore, an interval denotes disjoint time windows to track the spare capacity. Note that the intervals are not execution windows. The difference between an execution window and an interval is that jobs can execute in intervals prior to the ones they are assigned to [15].

The task-set belonging to an interval, τ_m, is defined as,

$$\tau_m = \{P_{i,k} |\ d^p_{i,k} = e_m\}. \tag{3.5}$$

A job of a periodic task belongs to an interval, if and only if the deadline of a job equals the end of an interval.

The spare capacity of an interval, sc_m, equals the length of interval I_m minus the sum of the activities assigned to it. A job, $P_{i,k}$, may start its execution earlier than the interval coinciding with its deadline, resulting in a negative spare capacity for an interval. The third part, *min*, takes into consideration the execution of tasks belonging to other (forthcoming) intervals [15]. The off-line derived spare capacity of an interval based on [15] is:

$$\forall I_m : j = (m+1)\ \textbf{mod}\ N : sc_m = e_m - st_m - \sum_{P_i \in \tau_m} C^p_i + min(sc_j, 0). \tag{3.6}$$

Note that the spare capacity is also defined for the final interval since per definition holds that $sc_0 \geq 0$.

According to [15], the number representing a spare capacity of an interval can be negative, while the amount of unused resources in that interval cannot be less than zero. The negative representation of spare capacity ensures that the tasks which are executed outside their intervals will not claim the same resources in their own intervals. During run-time, the negative spare capacity turns into positive, because of the borrowing mechanism proposed in [15]. The borrowing and lending mechanisms are described later.

The critical slot, t_c of an interval I_m is the time slot in I_m, such that if a dynamically arrived task arrives at t_c, its execution will be maximally delayed compared to all other slots in I_m due to the execution of the off-line scheduled tasks [15]. The critical slot is the slot after consumption of all spare capacity, so that the off-line guaranteed tasks must execute in order to make their deadline, i.e.:

$$t_c(I_m) = st_m + max(sc_m, 0). \tag{3.7}$$

The length of an interval, I_m, is,

$$Length(I_m) = e_m - st_m. \tag{3.8}$$

The pending work (PW) for a periodic task P_i at time t is represented as [2],

$$PW(P_i, t) = C_i^p - C_i^{p,completed}(t) \tag{3.9}$$

where, C_i^p is the Worst-Case Execution Time (WCET) of a task and $C_i^{p,completed}(t)$ denotes the amount of work already completed for a task P_i at time t.

Periodic and sporadic tasks are immediately placed in the ready queue upon arrival. Aperiodic tasks are only placed in the ready queue after passing an acceptance test, implemented by a guarantee algorithm. All tasks in the ready queue are sorted by their absolute deadline. The ready queue, Q, contains the following jobs at a time t:

$$Q(t) \quad = \quad \{P_{i,k} | ar_{i,k}^p \leq t \leq d_{i,k}^p\} \ \cup \ \{S_{i,k} | ar_{i,k}^s \leq t \leq d_{i,k}^s\} \ \cup \ \{A_i | ar_i^a \leq t \leq d_i^a\} \tag{3.10}$$

At the start of an interval, the reserved capacity, $C_{res}^m(st_m)$, for the periodic tasks equals the computation demand of periodic tasks belonging to the interval, I_m.

$$\forall I_m : C_{res}^m(st_m) = \sum_{P_i \in \tau_m} C_i - \sum_{\substack{P_i \in \tau_m \\ est_i \notin I_m}} C_i^{p,completed}(st_m). \tag{3.11}$$

Where, $C_i^{p,completed}(st_m)$ denotes the amount of work already completed for a periodic task P_i in the previous intervals.

Within an interval, I_m, the reserved capacity at time t, $C_{res}^m(t)$, for the periodic tasks equals the computational demand of periodic tasks belonging to the interval, I_m.

$$\forall I_m : st_m \leq t \leq e_m : C_{res}^m(t) = \sum_{P_i \in \tau_m} C_i - \sum_{P_i \in I_m} C_i^{p,completed}(t). \tag{3.12}$$

Equation 3.6 calculates the spare capacity of an interval with a slightly modified Equation (3.13)), which takes into consideration the amount of work already performed for periodic tasks belonging to an interval. The calculation of reserved capacity in Equation (3.12) takes into consideration the amount of work already performed for the task belonging to an interval.

The spare capacity of an interval, sc_m, is the remainder of the reserved capacity C_{res}^m plus the execution of early-arriving tasks belonging to the preceding interval(s).

$$\forall I_m : j = (m+1) \ \textbf{mod} \ N : sc_m = Length(I_m) - C_{res}^m(st_m) + min(sc_j, 0). \tag{3.13}$$

Note that the spare capacity is also defined for the final interval since per definition holds that $sc_0 \geq 0$. The **min** term considers the negative spare capacity of the preceding interval and reserve resources in the current interval for the execution of task(s) belonging to the preceding interval. If the spare capacity of the preceding interval is negative, then tasks belonging to the preceding interval will use the resources available in the current interval. This behavior is

[2]Since we assume implicit deadlines for periodic tasks, only a single job per task can be active so that we leave the job numbering implicit in the remainder of this document.

due to the early-arrival of a task. The run-time mechanisms of this phenomenon are called as the borrowing and lending mechanism. The borrowing and lending mechanism is described in 4.4.

Static versus dynamic spare capacity: The spare capacity is computed off-line using Equation (3.6), but during run-time (online) we use Equation (3.13) to compute the spare capacity before the start of an interval. During run-time, we take into account the amount of work already completed for a periodic task belongs to an interval. We therefore reallocate unused resources by periodic tasks to execute dynamically arriving tasks during run-time by using Equation (3.13).

Note: The static system properties presented in this section are applied to a case study, see Appendix A.

3.5 Interval invariants

With the properties described in Section 3.4, we define the invariants for the intervals in this section.

- The absolute start time of the interval should be greater than or equal to the end of previous interval. However, the start of the first interval is zero as indicated in Equation (3.3).

$$\forall I_m : \ st_m \geq e_{m-1} \tag{3.14}$$

- The spare capacity at the start of interval must be non-negative, i.e. all tasks should have sufficiently reduced their pending work:

$$\forall I_m : \ sc_m(st_m) \geq 0 \tag{3.15}$$

- The spare capacity at the end of interval must be zero.

$$\forall I_m : \ sc_m(e_m) = 0 \tag{3.16}$$

3.6 Summary

The assumptions and boundary conditions of our work is presented. The time model employed in the slot-shifting scheduling strategy is discussed. The mixed task set support of the slot-shifting scheduling requires dealing with tasks of different types. We presented the task model followed by the slot-shifting scheduling algorithm. Static system properties, which are derived from [15], are presented to formally understand the behavior of slot-shifting. We derived the run-time requirements from the static system properties. Invariants are also extracted from the static system properties.

Chapter 4

Slot-shifting revisited

In this chapter, the slot-shifting scheduling strategy is described in detail. In Section 4.1, the evolution of the slot-shifting scheduling strategy is discussed. In Section 4.2, the slot-shifting scheduling algorithm is explained with a case study. In Section 4.3, the resource reclaiming mechanism is presented with an example. In Section 4.4, the borrowing and the lending mechanisms of slot-shifting are explained. Section 4.5 presents the run-time mechanisms for the slot-shifting. Finally, this chapter ends with a summary in Section 4.6.

4.1 Evolution of Slot-shifting

The idea of combining off-line and online scheduling algorithms was proposed by Fohler [11] in 1995. The algorithm was designed to handle periodic and aperiodic tasks deployed on distributed nodes. The original idea provided rules to create intervals based on the deadline and the arrival time of the periodic tasks. The unused resources in each of these intervals are called *spare capacities*. By the end of his off-line scheduling part, the periodic task parameters leads to a set of intervals with spare capacities. During run-time, aperiodic tasks, tasks with unknown arrival patterns, are accepted based on the available resources. An aperiodic task can only be accepted, if it can make its deadline in the presence of off-line scheduled periodic tasks. Otherwise, an aperiodic task is not accepted. In this way, the original slot-shifting, proposed by Fohler [11] handled periodic and aperiodic tasks.

Later on, in the year 2009, Isovic and Fohler [15] extended the slot-shifting algorithm with the support for handling sporadic tasks along with periodic and aperiodic tasks. The enhanced slot-shifting algorithm guarantees periodic tasks in the same way as the original algorithm in [11]. In addition, sporadic tasks are guaranteed off-line by considering their worst-case arrival behavior at critical slots. In the worst-case, the sporadic tasks behaves similar to the periodic tasks, and therefore arrives periodically with period equals to its minimum inter-arrival time. By the end of of off-line scheduling part, the periodic task parameters yields information about intervals and spare capacities, while the sporadic tasks are guaranteed based on their worst-case arrival behavior. During run-time, the aperiodic tasks are guaranteed if and only if an aperiodic task can be accepted together with all the previously guaranteed periodic tasks, sporadic tasks and aperiodic tasks. If an aperiodic task cannot be accepted, then a flexible rejection strategy can be applied to reject either the newly arrived aperiodic task or (one or

more of) the previously guaranteed aperiodic task(s).

Unlike, the original slot-shifting algorithm, the enhanced slot-shifting algorithm reclaims unused resources during run-time for aperiodic tasks. Although, with the original approach, accepting an aperiodic task requires the creation of new intervals and spare capacities based on the deadline of a task, the enhanced slot-shifting approach allows handling of aperiodic task without creating new intervals during run-time. With the enhanced slot-shifting approach, resources are not explicitly reserved for an aperiodic task upon guarantee, as compared to the original approach proposed by Fohler. This leads to a flexible rejection strategy, as the task(s) can be rejected without affecting the number of intervals.

4.2 Slot-shifting - case study

Having discussed the evolution of the slot-shifting scheduling algorithm, in this section, the slot-shifting scheduling algorithm is described using a case study. We focus on the enhanced slot-shifting scheduling algorithm to handle with periodic, aperiodic and sporadic tasks. To better understand the behavior of the algorithm, the off-line and the online scheduling parts are presented separately[1].

4.2.1 Off-line preparation

The periodic task-set is the basis for the slot-shifting scheduling algorithm, as it provides information about the intervals and the unused resources in each of these intervals. The off-line scheduling part of the slot-shifting starts with the periodic task set. Let us consider the periodic task-set presented in the Table 4.1.

Tasks	**Release Time**	**Period**	**Comp. Time**	**Deadline**
P_1	0	20	1	3
P_2	4	20	1	8
P_3	8	20	1	10
P_4	10	20	1	14
P_5	10	20	2	20

Table 4.1: An example periodic task-set.

The task parameters provided in the Table 4.1 are essential to create intervals and spare capacities. The rules for the creation of intervals based on the periodic task-set is provided below:

- The start-time of an interval is the end of the previous interval. However, the start time of the first interval is always zero.

[1]This case-study describes the behavior of slot-shifting with a simple task-set. We refer the interested reader to Appendix A for a case-study with the borrowing and lending mechanisms.

- The end-time of an interval is identified by the deadline of a task. Hence, if a task is said to be *belongs to an interval*, then its deadline marks the end of that interval. Multiple tasks with the same deadline can belong to the same interval.
- The spare capacity of an interval is computed by calculating the amount of unused resources in that interval (see Section 3.4). This value can be negative to indicate that one or more tasks belonging to that particular interval must start their execution in the previous interval.

Based on the rule described above, the periodic task-set is used to create set of intervals. Table 4.2 presents the intervals for the periodic task-set presented in Table 4.1.

Interval	**Start**	**End**	**Spare Capacity**	**Task set**
I_0	0	3	2	$\{P_1\}$
I_1	3	8	4	$\{P_2\}$
I_2	8	10	1	$\{P_3\}$
I_3	10	14	3	$\{P_4\}$
I_4	14	20	4	$\{P_5\}$

Table 4.2: Intervals for the task-set presented in Table 4.1.

As we discussed earlier in Section 4.1, the sporadic task-sets are guaranteed off-line on top of the periodic tasks. Let us consider the sporadic task-set presented in Table 4.3. The sporadic task-set is guaranteed off-line based on the worst-case behavior as presented earlier in Section 4.1. In [15], the off-line guarantee algorithm for the sporadic task-set is presented. For the off-line guarantee, the sporadic tasks are assumed to be arriving at **the critical slot of an interval**. The critical slot of an interval is the slot where the execution of the dynamically arriving tasks are delayed due to the execution of off-line guaranteed periodic task. If a sporadic task-set can be scheduled at the critical slot of an interval, then it can be scheduled at any slot within the interval (for the detailed proof, see [15]).

Tasks	**Minimum Inter-arrival time**	**Comp. Time**	**Deadline**
S_1	20	1	9
S_2	20	1	16

Table 4.3: An example sporadic task-set.

Assuming the worst-case behavior, the sporadic task-set is accepted only if all tasks can make their deadline. If a sporadic task-set is failed, then either the sporadic task-set is changed or the entire system is redesigned by changing the periodic task-set to allow provision for a sporadic task handling. The off-line preparation details of sporadic tasks is explained in [15].

4.2.2 Online scheduling

The off-line preparation results in set of intervals and spare capacities, as illustrated in the previous section. Scheduling of tasks is performed based on the earliest-deadline-first (EDF)

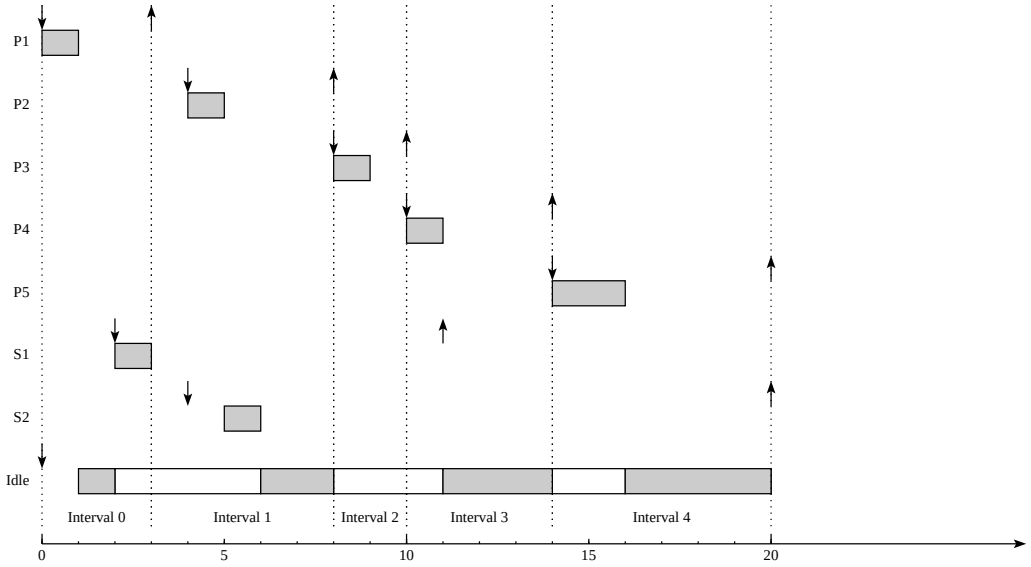

Figure 4.1: Case-study : Execution trace for one hyper period of the example task set for which the characteristics are presented in Table 4.1, Table 4.2 and Table 4.3.

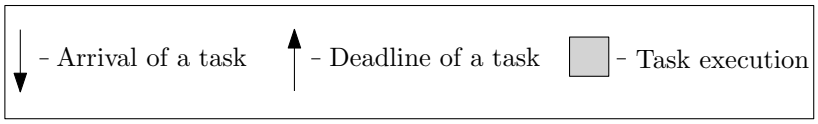

Figure 4.2: Legend for Grasp trace presented in Figure 4.1.

strategy. Tasks are allowed to execute based on their deadlines. If a periodic task executes in the interval to which it belongs, then the spare capacity of the interval is not affected. If the dynamically arriving tasks (sporadic and aperiodic) or periodic tasks which do not belong to the current interval execute, then the spare capacity of the interval is decremented. The execution of a periodic task can be shifted according to the EDF strategy any where within an interval, starting from its activation until its deadline (spanning all intermediate intervals), to allow leeway for the execution of dynamically arriving tasks.

However, the dynamically arriving aperiodic task cannot be accepted straightaway for the scheduling. An aperiodic task must undergo an guarantee test before it becomes ready for the scheduling. The purpose of the guarantee test is to verify the availability of the system resource for the execution of an aperiodic task. The result of the admission test depends on the availability of the system resource from the arrival of an aperiodic task until its deadline. If an aperiodic task can be accepted along with the off-line guaranteed periodic tasks, sporadic tasks and previously guaranteed aperiodic task(s), then the aperiodic task becomes ready for the scheduling. Otherwise, the newly arrived aperiodic task or the previously guaranteed aperiodic task(s) are rejected.

Figure 4.1 shows the execution trace of the task-sets presented in Table 4.1 and 4.3. The execution trace indicates the scheduling of tasks based on the deadline. Tasks are executed

within their intervals as presented in Table 4.2. The execution of tasks can be shifted anywhere within its interval to allow for the execution of dynamically arriving tasks. The off-line guaranteed sporadic tasks are executed with spare capacities of intervals.

4.3 Resource reclaiming

In the previous section, the scheduling of an example task set using the slot-shifting scheduling algorithm was presented. The off-line preparation part of the slot-shifting algorithm uses information about the worst-case execution time of the tasks. During the run-time, the task may consume less resources than the resources assigned for the worst-case execution. Typically, pessimistic assumptions on the execution time of a task leads to an over-provisioning of system resources. In order to utilize the unused resources for the execution of the dynamically arriving tasks, resource reclaiming is performed during the run-time. In [15], a resource- reclaiming mechanism is discussed with respect to the early completion of an aperiodic task. In this work, we further extended the resource-reclaiming mechanism for periodic and sporadic tasks, by deploying a monitoring mechanism.

4.4 Borrowing and lending of the spare capacity

In slot-shifted scheduling, the end of an interval is based on the deadline of a task and the start of an new interval can either be the arrival time of a task or the end of the previous interval. It is not necessary to have an interval with a start time equal to the arrival time of a task. The start time of an interval may also be the end of the previous interval, so that a task can arrive in an earlier interval or somewhere within the interval where it belongs to. If a task arrives in an earlier interval, than its own interval, the task may execute outside its own interval by consuming the spare capacity of earlier intervals. This phenomenon is called borrowing and the lending of the spare capacity.

4.5 Run-time mechanisms for Slot-shifting

In this section, we present the run-time mechanisms for slot-shifting. Subsection 4.5.1 discusses about the tracking of intervals. WCET monitoring is discussed in Subsection 4.5.2. Subsection 4.5.5 presents the run-time support for the guarantee algorithm. Scheduling of tasks is discussed in Subsection 4.5.6.

4.5.1 Tracking intervals

The slot-shifting scheduling strategy requires tracking of intervals. The periodic task set is transformed into a set of intervals. The basic idea for the creation of intervals based on the characteristics task set is already presented in Section 4.2. During run-time, task executions monitored to calculate the reserved and the spare capacities within an interval. To keep track of the spare capacity for handling dynamically arriving tasks, **the interval tracking** feature is desired.

4.5.2 WCET monitoring

The slot-shifting scheduling strategy requires information about the WCET and the deadline of a task. The WCET of a task is essential for correct functioning of slot-shifting and even more essential to perform the resource reclaiming.

4.5.3 Task attributes

The absolute deadline of a task is essential to perform scheduling based on the deadline, e.g. like EDF. To deal with a mixed task set, a type attribute is needed for tasks. A type of a task can be periodic, aperiodic and sporadic. This attribute is essential to differentiate tasks based on their type on arrival. The WCET of a task is added in a Task Control Block (TCB) to perform WCET monitoring. In order to keep track of a work performed for a task, an additional attribute (WorkToBeDone) is added in a TCB. These **task attributes** needs to be dynamically maintained during run-time to schedule tasks using the slot-shifting.

4.5.4 Spare capacity monitoring

The slot-shifting scheduling algorithm supports handling of mixed task sets. In the off-line preparation phase, the notion of intervals and spare capacities are derived from the periodic task set. During online, dynamic tasks (sporadic and aperiodic) or an idle task are executed with the spare capacity of an interval. In addition, a periodic task execution outside its own interval is treated similar to dynamic tasks, and the spare capacity is consumed by a periodic task. But, when a periodic task executes in its own interval do not consume the spare capacity of the interval, since the resources are reserved explicitly during the off-line preparation phase.

4.5.5 Guarantee algorithm

A guarantee algorithm is executed before admitting an aperiodic task into the run-time schedule. Sporadic tasks and a previously guaranteed aperiodic tasks can be affected if an aperiodic task is accepted without a guarantee algorithm. Thus, the guarantee algorithm takes into consideration the spare capacity of the interval at the time of the test, based on the previously guaranteed aperiodic tasks, and the current execution sporadic tasks and the arrival of sporadic task in the future that can interfere with the execution of the aperiodic task in test. If the aperiodic task cannot be accepted, then a rejection strategy should decide to reject either the newly arrived aperiodic task or one of the previously guaranteed aperiodic task(s). This guarantee algorithm is therefore essential to accept dynamically arriving tasks [15].

4.5.6 Scheduling

Within intervals, tasks are scheduled using the EDF scheduling algorithm. The **EDF scheduler** is therefore essential to schedule tasks based on the deadline. To our knowledge, the support for the EDF scheduling is not available in most of the COTS RTOSes, which are

present in the market today. The integration of slot-shifting in RTOS therefore requires an implementation of the EDF scheduler.

4.6 Summary

First, we discussed the evolution of slot-shifting, including the apparent improvements made by Isovic and Fohler [11] to support sporadic task execution with a dynamic resource-reclaiming mechanism. The principle of slot-shifting is presented with the help of an example task set. To better understand the combination of the off-line and online scheduling parts, the off-line preparation and the online scheduling were discussed separately. For efficient resource usage, a resource reclaiming mechanism is used in the slot-shifting scheduling algorithm. The information regarding how slot-shifting reduces the pessimism during run-time by the efficient resource reclaiming was presented: in order to deal with the early arriving tasks, a mechanism to borrow and lend spare capacities is presented.

From case-studies, the required run-time mechanisms for the integration of slot-shifting are identified, i.e. interval tracking, task management, WCET monitoring and (online) admission control. Tracking of intervals is necessary to manage resources during run-time. The slot-shifting scheduling enables handling of mixed task sets. The support for the multiple task types needs to be provided. To deal with the resource reclaiming and the borrowing mechanism, the WCET of a task is monitored during run-time. The online guarantee algorithm is essential to schedule dynamically arriving aperiodic tasks. Most importantly, scheduling of tasks within these intervals requires EDF scheduler. The implementation of an EDF scheduler is required for the integration of slot-shifting. In the next section we are going to introduce the preliminary operating system support provided by uC/OS-II, our target RTOS for including the mechanisms presented in this section for slot-shifting.

Chapter 5

The μC/OS-II and OpenRISC 1000

μC/OS-II is a completely portable, ROMable, scalable, preemptive, real-time, multitasking kernel [18]. μC/OS-II is written in the C programming language and is compliant with the American National Standards Institute (ANSI) - called ANSI C. A small part of the kernel is written in assembly code for compatibility with different processor architectures. This operating system is widely used in different application domains, such as cameras, avionics, medical instruments, and more. The source code of the μC/OS-II is approximately 5500 lines. For more information on the Application Programming Interface (API) of μC/OS-II, see [18].

In this Chapter, the μC/OS-II RTOS is discussed briefly. Section 5.1 presents the task management features, while the timer management features are discussed in Section 5.2. The scheduling of tasks in the μC/OS-II is described in Section 5.3. Section 5.4, the extensions are presented briefly. OpenRISC 1000 hardware simulator platform details are presented in Section 5.5. Measurement details are presented in Section 5.6. Grasp visualization tool is briefly described in Section 5.7.

5.1 Task management

In RTOS, applications are divided into a small number of concurrent functional units, called tasks. μC/OS-II manages each individual tasks via a Task Control Block (TCB). When a task is created, it is assigned a TCB, where during runtime, the state of that task is recorded, e.g. its context information when it is preempted. Tasks must be created using either the OSTaskCreate() or OSTaskCreateExt() function provided by the μC/OS-II kernel. The OSStart() function initializes the operating system. μC/OS-II requires at least one task to be created before calling the OSStart() function.

5.2 Timer management

A periodic timer is responsible for keeping track of the time delays and timeouts in μC/OS-II. This timer can be configured to produce a constant number of tick per second varying between 10 and 100 times per second. The faster the tick rate, the more overhead μC/OS-II

imposes on the system [18]. The OSTimeTick() function is called at each tick interrupt and keeps track of task delays and timeouts. The tick Interrupt Service Routine (ISR) execution time of μC/OS-II depends on the number of tasks in the application. The tick execution time increases linearly, because in each ISR execution all tasks are traversed and their delay value is decremented.

5.3 Scheduling

The scheduler is responsible for selecting a task for execution. μC/OS-II uses a fixed-priority scheduler. The operating system always executes the highest priority task ready to run. The OS_Sched() function is invoked to perform scheduling at the task-level. At the ISR-level, the OSIntExit() function is responsible for the scheduling of tasks. The dispatching overhead is constant in μC/OS-II irrespective of the number of tasks created in an application.

5.4 Proprietary support for relative timed-events

A real-time applications may need to manage multiple real-time events, which are difficult to realize by the basic support of a delay-statement (i.e. task self-suspension) as provided by the μC/OS-II API. For example, implementing periodic tasks by means of delays may cause drift due to jitter in the execution times of tasks. RELTEQ is such a timed-event management system targeted at small embedded systems, because it has a small timer representation and reduces the overhead for handling many timed events compared to the default μC/OS-II setup [12]. The basic idea of RELTEQ is to store timed events - called timers - relative to each other by expressing the arrival time of the timer relative to the expiration of the previous timer [28]. An example of such a setup is shown in Figure 5.1.

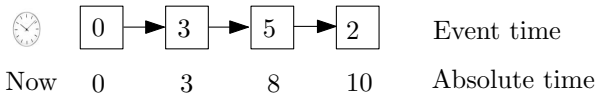

Figure 5.1: Time representation in RELTEQ.

When the head of the RELTEQ expires, the timer is popped from the queue and an event handler for that specific event type is executed. This event handler may on its turn manipulate a task ready queue or the RELTEQ timer queue itself, for example by releasing a task or inserting a new timer to plan a new event in the future. On top of this support for timed-events, several scheduling extensions have been presented for μC/OS-II, e.g. periodic-task support and two-level reservation-based scheduling [8, 12]. A more detailed description of RELTEQ and its functionalities can be found in [12]. In addition, we give a state-of-the-art overview of the API of RELTEQ in Appendix B, including some undocumented features.

5.5 OpenRISC 1000 port

In this work, the OpenRISC 1000 simulator maintained by OpenCores [3] is used. It is a completely free and open architecture [2]. The size of the registers of the OpenRISC processor is 16 bits. The OpenRISC 1000 simulator contains an open-source Reduced Instruction Set Computing (RISC)-based architecture and a GNU's Not Unix (GNU) compiler tool-chain to develop applications in the C language. The hardware simulator is cycle-accurate, and the performance of the simulator is not affected by the load of the host operating system, which makes it suitable to perform measurements for real-time systems. The versions of the tools that we used for our performance analysis of μC/OS-II are available in Appendix D.

5.6 Profiling

The OpenRISC platform has a cycle-count register, which allows to profile the code at the accuracy of a few processor cycles. μC/OS-II is extended with the support for the profiling of the code [8]. The output of the profiler is dumped into a text file. In all our measurements, the OpenRISC simulator was running at 10 Mhz with a tick frequency of 10 ms.

5.7 Grasp extension

Grasp is the visualization tool intended for the real-time system applications [13]. The visualization feature can be added through a logging mechanism in a RTOS. The visualization of slot-shifting requires further an extension to Grasp and the logging mechanisms, i.e. the events specific to slot shifting are supported within Grasp as part of this work. The new mechanisms for visualizing intervals and spare capacities can be derived from the server visualization features of the Grasp. A more detailed description of the visualization tool, Grasp, can be found in [13]. Finally, this chapter is concluded with a summary.

5.8 Summary

We described our target RTOS, μC/OS-II, briefly in this chapter. We presented the task and timer management features. The scheduling algorithm supported by the standard μC/OS-II is also discussed briefly. The RELTEQ extension to μC/OS-II manages timed-events. We introduced the RELTEQ framework and gave an overview of its run-time behavior.

In this work, we use OpenRISC hardware simulator to perform measurements in μC/OS-II. For the visualization of tasks, we use the Grasp visualization tool, which works with the logging feature in μC/OS-II.

Chapter 6

Design and implementation considerations for Slot-shifting

Having defined the run-mechanisms for the slot-shifting algorithm, we present our design of a corresponding implementation in this chapter. An interval tracking mechanism is presented in Section 6.1. Section 6.2 presents TCB extensions. The support for mixed type of tasks is presented in Section 6.3. Section 6.4 discusses WCET monitoring. A spare capacity monitoring is discussed in Section 6.5. Scheduling of tasks within intervals is presented in Section 6.6. The implementation considerations of the online guarantee algorithm is presented in Section 6.7.

The run-time behavior of slot-shifting requires dealing with timed events. We present the event-handlers associated with the slot-shifting scheduling algorithm in Section 6.8. Slot-shifting configuration details are presented in Section 6.9. Finally, this chapter is concluded with a summary.

6.1 Tracking intervals

The off-line scheduling part of slot-shifting converts task sets into intervals with spare capacities, i.e. the task-set presented in Table 6.1 cannot be directly used online by the slot-shifting scheduling algorithm before an off-line scheduling method is applied on the task-set to prepare it for online scheduling. The purpose of the off-line preparation in slot-shifting is to resolve constraints such as precedence relation between tasks and to create intervals based on the task-set.

The result of an off-line schedule is a set of intervals with spare capacities. This set of intervals for the task-set presented in Table 6.1 is given below in Table 6.2. Table 6.2 provides information about the beginning and the end of intervals and their spare capacities. This information must be maintained by the operating system to schedule tasks. A new data structure is therefore required to store these information. An Interval Control Block (ICB) is created to store the interval-associated attributes. The start of an interval (st_k), the end of an interval (e_k) ,the spare capacity of an interval (sc_k) , the number of tasks belongs to an interval (n) and a task-set (contains n tasks) belong to an interval (τ_k) are stored in an ICB.

Tasks	Release Time	Period	Comp. Time	Deadline
P_1	0	15	1	3
P_2	4	15	1	8
P_3	8	15	1	10
P_4	10	15	1	15
P_5	10	15	2	15

Table 6.1: An example periodic task-set.

Interval	Start	End	Spare Capacity	Task set
I_0	0	3	2	$\{P_1\}$
I_1	3	8	4	$\{P_2\}$
I_2	8	10	1	$\{P_3\}$
I_3	10	15	2	$\{P_4,P_5\}$

Table 6.2: Intervals for the task set presented in Table 6.1.

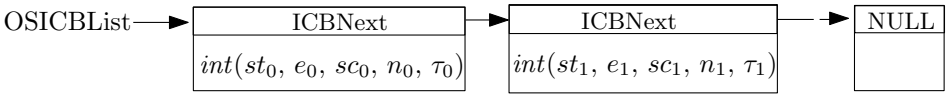

Figure 6.1: Single linked list of interval control blocks.

The complete off-line schedule is represented by a list of ICBs. Such a list of ICBs are shown in Figure 6.1. An ICB is similar to a TCB for tasks in μC/OS-II RTOS [18]. The use of a NULL at the end of the list eases the iterations through the list. For instance, if m number of intervals are in the application, then we always create *m+1* intervals. An additional ICB with the NULL pointer is used to denote the end of the ICBList. Such a NULL-node in a tree or list data structure is called a *sentinel*.

Variable name	Variable type	Notation	Explanation
OSICBStartTime	INT16U	st_k	Start of an interval
OSICBEndTime	INT16U	e_k	End of an interval
OSICBSpareCapacity	INT16S	sc_k	Spare capacity of an interval
OSICBNumberOfTasks	INT8U	n	Length of a task-set array
OSICBTaskSet	Pointer to TCBs	τ_k	Array of tasks
OSICBNext	Pointer to ICB	–	Pointer to next ICB

Table 6.3: Static ICB attributes.

Initialization procedure of an ICB: The ICB attributes are statically initialized as follows:

1. *OSICBStartTime:* The start time of an interval is defined by Equation (3.3). The start-time of an interval is the end-time of the previous interval. The start-time of the first interval is zero.

2. *OSICBEndTime:* The end time of an interval is defined by the deadline of a task, see Equation (3.4).

3. *OSICBSpareCapacity:* The spare capacity of an interval is calculated using Equation (3.6).

4. *OSICBNumberOfTasks:* This variable contains the number of tasks belongs to an interval.

5. *OSICBTaskSet:* This array contains pointers to tasks belongs to an interval. The length of an array is defined by OSICBNumberOfTasks. Tasks with the deadline at the end of an interval is stored in an array, see Equation (3.5).

Having defined these intervals, the scheduling of intervals is the next step. One way of scheduling intervals is by using a *timer* or *counter*. We present a counter-based interval scheduling method in Pseudo code 1 from line-11 to line-36. At the initialization time of the OS, the OSICBCur variable is updated to point to the sentinel. The OSICBCur is validated at line-23 and the pointer is updated to point to the first ICB of a list each time, when the hyper period repeats. The slotCounter variable is increased at each time-tick[1], and this variable is used for the tracking of intervals.

As indicated in Pseudo code 1 in line-29, the slotCounter value is compared with the start time of an interval to identify the start of an interval. At the start of each interval, the spare capacity of the interval is calculated and the **cursparecapacity** variable is updated as indicated in line-31. The spare capacity of an interval is calculated by identifying the resource requirement of periodic tasks belonging to an interval. The resources unused by periodic tasks, called spare capacity of an interval, are used for dynamically arriving tasks. In Section 6.8.1, the spare capacity computation is described in detail.

Next in line-16 to line-19, the slotCounter variable is compared with the end-time of an interval to identify the end of an interval. At the end of each interval, the OSICBCur is updated to point to the subsequent interval. At the end of the ICB list (indicated by NULL), the OSICBCur is updated to point to the first ICB of a list. The logStartInterval() and logEndInterval() routines are used for the visualization of intervals using Grasp. More detailed description of the interval start and interval end event handlers are presented in Section 6.8.

[1]We use terms tick and slot interchangeably to refer to a time slot.

Pseudo-code 1 Scheduling-Interval within the tick ISR

```
1: /* WCET monitoring begins here */
2: if OSPrioCur != OS_TASK_IDLE_PRIO then
3:     OSTCBCur → OSTCBWorkToBeDone−−;
4: end if
5: /* WCET monitoring ends here */
6:
7: /* Spare capacity monitoring starts here */
8: if OSTCBCur → OSTCBInterval ≠ OSICBCur → OSICBId or OSPrioCur == OS_TASK_IDLE_PRIO
   then
9:     cursparecapacity−−;
10: end if
11: /* Spare capacity monitoring ends here */
12:
13: /* Interval tracking begins here */
14:
15: /* OSICBCur points to NULL at the system start and the Counter is initialized to 0 in the initMisc()
    function before OSStart is called */
16:
17: /* Begin: Interval end event (Handler is described in Section 6.8.2) */
18: if OSICBCur ≠ NULL and OSICBCur → OSICBEndTime == slotCounter then
19:     logEndInterval();
20:     OSICBCur = OSICBCur → OSICBNext;
21: end if
22: /* End: Interval end event */
23:
24: /* Resetting counter and ICBPointer */
25: if OSICBCur == NULL then
26:     slotCounter = 0;
27:     OSICBCur = & OSICBList[0];
28: end if
29:
30: /* Begin: Interval start event (Handler is described in Section 6.8.1) */
31: if OSICBCur → OSICBStartTime == slotCounter then
32:     logStartInterval();
33:     cursparecapacity = GetSpareCapacity(OSICBCur);
34: end if
35: /* End: Interval end event */
36:
37: slotCounter++;
38: /* Interval tracking ends here */
```

6.2 Task attributes

The TCB of a task is extended with additional attributes for the slot-shifted scheduling support. Table 6.4 presents the TCB extensions.

Variable name	Variable type	Notation	Explanation
OSTCBType	INT8U	-	Type of a task
OSTCBDeadline	INT16U	D	Relative deadline of a task
OSTCBCompTime	INT16U	C	WCET of a task
OSTCBWorkToBeDone	INT16U	PW	Pending work of a task
OSTCBInterval	INT8U	-	Interval of a task

Table 6.4: Extended TCB attributes for slot shifting.

The type of a task, the WCET of a task, and the deadline of a task are static variables, as they do not change during run-time. The pending work of a task and the interval of a task are dynamic variables, as they change dynamically during run-time.

Initialization procedure for TCB attributes: TCB attributes associated with slot-shifting scheduler are initialized as follows:

1. *OSTCBType:* This static variable holds the type of a task. The type can be periodic, aperiodic or sporadic.
2. *OSTCBDeadline:* The relative deadline of a task stored in this static variable. This variable is used to schedule tasks based on their computed absolute deadline, see Pseudo-code 7.
3. *OSTCBCompTime:* The WCET of a task is stored in this static variable. This variable is used to monitor the execution of a task.
4. *OSTCBWorkToBeDone:* The OSTCBWorkToBeDone of a job is a dynamic variable. This variable is used to keep track of the execution of a job. This variable is initialized as follows: OSTCBWorkToBeDone = OSTCBCompTime+1, see Section 6.4 for more details on WCET monitoring.
5. *OSTCBInterval:* The OSTCBInterval is a dynamic variable. This variable is used to store the interval to which a task belongs to. This variable is updated inside the interval arrival event handler, see Pseudo-code 8.

6.3 Mixed task support

Having created tasks with the *type* parameter, the task should be placed in a ready queue upon arrival, possibly after the acceptance procedure. We use periodic timers for all the task types. When the periodic events expire, we place tasks in the ready queue based on its type. As shown in Figure 6.2, periodic and sporadic tasks are placed in the ready queue for the scheduling, while, aperiodic tasks are undergoing an online admission test to guarantee

resources for the execution. Periodic and sporadic tasks do not need to undergo an online admission test since, they are guaranteed resources off-line.

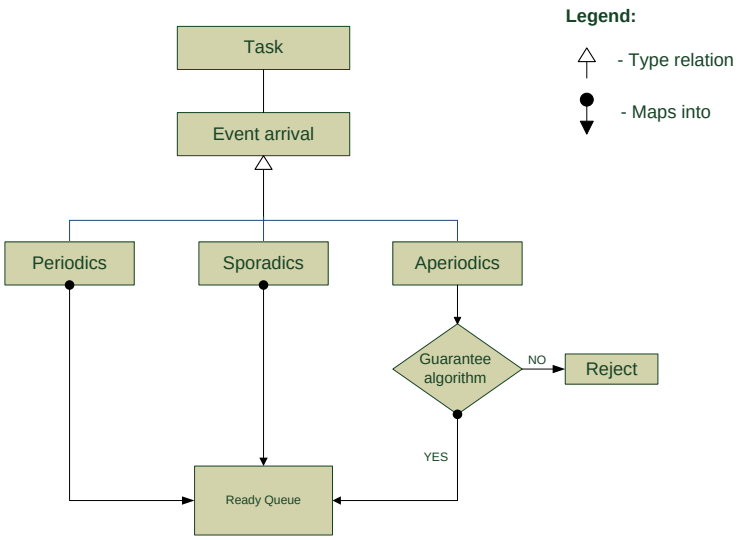

Figure 6.2: Mapping of tasks into queues on arrival.

6.3.1 Periodic task handling

On arrival of a periodic task, OSTCBWorkToBeDone variable is initialized as presented in Section 6.2. In addition, OSTCBInterval variable is updated with a interval identifier, see Pseudo code 5. An interval associated with a job is essential to perform resource reclaiming at the job completion. In Pseudo-code 8, the resource reclaiming is performed only if a completing job belongs to an active interval. This is verified with a if statement presented in line-9 to line-11 in Pseudo-code 8.

The resource reclaiming of periodic tasks has two cases.

1. **Execution outside its own interval:** When a periodic task completes its execution outside its own interval, the task is said to have borrowed spare capacity from neighboring intervals. In this scenario, the reserved resources are reclaimed at the start of its own interval to execute dynamically arriving tasks.

2. **Execution within its own interval:** When a periodic task executes in its own interval, then the spare capacity of an interval is not affected unless the execution time of a task equals its WCET. When the task completes earlier than its WCET, then the unused resources by the task are reclaimed and added to the spare capacity of the current interval to deal with dynamically arriving tasks. When the execution demand of the task exceeds its WCET, then the application is terminated with a task misbehavior message.

During run-time, our resource reclaiming mechanism is better than the approach presented in [15], because we also reclaim unused spare capacity of early completing periodic tasks. The remainder of the resources are used to execute dynamically arriving tasks selected by the EDF scheduler.

6.3.2 Aperiodic task handling

An aperiodic task is typically triggered by an event that wakes up the task. One way to develop an aperiodic task is by waiting for an semaphore or other similar events. When a task is blocked, it can be activated by another task (or ISR), which releases the semaphore (S_p), see Pseudo code 2 and Pseudo code 3. It is the conventional way of making an aperiodic task. In this work, an aperiodic task can become ready only after it is accepted by the guarantee algorithm. We add an additional semaphore S_g as shown in Pseudo code 2 and Pseudo code 3, which is released by the guarantee algorithm. If an aperiodic task is accepted by the guarantee algorithm, then the semaphore (S_g) is released. Otherwise, an aperiodic task is rejected.

Pseudo-code 2 An aperiodic task

```
1: /* Aperiodic task */
2:
3: WaitForSemaphore(S_g);
4: Dosomething();
5:
```

Pseudo-code 3 An invocation of guarantee test within a task or an ISR

```
1: /* Task or ISR */
2:
3: flag = Perform_acceptance_test();
4: if flag == True then
5:     ReleaseSemaphore(S_g);
6: end if
```

In this work, we have not implemented a online guarantee algorithm for aperiodic tasks. Assuming an aperiodic task is guaranteed, the run-time for support for the scheduling of aperiodic tasks are present already. In slot-shifting scheduling, tasks are scheduled based on the deadline, and the insertion of an aperiodic task into the ready-queue is decided by a guarantee algorithm. The run-time support for an aperiodic task is therefore present in our implementation of slot-shifting in μC/OS-II. Perform_acceptance_test() function presented in Pseudo code 3 is not implemented in this work.

6.3.3 Soft aperiodic task handling

In this work, we have not considered soft aperiodic task handling as indicated in [15]. A soft aperiodic task is a task without a deadline. We execute an idle task when there are no other tasks ready to execute in the system. The idle task can be seen as a soft aperiodic task on itself. We therefore propose an implementation of a policy in the idle task to handle soft aperiodic requests. However, we propose such implementation as a future work.

6.3.4 Sporadic task handling

For a sporadic task, the combination of event arrival and the minimum inter-arrival time is essential to identify the event as a sporadic event. The event that arrives after the minimum inter-arrival time can only activate the sporadic task. The sporadic task can be developed by using the similar method as presented in Pseudo code 2 and Pseudo code 3, but with an additional *minimum inter-arrival time* enforcement mechanism, rather than an acceptance test. This is an online enforcement mechanism, i.e. we enforce minimum inter-arrival time to prevent overload in case of a task misbehavior. The online enforcement mechanism is presented in [25] for enforcing the period of suspending periodic tasks and has been included in RELTEQ [28]. The period enforcement presented by [25] is similar to using guards, i.e. task arrivals that do not satisfy the minimum inter-arrival time are delayed. In this work, we do not implement further the implementation of a sporadic tasks with the *minimum inter-arrival time* enforcement. In this work, we implement sporadic tasks as periodic tasks for test purposes, but our implementation of the resource-reclaiming mechanisms are compliant with concepts described in [15].

6.4 WCET Monitoring

The WCET monitoring feature is essential for the functional correctness of slot-shifting, as the spare capacity of an interval is decided by the execution demand of tasks belonging to an interval. In addition, WCET monitoring is used to reclaim unused resources during run-time. For WCET monitoring, we added new attributes such as OSTCBCompTime and OSTCBWorkToBeDone in a TCB. The OSTCBCompTime stores the WCET of a task, and it is a static variable. The OSTCBWorkToBeDone variable is used to monitor the execution of a task. This variable is updated dynamically during the execution of a task.

The OSTCBWorkToBeDone variable is initialized with OSTCBCompTime+1 on arrival of a job. OSTCBWorkToBeDone is decreased in the tick ISR as indicated in line-2 in Pseudo code 1, before a job uses the slot. The early completion of a job is accounted to the completing job. The newly arriving job does not need to pay for this remainder. Based on two decisions, (1) we prevent variable OSTCBWorkToBeDone to become negative and (2) jobs pay for a slot before consumption, we initialize the OSTCBWorkToBeDone with the WCET of a task plus one additional time slot. The application terminates with a task misbehavior message, when OSTCBWorkToBeDone touches zero.

With the borrowing and lending mechanism, the earlier arriving tasks may execute outside its own interval. In an interval, the resources are allocated for periodic tasks based on the execution demand of tasks. The OSTCBWorkToBeDone variable provides the resource requirement for a task, as it maintains the pending work for a task (see Equation (3.11) in Section 3.4). If the OSTCBCompTime variable is used for the computation of spare capacity at the start of an interval, then resources may be allocated in excess. We avoid the over-provisioning of resources for periodic tasks at the start of an interval by using OSTCBWorkToBeDone variable. This mechanism is presented in the GetSpareCapacity() function described in Pseudo code 5.

When a task completes its execution earlier than its worst-case execution demand, the unused resources are added with a spare capacity of an interval to handle the dynamically arriving

tasks. When a task completes its execution, the OSTCBWorkToBeDone variable is added to a cursparecapacity to reclaim unused resources. In Section 6.8.4, We described this functionality in the task completion event handler.

6.5 Spare capacity monitoring

The spare capacity of an interval is affected by the execution of dynamically arriving tasks (aperiodic and sporadic) or the execution of a periodic task outside its own interval. In the absence of such tasks, the spare capacity of an interval is consumed by an idle task. The spare capacity monitoring is performed in the tick ISR. In Pseudo code 1, the spare capacity monitoring is presented in line 5 to line 9. The interval identifier is stored in a TCB to identify the interval of a job. If the currently executing job does not belong to the current interval or the currently executing task is an idle task, then the spare capacity of an interval is decremented, as indicated in line 9 of Pseudo code 1.

6.6 Scheduling

The slot-shifting scheduling strategy uses an EDF scheduler to schedule tasks. The EDF scheduling requires up-to-date information about the *absolute* deadline of a task to perform the scheduling. The absolute deadline of a task is stored in a TCB. μC/OS-II only supports scheduling based on a fixed priority for each task by means of a ready-table. For scheduling of tasks based on their deadline, we require a new scheduling mechanism. We use a RELTEQ based ready-queue to schedule tasks based on the absolute deadline. More information on RELTEQ can be found in Appendix B and [12].

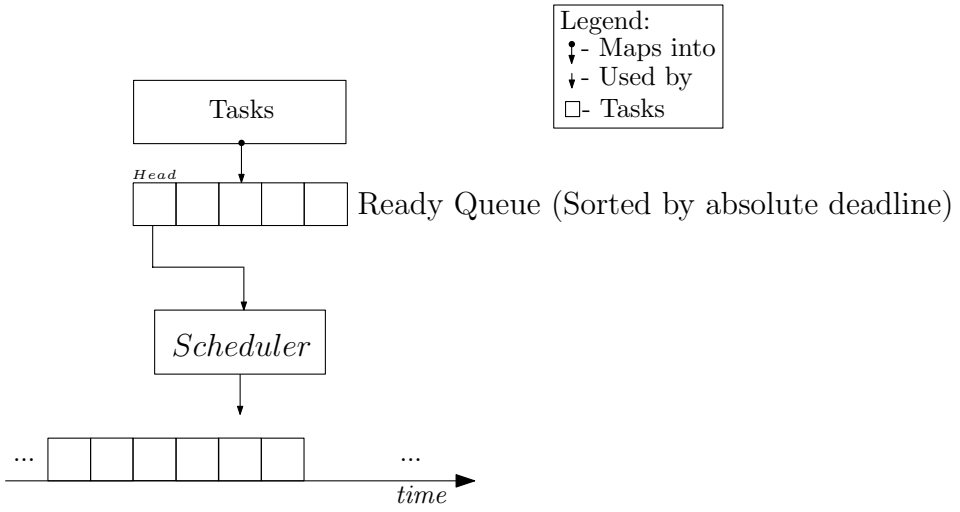

Figure 6.3: Design of the EDF scheduler.

Figure 6.3 shows the design of the EDF scheduler. Upon arrival, tasks are placed in the ready queue. The ready queue based on the RELTEQ framework is sorted based on the

deadline. The feature of sorting the queue based on the absolute event-time with respect to the current system time is provided by the RELTEQ. This feature makes the handling of ready queue easier. Figure 6.3 depicts the insertion of tasks into the queue. Although, only the deadline corresponding to a task is inserted in the ready queue, we make a one-to-one mapping from active tasks to events in the ready queue. Upon task arrival, a deadline event is created and inserted into the ready queue. Without violating the EDF-policy, tasks with the same absolute deadline are served on a first-come-first-serve basis. Aperiodic tasks are placed in the ready queue only after it is accepted by the guarantee algorithm. The job of the scheduler is to pick a task from the head of the ready queue for the execution.

6.7 Online guarantee algorithm

To facilitate an implementation of the guarantee algorithm presented in [15]. we need to keep track of the following run-time parameters:

- The spare capacity of the currently executing interval. This information stored in the ICB of each interval.
- The previously guaranteed aperiodic tasks, which are ready to execute. These can be accessed from the ready queue.
- The OSTCBCompTime (WCET) and OSTCBWorkToBeDone of the aperiodic and sporadic task. These are stored in TCB of the tasks.
- The minimum inter-arrival time of a sporadic task. It is stored in the TCB of a sporadic task.

This guarantee algorithm can be invoked at the beginning of each slot within the tick interrupt, after checking for the arrival of aperiodic task. However, this could potentially lead to longer tick ISR, hence large tick interrupt overhead. Instead, we propose to reserve capacity for guarantee algorithm by means of a sporadic task with a minimal inter-arrival time. The down side of this is that it may reduce the granularity of the aperiodic tasks due to the delayed acceptance test.

We have added all the attributes required to perform a guarantee test online. However, the implementation of a online guarantee test is considered as a future work.

6.8 Event Handlers

In this section, we will describe how to maintain the scheduling properties of slot shifting during execution by means of handling event timers. We use assertions to check the system invariants from Section 3.5 and to guarantee correct execution of the scheduler. The *assert* expression should always be true. When the expression evaluates to false, the system will terminate with an error.

The event handlers associated with the slot-shifting scheduling strategy are presented below:

1. Interval arrival event

2. Interval end event
3. Task arrival event
4. Task completion event

First three event-handlers are executed in the context of tick ISR, while the task completion event is executed in the context of a task.

The interval arrival event and the interval end event are the static properties of the slot-shifting scheduling algorithm, as the arrival-time of these events are computed off-line. The RELTEQ extension to μC/OS-II discussed in Section 5.3 is used to handle timed events. The RELTEQ extension is not preferred for interval-associated events, as their run-time arrival pattern is stored in the form of a off-line schedule in a list of ICBs. Due to the static nature of the interval events, we followed a counter based approach to handle with the interval arrival and the interval end events.

The dynamic nature, unknown arrival patterns of tasks, e.g. sporadics, of the task arrival events leads to the use of RELTEQ, as the RELTEQ supports efficient timer management with a long inter-arrival time and a low drift [12]. Moreover, the implementation of a EDF scheduler requires a ready-queue, sorted based on the deadline of a task, which comes for free with RELTEQ. The task completion event is executed in the context of a task.

6.8.1 Interval arrival event

At the start of an interval, an event handler is executed to update the spare capacity, in which dynamically arriving tasks may execute within the upcoming time interval.

This event signals the start of an interval. The actions performed in this event handler are presented in Pseudo-code 4. The spare capacity is calculated at the start of an interval based on the up-to-date resources that are still required by the off-line guaranteed periodic tasks. The off-line computed spare capacity is not used during run-time, as the use of off-line computed spare capacity might lead to over-provisioning of resources during run-time. Instead, we take into the account the amount of work already completed for periodic tasks belonging to an interval, at the start of an interval.

The unused resources at the start of an interval is computed using the static system property presented in Equation (3.13), i.e. the GetSpareCapacity() function computes the spare capacity of an interval by taking into consideration the amount of work already performed for periodic tasks belonging to an interval, the early-arriving task(s) belonging to the preceding interval and possibly the negative spare capacity of the next interval.

Pseudo-code 4 Interval arrival event

```
1: /* Invoke the GetSpareCapacity() function to update the current spare capacity */
2: /* GetSpareCapacity() function is defined in Pseudo code 5 */
3:
4: CurSpareCapacity = GetSpareCapacity();
5:
6: logStartInterval();
```

The GetSpareCapacity() function is defined in Pseudo code 5. As we discussed in Section 3.4, the tasks are mapped into intervals based on their deadlines. For each interval, the

computation of the spare capacity requires information about the amount of unused resources. The information about the amount of resources reserved for the execution of the off-line guaranteed periodic tasks is essential for the computation of the spare capacity. Line-4 to Line-8 in Pseudo code 5 calculate the amount of resources needed for the execution of off-line guaranteed periodic tasks. The OSTCBWorkToBeDone holds the amount of work to be done for the periodic task. With the information about the reserved resources, the spare capacity is calculated by subtracting the reserved resources from the length of the interval. This operation is performed in line-10. The computed value is returned by the GetSpareCapacity() function, if the subsequent interval contains sufficient resources to handle the off-line guaranteed periodic tasks.

However, in certain cases, the resource requirement of the off-line guaranteed periodic tasks may exceed the amount of resources available in an interval. This scenario can occur due to the early arrival of a periodic task. For instance, a periodic task belonging to the interval I_n can arrive in any of the earlier intervals, I_{n-1}, I_{n-2} etc. The borrowing and lending mechanism of the slot-shifting provides the possibility of dealing with the tasks which are arriving earlier than their mapped interval. Due to the borrowing and the lending of spare capacities between intervals, the off-line computed spare capacity of the interval may be negative before execution within this interval actually starts. Hence, when we calculate the spare capacity of an interval, we also need to check the spare capacity of the subsequent interval. When the spare capacity of the subsequent interval is negative, then borrowing of resources from the subsequent interval is taken into consideration for the spare capacity calculation. This operation is performed in line-14 to line-16, see Equation (3.12) and (3.13) in Section 3.4.

Pseudo-code 5 GetSpareCapacity()

```
 1: /* OSICBCur points to current interval */
 2: /* spare and res variables are initialized to 0 at the start */
 3:
 4: for i = 0 to OSICBCur → OSICBNumofTasks-1 do
 5:     res = res + OSICBCur → OSICBTaskSet[i] → ÕSICBWorkToBeDone;
 6:     OSICBCur → OSICBTaskSet[i] → OSTCBInterval = OSICBCur → OSICBId;
 7:     i = i + 1;
 8: end for
 9:
10: spare = (OSICBCur → OSICBEndTime - OSICBCur → OSICBStartTime) - res;
11:
12: /* If the spare capacity of the next interval is negative, then the resources are reserved in the current
    interval considering the earliest start time of a task belonging to the next interval */
13:
14: if OSICBCur → OSICBNext ≠ NULL and OSICBCur → OSICBNext → OSICBSpareCapacity < 0 then
15:     spare = spare + OSICBCur → OSICBNext → OSICBSpareCapacity;
16: end if
17:
18: return spare;
```

The first interval I_0 can not have a negative spare capacity at any moment in time, because this would mean that we have an invalid startup condition, i.e. the first execution when starting the system would have insufficient resources to complete all allocated workloads. It is therefore safe to ignore the spare capacity beyond the boundary of the hyper period (see the if-statement in line 14).

6.8.2 Interval end event

This event signals the end of an interval. The actions performed in this event handler are presented in Pseudo-code 6.

Pseudo-code 6 Interval end event

```
1: /* The spare capacity at the interval must be zero */
2: ASSERT(cursparecapacity == 0)
3:
4: logEndInterval();
```

At the end of an interval, the *assert* expression presented in line-2 should always be true. The expression in the assert can fail due to task misbehavior or other system malfunction. However, mechanisms for overload management are beyond the scope of this work. All resources in the interval should therefore be consumed; if there is no pending workload, then it should be idled away. This assertion is essentially a validity check for the resource-monitoring mechanism.

6.8.3 Task arrival event

The task-arrival event handler is meant for the periodic and sporadic tasks. Aperiodic task arrivals which requires a guarantee test, are not discussed in the Pseudo-code 7.

Pseudo-code 7 Task arrival event

```
1: task → OSTCBWorkToBeDone = task → OSTCBCompTime + 1;
2:
3: /* Create an absolute deadline event of the type kRelteqEventDeadline */
4: RelteqEvent* deadline-event = RelteqEventCreate(kRelteqEventDeadline, task);
5: /* Insert the deadline event into the ready queue */
6: RelteqQueueInsertEvent(RelteqEDFReadyQueue, deadline-event, task → OSTCBDeadline, &err);
7: ASSERT_OK(err);
8: task → ReadyQueueEvent = deadline-event;
```

Upon task arrival, the completed work variable is updated with the worst-case computation time of the task. The completed work variable is used to keep track of the task execution. This variable is used for the resource reclaiming and the borrowing, lending of the spare capacity. The slot-shifting scheduling algorithm uses EDF to schedule tasks. The RELTEQ framework is used to implement the EDF scheduler. We extended μC/OS-II using RELTEQ to support EDF scheduling. For this purpose, we introduce the notion of deadline events. A deadline-event is inserted into the EDF ready-queue with an absolute deadline of a task (see line-4). The events are sorted by absolute deadlines and inserted into a RELTEQ queue which represents the ready-queue for the EDF scheduler. In line-6, a pointer to an event is maintained to access the deadline event corresponding to a task. For the detailed description of the run-time mechanisms of RELTEQ, see Appendix B.

The expiration of a deadline timer is the result of a task that misses its deadline. Although the event handler of these deadline timers leaves space to implement a policy for overload handling, we consider such policies and their implementation as a future work.

6.8.4 Task completion event

The task-completion event handler presents the operations involved upon completion of a task. Pseudo-code 8 presents the operations performed in the task completion event handler.

Pseudo-code 8 Task completion event

```
 1: /* Delete the deadline event from the ready queue */
 2:
 3: event = task → ReadyQueueEvent;
 4: RelteqQueueDeleteEvent(RelteqEDFReadyQueue, event, &err);
 5: ASSERT_OK(err);
 6: task → ReadyQueueEvent = NULL;
 7:
 8: /* Update spare capacity (resource reclaiming) */
 9: if task → OSTCBInterval == OSICBCur → OSICBId then
10:     cursparecapacity = cursparecapacity + task → OSTCBWorkToBeDone;
11: end if
```

The deadline event associated with the task is deleted from the ready queue on completion of a task execution. In line 3 of Pseudo-code 8, the deadline-event associated with the task is obtained by the event pointer stored in the TCB. The deadline event is deleted from the EDF ready-queue in line 4.

Finally, the spare capacity of the interval is updated after the completion of a task. If the actual execution time of a task is less than the worst-case execution time, any unused resources are reclaimed and added to the spare capacity.

6.9 API support for slot-shifting in μC/OS-II

We developed APIs for setting up the application using the slot-shifting scheduling algorithm. The static parameters associated with intervals and tasks are stored using a set of functions.

API for interval support: The following function is used to create intervals.

OSIntervalCreate(START_TIME, END_TIME, SPARE_CAPACITY, NUMBER_OF_TASKS, POINTERS_TO_TCBs..);

We store the static interval associated attributes in a ICB using OSIntervalCreate function.

API for EDF scheduling: The following function is used to store the relative deadline of a task into an ICB.

OSTaskSetDeadline(OSTCBPrioTbl[TASK_PRIORITY], TASK_DEADLINE);

After creating a task using OSTaskCreateExt function, we get a pointer to a TCB. Using this TCB pointer in the OSTaskSetDeadline function, we store the relative deadline of a task in its TCB.

API for mixed task support: The following function is used to store the type of a task and the WCET of a task.

```
OSTaskSetParam(OSTCBPrioTbl[TASK_PRIORITY], TASK_TYPE, TASK_COMP_TIME );
```

After creating a task using OSTaskCreateExt function, we get a pointer to a TCB. Using this TCB pointer in the OSTaskSetParam function, we store the type of a task and the WCET of a task in its TCB.

We refer the interested reader to Appendix C to understand the configuration of compilation flags and a detailed description of the APIs associated with the slot-shifting scheduling algorithm.

6.10 Summary

In this chapter, we presented the design of tracking intervals. In order to track intervals, interval related attributes are stored in the operating system. We proposed a new data structure using a linked list of Interval Control Blocks (ICBs) to maintain intervals in an RTOS. A counter is used to count the number of slots and keep track of the current scheduling interval. Having stored intervals, the next job is to schedule them to manage the dynamic properties of spare capacities.

Furthermore, the slot-shifting scheduling algorithm is targeted for mixed task sets. An RTOS should provide support for periodic, aperiodic and sporadic tasks for the integration of slot-shifting. The task control block of a operating system is extended with additional parameters to hold the type of a task, the WCET of a task, the completed work of a task and the deadline of a task. A mechanism for WCET monitoring is presented to track the execution of each individual tasks. In addition, the resource-reclaiming mechanism and the borrowing, lending mechanisms are implemented using the WCET monitoring feature.

The spare capacity monitoring is performed to update the spare capacity of an interval based on the execution of all tasks during that interval. The spare capacity of an interval is affected by the execution of a dynamic (sporadic and aperiodic) tasks or the execution of a periodic task outside its own interval. When no task is ready to execute, an idle task consumes the spare capacity of an interval.

Upon arrival, the information regarding the type of a task is deciding the steps involved in placing of a task in the ready-queue. Periodic and sporadic tasks are directly placed in the ready queue upon arrival, while an aperiodic task is undergoing an online admission test to decide on the acceptance of a task. An aperiodic task is placed in the ready-queue, if and only if it can be scheduled together with the off-line guaranteed tasks (periodic and sporadic) and the previously guaranteed aperiodic task(s). Otherwise, a rejection strategy is employed to reject either the newly arrived aperiodic task or the previously guaranteed aperiodic task(s).

To design aperiodic tasks, an approach based on semaphore event is presented. We propose two semaphores to design aperiodic tasks, one semaphore is released from other tasks or ISR's, while the other semaphore is released based on the result of the guarantee algorithm.

The scheduling of tasks requires an EDF scheduler. The EDF design based on the RELTEQ framework is presented. A ready-queue sorted based on the absolute deadline of a task is used by the scheduler. Upon arrival of the task, the deadline event is created and placed in the ready-queue. The head of the ready-queue contains the task with the earliest deadline. The job of the scheduler is pick the task associated with the head event for the execution.

The run-time behavior slot-shifting requires handling number of events. We presented the events associated with the implementation of the slot-shifting. Each event is described with the event handler and the actions performed in the handlers are also explained.

The slot-shifting initialization procedure is presented to schedule tasks using slot-shifting scheduler in μC/OS-II RTOS. We presented the configuration details of flags and APIs associated with slot-shifting scheduling algorithm in Appendix C.

Chapter 7

Results and evaluations

The design of slot-shifting scheduling presented in the previous chapter has been integrated into the μC/OS-II RTOS. In this Chapter, the evaluation results are presented. Grasp visualization tool is briefed in Section 7.1. In Section 7.2, the slot-shifting results are presented with the periodic tasks. The various slot-shifting features are explained with the example task-set in Section 7.2. In Section 7.3, we include sporadic tasks with the periodic tasks to depict the behavior of the slot-shifting scheduling strategy.

The experimental setup of our measurements are discussed in Section 7.4. The performance of the EDF scheduler is compared with FPS and the results are presented in Section 7.5. The run-time complexities of slot shifting in terms of time and memory are presented in Section 7.6. Our findings are discussed in Section 7.7. Finally, this chapter is concluded with a summary.

7.1 Grasp visualization

The visualization of the slot-shifting scheduler requires extension to the logging mechanisms of μC/OS-II. Grasp visualization tool [1] is extended with the new plug-in for the tracking of intervals. In order to register the interval traces in the Grasp output, **logstartinterval** and **logendinterval** mechanisms are added in the OS_LOG.c.

Furthermore, the visualization of the spare capacity requires new logging mechanisms. Run-time mechanisms are already present in the μC/OS-II for the visualization of servers [13]. The server visualization features are adapted to support for the visualization of spare capacities.

In order to visualize the execution trace of tasks with the slot-shifted scheduler, the OS_LOG_EN flag is enabled in OS_CFG.h. We present the results of our Grasp visualizer in Section 7.2 and 7.3.

[1] The Grasp visualization tool can be downloaded from: `http://www.win.tue.nl/~mholende/grasp/`.

7.2 Slot-shifted scheduling of periodic tasks

The functional behavior of the slot-shifting scheduling strategy and its run-time features are illustrated with an example periodic task-set. Next, we present execution traces of the slot-shifting scheduling algorithm with resource-reclaiming and the borrowing and lending mechanisms.

7.2.1 Slot-shifting without resource reclaiming and borrowing mechanisms

The slot-shifting scheduling strategy without the resource reclaiming and borrowing mechanisms. The slot-shifting behavior is explained with the task set presented in Table 7.1. The intervals associated with the task-set presented in Table 7.2. The trace of the execution is presented in Figure 7.1.

Task	Arrival time	Deadline	WCET	Period
P_1	0	40	22	200
P_2	40	80	22	200
P_3	80	140	22	200
P_4	80	140	22	200
P_5	140	200	22	200

Table 7.1: An example periodic task set to elucidate the behavior of slot-shifted scheduler without the resource-reclaiming and borrowing mechanisms.

The trace in Figure 7.1 presents the execution scenario of the task-set presented in Table 7.1 for a single hyper-period. While the tasks belongs to intervals are executing, the spare capacity of the interval remains constant. The execution of idle task consumes the spare capacities of the intervals. The decrease in the spare capacity is identified by the slope in the Figure 7.1. When there is no other ready task to execute, then the idle task executes. The spare capacity of the interval is decreased during the execution of the idle task. At the end of the interval, the spare capacity of the interval reaches zero.

Interval	Start time	End time	Spare capacity	Tasks
Interval 0	0	40	18	$\{P_1\}$
Interval 1	40	80	18	$\{P_2\}$
Interval 2	80	140	16	$\{P_3, P_4\}$
Interval 3	140	200	38	$\{P_5\}$

Table 7.2: Intervals for the task set presented in Table 7.1.

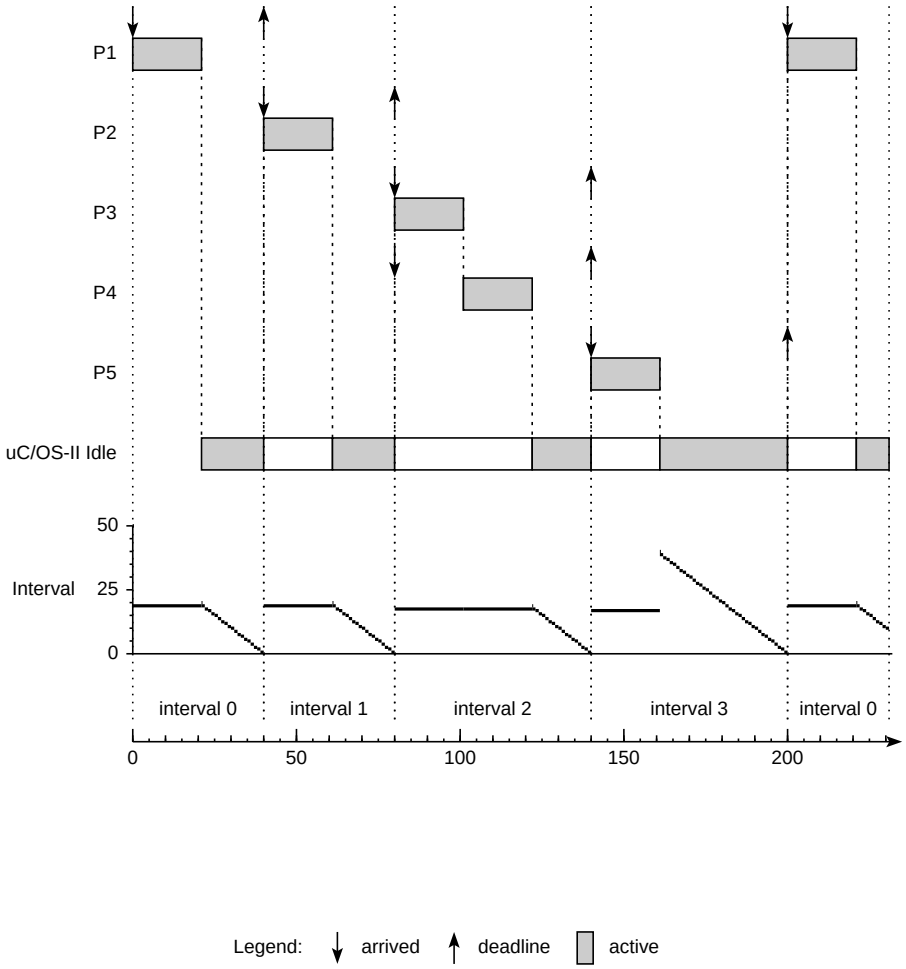

Figure 7.1: The execution trace of the task set presented in Table 7.1 with the slot-shifted scheduler.

7.2.2 Slot-shifting with resource reclaiming mechanism

In previous section, the slot-shifting scheduling is presented for the ideal scenario. However, a task may finish earlier then its worst-case execution time. In such cases, the unused resources can be reclaimed and used for the execution of other tasks in the system. This resource reclaiming phenomenon is presented with the modified task-set presented in Table 7.1.

Due to the modification in the WCET of tasks, spare capacities of intervals are modified. The new set of intervals are created based on the modified task parameters. Table 7.4 presents the new set of intervals for the task-set presented in Table 7.3.

Figure 7.2 presents the execution trace of the task-set presented in Table 7.3 for a single hyper-period. The execution of the task P2 and P3 indicates the resource reclaiming feature of the slot-shifting scheduling strategy. After the completion of the execution, the unused resources are reclaimed. The resource reclaiming is indicated by the increase in the spare capacity. When a task completes its execution, the execution time of a task is compared with

Task	Arrival time	Deadline	WCET	Period
P_1	0	40	22	200
P_2	40	80	25	200
P_3	80	140	25	200
P_4	80	140	22	200
P_5	140	200	22	200

Table 7.3: An example periodic task set to depict the behavior of slot-shifted scheduler with the resource reclaiming mechanism.

Interval	Start time	End time	Spare capacity	Tasks
Interval 0	0	40	18	$\{P_1\}$
Interval 1	40	80	15	$\{P_2\}$
Interval 2	80	140	13	$\{P_3, P_4\}$
Interval 3	140	200	38	$\{P_5\}$

Table 7.4: Intervals for the task set presented in Table 7.3.

the WCET of a task. If a task completes its execution earlier, then the unused resources are reclaimed and added with the spare capacity of the interval. With this mechanism, we use resources efficiently to deal with the dynamically arriving tasks.

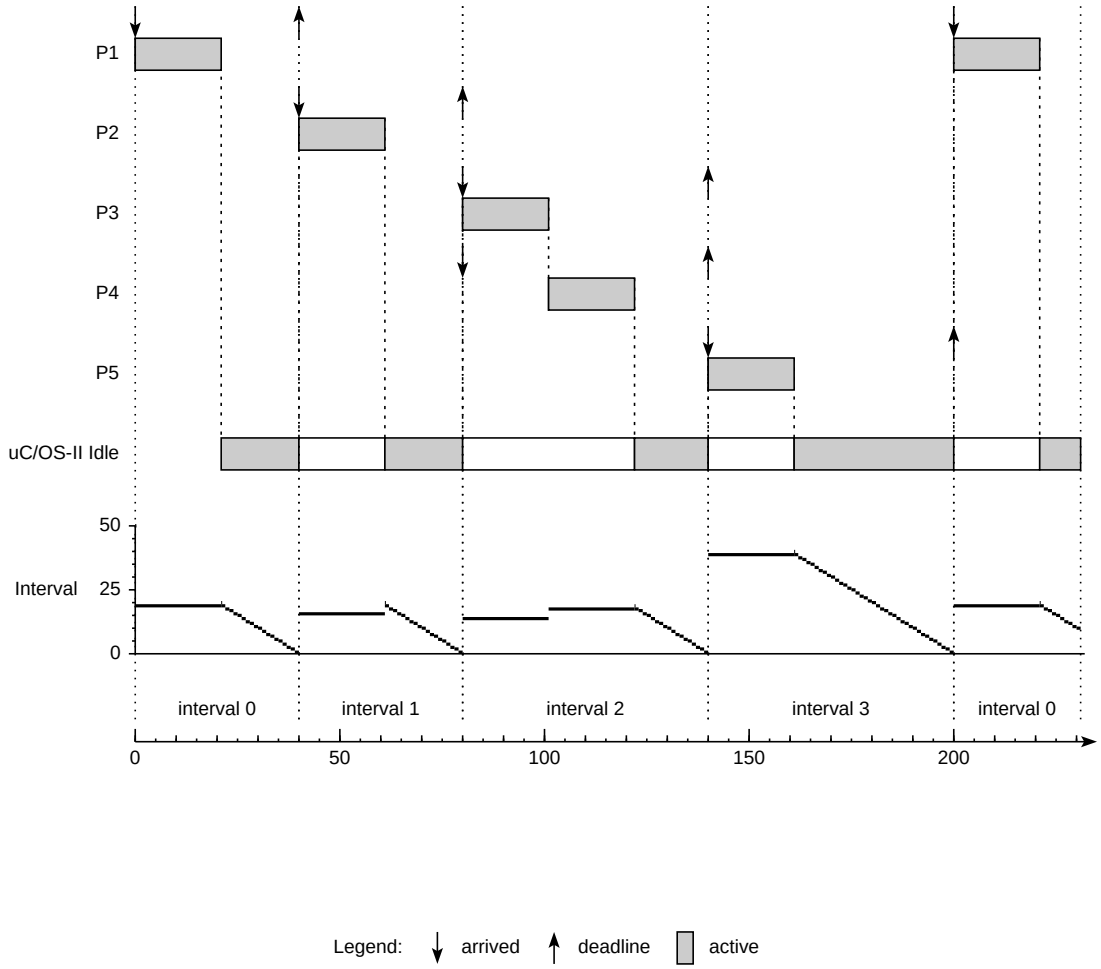

Figure 7.2: The execution trace of the task set presented in Table 7.3 with the slot-shifted scheduler. The resource reclaiming is taking place at the completion of tasks P_2 and P_3 in intervals 1 and 2 respectively.

7.2.3 Slot-shifting with borrowing and lending mechanism

The slot-shifting scheduling strategy proposes the borrowing and lending mechanism to deal with the early arrival of tasks. The early arrival refers to the arrival of a task outside its interval. Tasks are mapped onto intervals based on the deadline. The start time of a interval need not be the start time of a task. Tasks may therefore execute outside their own intervals. The execution of an early arriving tasks are taken care of by the borrowing and the lending mechanisms. The early arriving task executes outside its own interval by borrowing spare capacity from other intervals. The borrowing and lending of spare capacity is explained with the task-set presented in Table 7.5.

Table 7.6 shows the intervals for the task-set presented in Table 7.5. The negative spare capacity in the interval 1 indicates that the task is arriving earlier and the work-load of the task P2 is higher than the resources available in the interval. The task is therefore borrowing

Task	Arrival time	Deadline	WCET	Period
P_1	0	40	22	200
P_2	30	80	44	200
P_3	80	140	22	200
P_4	80	140	22	200
P_5	140	200	22	200

Table 7.5: An example periodic task set to depict the behavior of slot-shifted scheduler with the borrowing and lending mechanisms.

resources from the earlier interval, interval 0. The off-line calculation of the spare capacity takes into consideration the early arrival of a task. During the run-time, the spare capacity information of the interval is used to deal with the early arriving periodic tasks.

Interval	Start time	End time	Spare capacity	Tasks
Interval 0	0	40	14	$\{P_1\}$
Interval 1	40	80	-4	$\{P_2\}$
Interval 2	80	140	16	$\{P_3, P_4\}$
Interval 3	140	200	38	$\{P_5\}$

Table 7.6: Intervals for the task set presented in Table 7.5.

This borrowing and lending of spare capacities are illustrated by the trace in Figure 7.3. The off-line computed spare capacity (using Equation (3.6) is presented in Table 7.6 indicates that the spare capacity of an interval 1 is -4. The execution demand of task P_2 exceeds the amount of resources available in interval 1. The task P_2 is therefore borrowing spare capacity from interval 0, hence the spare capacity of an interval 0 is 14 (instead of 18).

From Figure 7.3, it is evident that the arrival of task P2 is within the interval 0 and therefore it borrows the spare capacity of the interval 0 for its execution. Furthermore, the spare capacity at the start of interval 1 is turning into positive, as the task P_2 have already executed partially in interval 0.

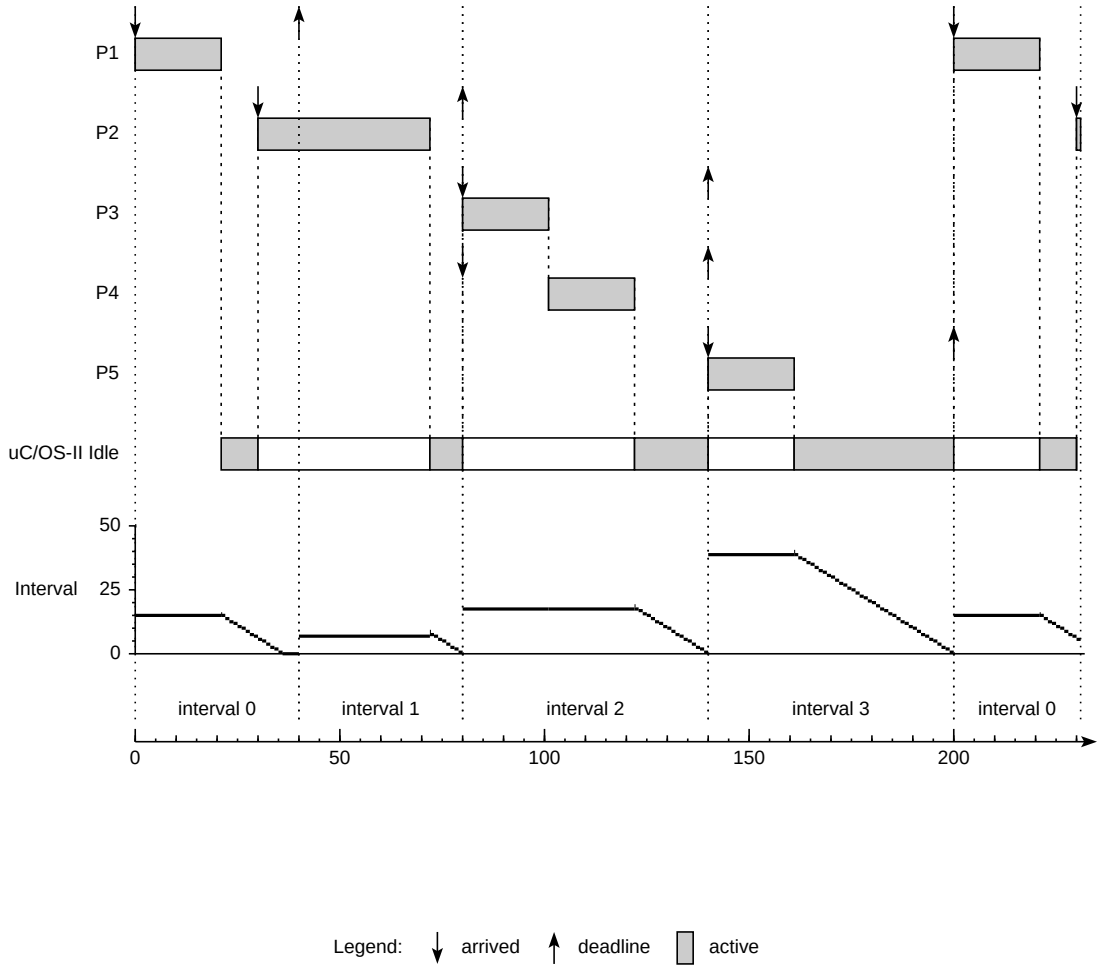

Figure 7.3: The execution trace of the task set presented in Table 7.5 with the slot-shifted scheduler. The execution of task P2 in interval 0, which originally belongs to interval 1, presents the borrowing and lending of spare capacity between interval 0 and interval 1.

7.2.4 Slot-shifting with periodic tasks with multiple instances within a hyper-period

Until now, we have presented the results of slot-shifting scheduling strategy with a simple periodic task set, where there is only one task instance arriving in a hyper-period. In this example, we consider a task set with an arrival of multiple task instance in a hyper-period. Let us consider the task-set presented in Table 7.7.

The hyper-period of the task-set presented in Table 7.7 is 200. In this example, the periodic task P_1 has four instances, the periodic task P_2 has two instances and the periodic task P_3 has one instance executing in the hyper-period. Table 7.8 presents the set of intervals associated with a task-set presented in Table 7.7. The deadline of a task P_1 is 50 and it has four instances in a hyper-period, hence there are four intervals in Table 7.8. The task P_2 has two instances with a deadline of 100, hence it is mapped onto interval 2 and interval 4.

Task	Arrival time	Deadline	WCET	Period
P_1	0	50	22	50
P_2	25	75	22	100
P_3	80	120	22	200

Table 7.7: An example periodic task set to depict the behavior of slot-shifted scheduler with an arrival of multiple instances within a hyper-period.

The deadline of a task P_3 is 200 which is equal to the hyper-period of the periodic task-set presented in Table 7.7, hence the task P_3 is mapped into interval 3.

Interval	Start time	End time	Spare capacity	Tasks
Interval 0	0	50	28	$\{P_1\}$
Interval 1	50	100	6	$\{P_1, P_2\}$
Interval 2	100	150	12	$\{P_1\}$
Interval 3	150	200	-16	$\{P_1, P_2, P_4\}$

Table 7.8: Intervals for the task set presented in Table 7.7.

Figure 7.4 presents the execution trace of the task-set presented in Table 7.7. In this example, we present the execution of multiple instance of a periodic task in a hyper-period. The periodic task P_1 is executing in all intervals.

Borrowing and lending of spare capacity and resource reclaiming: The periodic task P_2 belongs to interval 1 is executing in interval 0 due to the early arrival of a periodic task P_2 in interval 0. At the start of an interval 1, we do not reserve resources for the execution of a periodic task P_2, although it belongs to interval 1, since it has already completed its execution in interval 0. In this way, we reclaim resources resources during run-time and make sure that the resources are not allocated for a task, which has already completed its execution. The spare capacity of an interval 1 increase from 6 to 28, since the task P_2 has completed in interval 0 itself. This run-time behavior is achieved by the WCET monitoring mechanism.

Borrowing crosses multiple intervals: Note that the spare capacity at the start of the interval 3 turns into positive, although the off-line computed spare capacity shows a negative value. The variation in spare capacity is due to the execution of (a second instance of) the periodic task P_2 in interval 2 and the execution of periodic task P_3 in interval 1 and interval 2, although these tasks are originally belong to interval 3. In this example, the task P_3 belongs to interval 3 starts to execute in interval 1 and completes its execution in interval 2, hence the borrowing crosses multiple intervals. The task P_3 is therefore said to have borrowed spare capacity from interval 1 and interval 2, hence the borrowed resources will become available in its own interval.

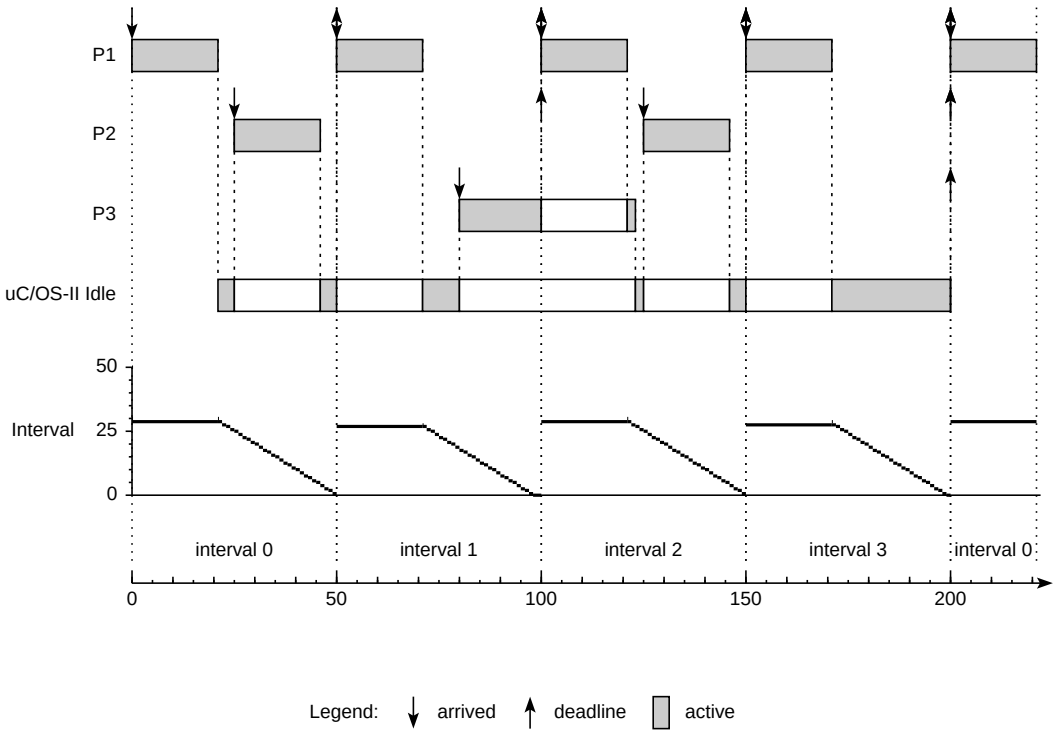

Figure 7.4: The execution trace of the task set presented in Table 7.7 with the slot-shifted scheduler. The task P_1 executes in all intervals, as it has four instances arriving in a hyper-period. In addition, the task P_2 belongs to interval 1 is executing in interval 0 hence, at the start of interval 1, we reclaim resources to allow dynamically arriving tasks.

7.3 Slot shifted scheduling of periodic and sporadic tasks

In previous section, the slot-shifting scheduling strategy is presented with the periodic tasks. In this section, the sporadic tasks are also added with the periodic tasks. The sporadic tasks are guaranteed off-line along with the periodic tasks. In our test setup, we implement sporadic tasks with periodic timers, because in the worst-case scenario, sporadic arrivals behave like periodic tasks. During the run-time, sporadic tasks are allowed for the execution based on its deadline. The execution of the sporadic task uses the spare capacity of the intervals. Let us consider the task-set presented in Table 7.9. Table 7.10 presents the intervals for the task-set presented in Table 7.9.

As we discussed earlier, we consider the worst-case behavior of the sporadic tasks. In the worst-case, the sporadic tasks behaves like periodic tasks. We therefore consider the periodic

Task	Arrival time	Deadline	WCET	Period
P_1	0	40	22	200
P_2	40	80	22	200
P_3	80	140	22	200
P_4	80	140	22	200
P_5	140	200	22	200

Table 7.9: An example periodic task set to elucidate the behavior of slot-shifted scheduler.

Interval	Start time	End time	Spare capacity	Tasks
Interval 0	0	40	18	$\{P_1\}$
Interval 1	40	80	18	$\{P_2\}$
Interval 2	80	140	16	$\{P_3, P_4\}$
Interval 3	140	200	38	$\{P_5\}$

Table 7.10: Intervals for the task set presented in Table 7.9

tasks as sporadic tasks in our applications. Let us consider the sporadic task-set presented in Table 7.3. The execution trace of the mixed tasks (periodic and sporadic) are presented in Figure 7.5.

Task	Arrival time	Deadline	WCET	Period
S_1	0	140	22	220
S_2	70	200	22	220

Table 7.11: An example sporadic task set to elucidate the behavior of slot-shifted scheduler.

The sporadic task execution consumes the spare capacity of intervals. From Figure 7.5, the spare capacity is decremented with the execution of sporadic tasks. In a way, the sporadic task execution is similar to the idle task execution, since both consumes the spare capacity of intervals. The execution of the sporadic task should not interfere with the execution of the periodic tasks. The off-line preparation of tasks takes into account the unused resources to schedule sporadic tasks. During the run-time, the execution of the sporadic task will not affect the execution of the off-line guaranteed periodic tasks, unless there is a task misbehavior.

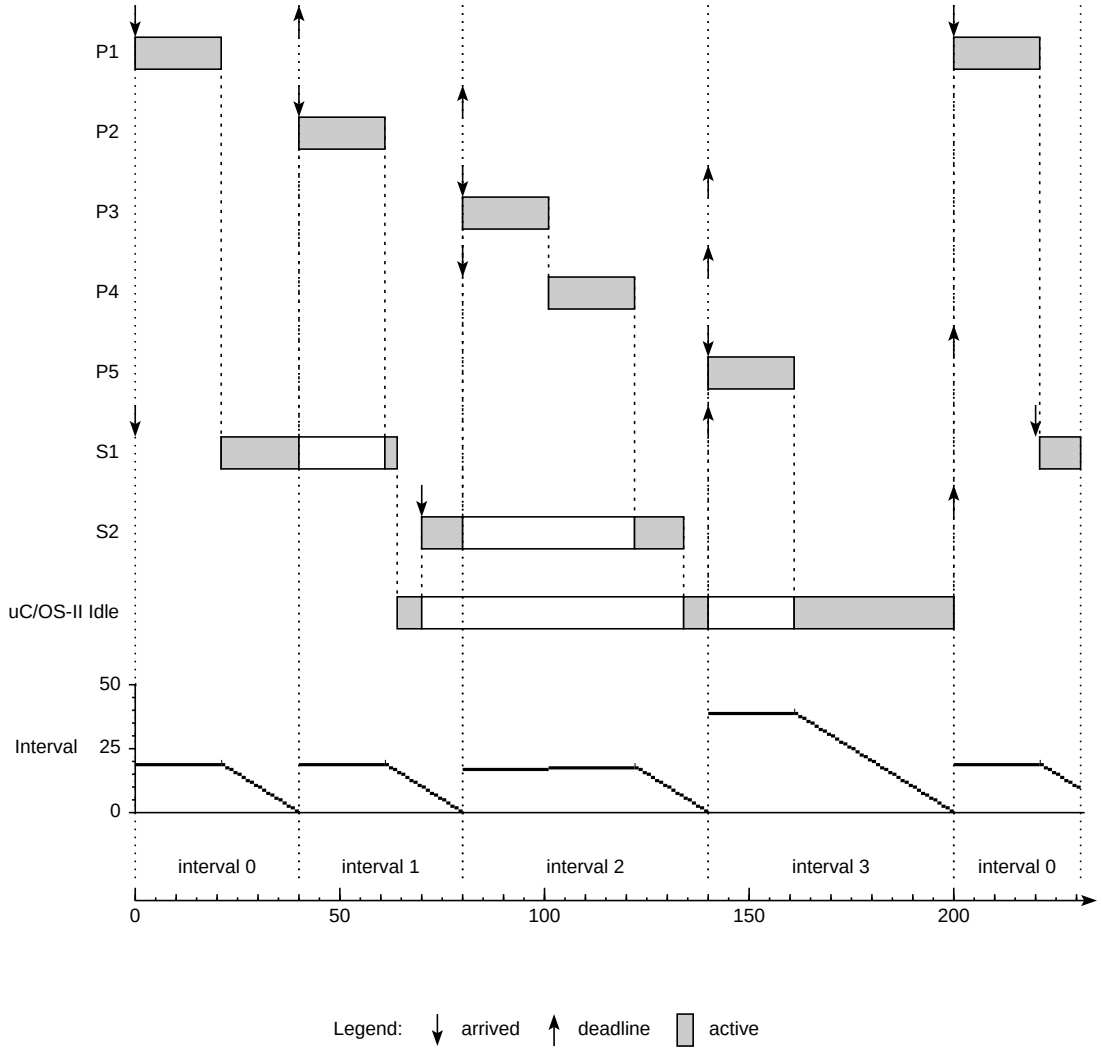

Figure 7.5: The execution trace of the task sets presented in Table 7.9 and Table 7.3 with the slot-shifted scheduler.

7.4 Experimental setup

In all our measurements, the OpenRISC simulator was running at 10Mhz, with a tick frequency of 10ms. The profiling feature discussed in earlier in Section 5.6 is used in our measurements. In OpenRISC simulator, the NOP instruction is used to perform measurements and the cycle count is printed on the console. It is hard to match the profiling point with the cycle counts, as the console is filled with list of cycle dumps. To ease the profiling in μC/OS-II, a separate profiling extension is developed [8]. With the profiling extension, the number of cycles required to execute the piece of code is dumped into a text-file. The number of profiling points are configurable, for our measurements, we used 20000 profiling points. In OS_CFG.h, the OS_PROFILING_EN and OS_PROFILING_TICK_EN flags are enabled to measure the execution time of the tick ISR. The dispatching overhead is measured by en-

abling the OS_PROFILING_EN and OS_PROFILING_SCHED_EN. During the performance measurements, we have disabled the support for assertions and the logging functionality of Grasp. We considered five samples and the data presented are the average of those five samples.

The evaluation of our implementation is carried out with example applications in μC/OS-II. We measure the tick ISR execution-time and the scheduling overhead. Event handlers associated with the slot-shifting scheduler are executed in the context of tick ISR in μC/OS-II. Pseudo-code 9 presents the tick ISR measurement setup. The measurement of the tick ISR execution time is essential to understand the run-time complexity of the slot-shifted scheduler.

Pseudo-code 9 Tick ISR measurement setup

```
1: /* Profiling starts here */
2:
3: Save processor registers;
4: WCET monitoring
5: Spare capacity monitoring
6: Interval scheduling (Counter-based scheduling, see Pseudo-code 1)
7: OSIntEnter() or increment OSIntNesting;
8:
9: RelteqTimeTick();
10: Clear interrupting device;
11: OSIntExit();
12: Restore processor registers;
13: Execute a return from interrupt instruction;
14:
15: /* Profiling ends here */
```

In addition, the scheduling overhead of μC/OS-II is measured by profiling the OSSchedNew() function. In our measurements, the scheduling overhead refers to the time taken to select the task for the execution. In μC/OS-II, the OSSchedNew() function is responsible for the selection of a task for execution. With the FPS, the OSSchedNew() function finds the highest priority ready task for the execution, while the EDF scheduler selects the task at the head of the ready-queue for the execution.

7.5 Performance measurement: EDF versus FPS

The integration of the slot-shifting scheduling algorithm in μC/OS-II requires the design and the implementation of the EDF scheduler. The μC/OS-II RTOS is already presented with the priority based scheduler. With the priority based scheduler, each task is assigned a unique priority, hence the name fixed-priority-scheduler (FPS). During the run-time, the scheduler selects the highest priority ready task for the execution. The earliest-deadline-first (EDF) scheduling requires scheduling of tasks based on the absolute deadline. Hence, the scheduler must be designed to select the task with the earliest deadline for the execution. The EDF implementation is carried on top of RELTEQ framework.

Although EDF scheduler is known for its better resource utilization, it introduces high overhead due to the dynamic nature of deadlines during run-time. The comparison of the EDF scheduler with the FPS would give us an indication of the performance. However, [7] presents

the misconceptions involved in the comparison of EDF versus Rate-Monotonic (RM) i.e. an optimal fixed-priority assignment. The run-time overheads of FPS and EDF are decided by numerous factors. For example, the implementation of the EDF scheduler on top of a priority-based scheduler would introduce overhead. Buttazo [7] also indicates that the overhead caused by EDF and FPS varies with the task-sets.

In this work, the performance of the EDF is compared with the FPS. An EDF scheduler can be implemented on top of a fixed-priority scheduler by manipulating priorities based on the absolute deadlines of tasks. Here, the EDF implementation is carried out on top of a existing RELTEQ framework. In our design, a ready queue is created using the RELTEQ mechanisms. To schedule tasks based on the deadline, the deadline events are inserted into the ready queue. The ready-queue is sorted based on deadline of tasks. The task associated with the head event of the ready queue is selected for the execution by the scheduler. Upon task arrival, the deadline event is inserted in the ready queue for the scheduling. The task arrival is handled by the timer-tick handler in μC/OS-II. Therefore, the overhead introduced by the EDF scheduler can be obtained by measuring the execution time of the tick ISR.

7.5.1 Tick ISR execution time

With the use of RELTEQ in μC/OS-II, the overhead of the timer-tick handler depends on the arrivals of tasks. For the better understanding of the performance, the task-set with the same arrival time is considered for the measurement. All tasks in the system also has a same period and a same deadline. Note that the simultaneous arrival of tasks requires dealing with number of arrival events and we expect the event handling overhead to increase with the increase in number of arrivals. Figure 7.6 presents the results of the EDF versus FPS performance evaluation with simultaneously arriving tasks.

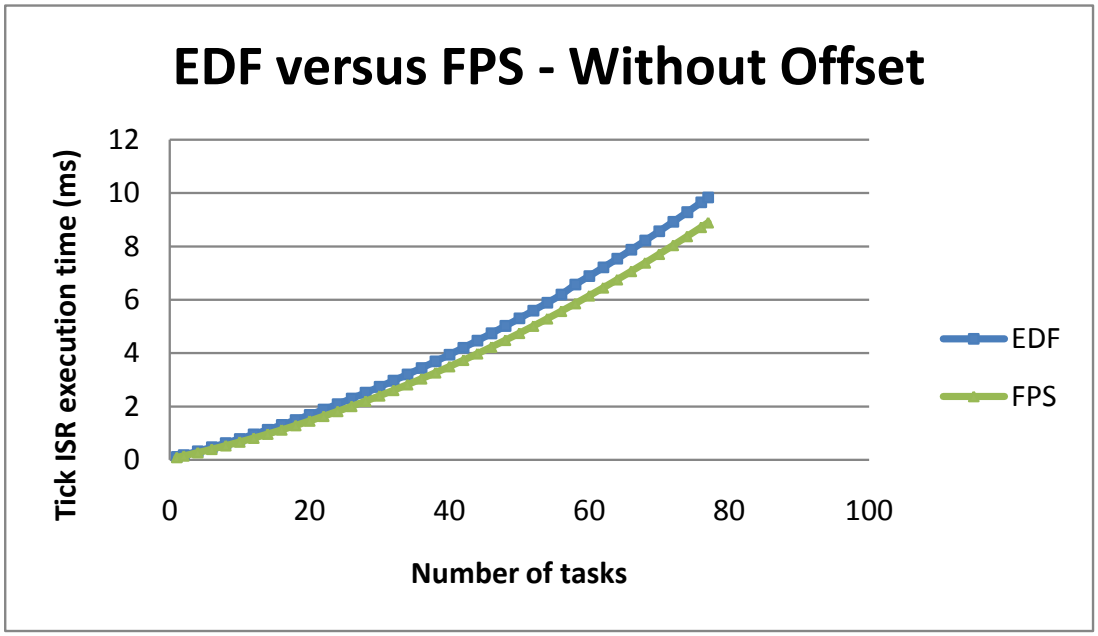

Figure 7.6: Event handling overheads of EDF versus FPS with simultaneously arriving tasks.

Results indicates that the EDF scheduling produces more overhead compared to the FPS. With the FPS, the task associated with a expired event is becoming ready for execution. The built-in μC/OS-II scheduler is responsible for the scheduling of tasks in the FPS using a ready-table. While, the EDF scheduling of tasks requires the insertion of deadline events into the ready queue, as part of the arrival event handler, which introduces the additional overhead. Figure 7.6 confirms the quadratic complexity of the RELTEQ event handlers. The tick ISR execution time of 10 ms corresponds to 100% processor utilization.

To understand the relation between the arrival patterns and the tick-handler overhead, we have considered a task-set with a different arrival times. The tasks are created with the offset in such a way that there is no more than single arrival at any time slot.

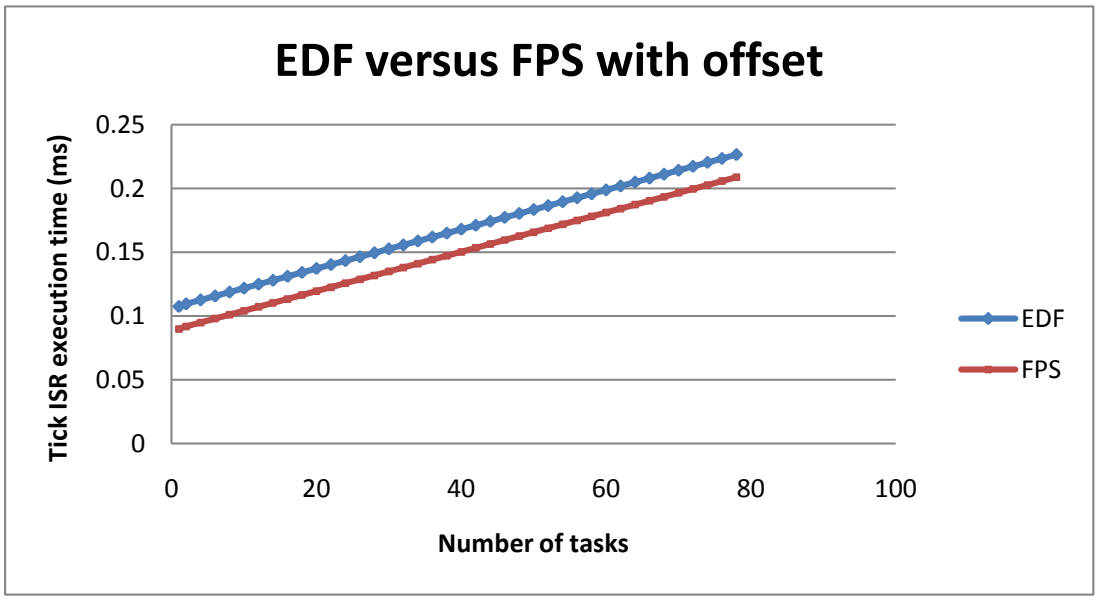

Figure 7.7: Event handling overheads of EDF versus FPS with offset, where tasks have relative activation offsets.

Figure 7.7 shows the similar trend as Figure 7.6, but the tick ISR execution time is reduced by the large extent due to the offset. The increase in the number of tasks increases the tick ISR execution time gradually, and the traversing of the RELTEQ to insert the event at the appropriate position is the reason behind this gradual increase in the tick-handler execution time, although there is only one arrival at any time slot.

From Figure 7.6 and Figure 7.7, we can conclude that the tasks with the same arrival time leads to the worst-case execution time of tick-handler and the tasks with the offset results in the best-case execution time of the tick-handler. The overhead is distributed over a larger interval, leading to smaller ISR overheads. The event handling overhead of the EDF scheduler, for distributed arrivals of tasks, is approximately 15% high as compared to de-facto FPS. Figure 7.6 and Figure 7.7 confirms the complexity in the number of tasks for inserting a new event. The run-time complexity of our implementation is $\mathcal{O}(n^2)$, where n is the number of tasks (arrivals). But, an implementation using binary trees would actually reduce it to

$\mathcal{O}(n * \log(n))$. The binary tree implementation is considered as a future work.

7.5.2 Scheduling overhead

The scheduling overhead of the μC/OS-II is obtained by measuring the execution time of the OSSchedNew() function. This function is responsible for selecting a task for the execution. The operations performed to select a task for the scheduling is constant irrespective of the number of tasks in the system. The scheduling overhead of the FPS and EDF schedulers are presented in Table 7.12.

Scheduling algorithm	**Scheduling overhead**
EDF	14.6 μs
FPS	7 μs

Table 7.12: Scheduling Overhead: EDF versus FPS.

The scheduling overhead of the EDF is higher than the FPS because of the operations involved in choosing the task for the execution. The EDF scheduler picks the task at the head ready queue sorted based on the deadline. With the FPS, the highest priority task is selected for scheduling from the ready-table. The scheduling overhead indicates the time taken to identify the task with the earliest deadline and the highest priority read task, for the EDF and the FPS respectively.

7.6 Implementation complexity of Slot-shifting

In this section, we present the implementation complexity by looking at the run-time complexity and the memory overhead. The slot-shifting run-time mechanisms are evaluated and as we discussed earlier in Section 7.4, the tick ISR execution time is essential to understand the run-time overhead. We evaluated the performance of the slot-shifted scheduler by measuring the event handling overheads. The memory overhead is identified by evaluating the memory requirement of the of the slot-shifting run-time mechanisms.

7.6.1 Run-time complexity

The slot-shifting run-time complexity is evaluated with an example application. The slot-shifting scheduling algorithm requires the WCET monitoring, the spare capacity monitoring and the interval-tracking mechanisms. As presented in earlier in Pseudo-code 9, all these mechanisms are executed within the context of tick ISR. We created an example application with a one to one mapping between tasks and intervals. In other words, the number of intervals equals the number of tasks. For this measurement, we did not consider the borrowing, lending and the resource-reclaiming mechanisms of slot-shifting. The arrival time of tasks is expected to cause the worst-case overhead and it is also the reason behind the exclusion of these slot-shifting run-time mechanisms.

We consider two different types of task sets. Firstly, a task-set is considered with simultaneous arrival times, where all tasks in the application arrives at a single time slot. The measurement results are presented in Figure 7.8. From Figure 7.8, it is apparent that the tick ISR execution time increases quadratically with the increase in tasks. An event handler is invoked upon arrival of a task. The higher the number of arrivals in a slot, the higher the tick ISR execution time, due to the execution of arrival event handlers within the context of tick ISR.

In addition, we considered a different task set with distributed arrival times. The slot-shifting scheduling technique proposes the notion of intervals which are decided by the arrival-time and the deadline of a task. Typically, we expect tasks to have distributed arrival times, although tasks with simultaneous arrival-time is not completely ruled-out by the slot-shifting scheduling algorithm. Figure 7.8 indicates that tasks with off-set performs better then tasks with simultaneous arrival times (with offset). At any-time slot, there is only one arrival which results in the execution of only one arrival event-handler. The tick ISR execution time for tasks with offsets is not a constant. Although one task is released at any slot, there is a slight increase in the tick ISR execution time with the increase in the number of tasks in the application, due to the traversing of a queue during the insertion of a new arrival event.

From Figure 7.9 and 7.10, it is apparent that the event handling overhead of slot-shifting is high compared to EDF. For the EDF scheduler, the event handling overhead comes from the execution of the task arrival event handler. But, with slot-shifting, the WCET monitoring, the spare capacity monitoring and the interval event handlers are also executed additionally in the context of tick ISR. The linear increase in tick ISR overhead for the slot-shifting scheduler comes from the execution of the slot-shifting run-time mechanisms such as WCET monitoring, Spare capacity monitoring and the interval tracking. The run-time complexity of our slot-shifting implementation is $\mathcal{O}(N)$. It is the reason behind the increase in the quadratic slope.

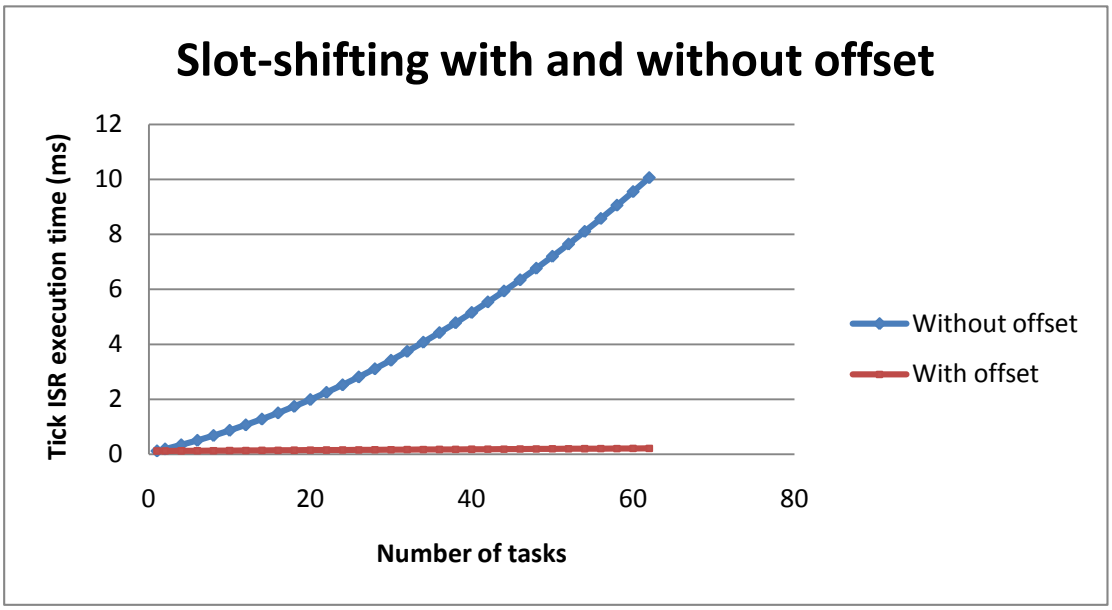

Figure 7.8: Event handling overhead of slot-shifting with simultaneous arrival of tasks (without offset) and with offsets for task arrivals.

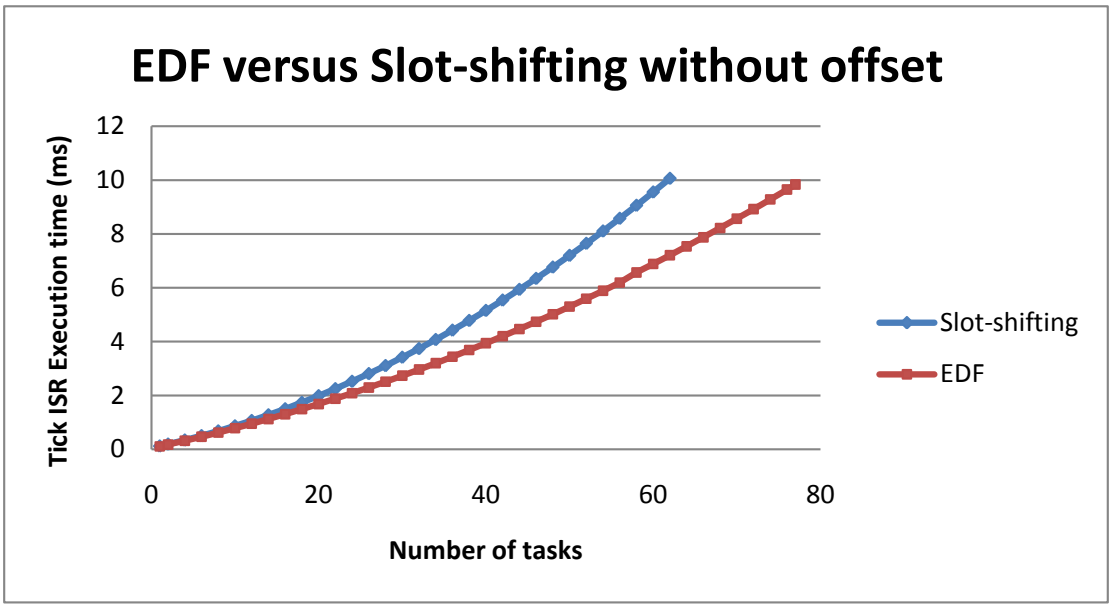

Figure 7.9: Event handling overheads of EDF versus Slot-shifting with simultaneously arriving tasks (without offset).

The event handling overhead of slot-shifting scheduler is approximately 26% high as compared to the de-facto FPS scheduler of μC/OS-II RTOS. For slot-shifting scheduler, overheads comes

from interval tracking, WCET monitoring, spare capacity monitoring and EDF scheduling of tasks.

Furthermore, our implementation of slot-shifting in μC/OS-II allows us to use either FPS or EDF scheduling for tasks. We evaluated the performance of slot-shifted scheduler with FPS and EDF. Figure 7.11 and Figure 7.12 presents our results of slot-shifting with EDF and FPS. The slot-shifted scheduler performs better with FPS as compared to EDF, since the EDF scheduling of tasks requires creation and insertion of deadline events into the ready-queue. While, the de-facto FPS of μC/OS-II uses a ready-table for the scheduling tasks based on a priority.

The tick ISR execution time of a slot depends on the number of events handled, which depends on the number of task arrivals. The number of task arrivals depends on the number of tasks in the system.

However, the core concept of slot-shifting scheduling focuses on tasks with complex constraints. Having studied the behavior of operating system with simultaneously arriving tasks, it is not advisable to allow tasks with simulations arrival times on resource-constrained platforms. During the off-line preparation phase, the arrival times of tasks can also be taken into consideration as one of the system constraint to improve the run-time behavior. Typically, applications will have tasks with distributed arrival times and hence the run-time overhead of the slot-shifting scheduling algorithm is minimized. Tasks with a simultaneous arrival time is a rarity and the handling of such a task-set poses challenges independent of the scheduling algorithm.

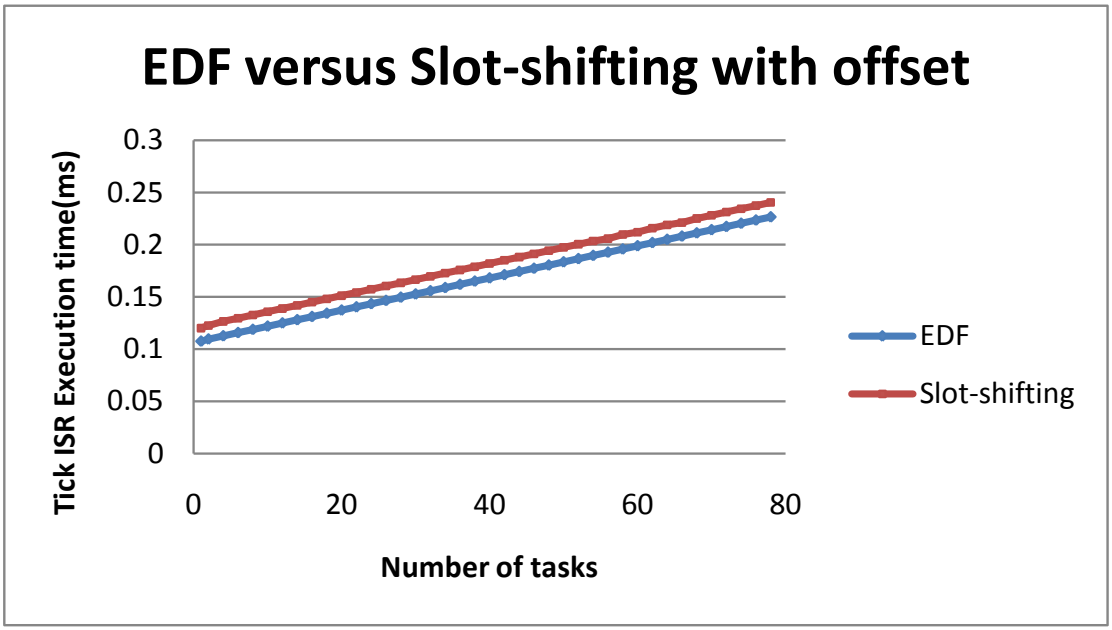

Figure 7.10: Event handling overheads of EDF versus Slot-shifting for tasks with relative arrival offsets.

Our implementation of the slot-shifting run-time mechanisms uses the RELTEQ timer management mechanisms. The task-arrival is handled within the context of rick ISR in the existing RELTEQ configuration. The handling of task-arrival event outside the tick ISR might im-

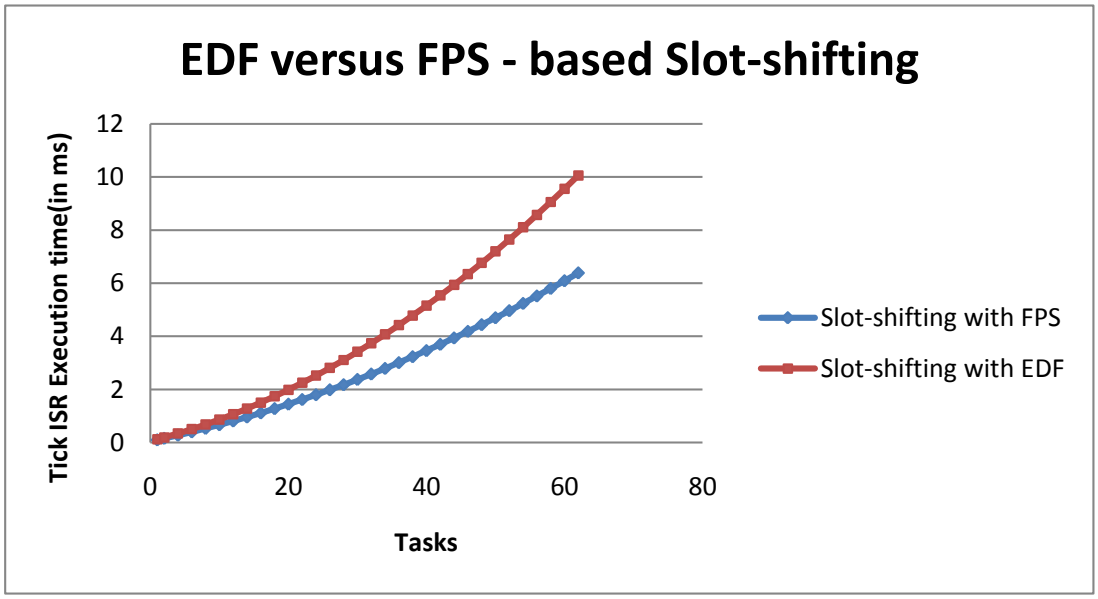

Figure 7.11: Event handling overheads of Slot-shifting with simultaneously arriving tasks: Slot-shifting with EDF versus Slot-shifting with FPS.

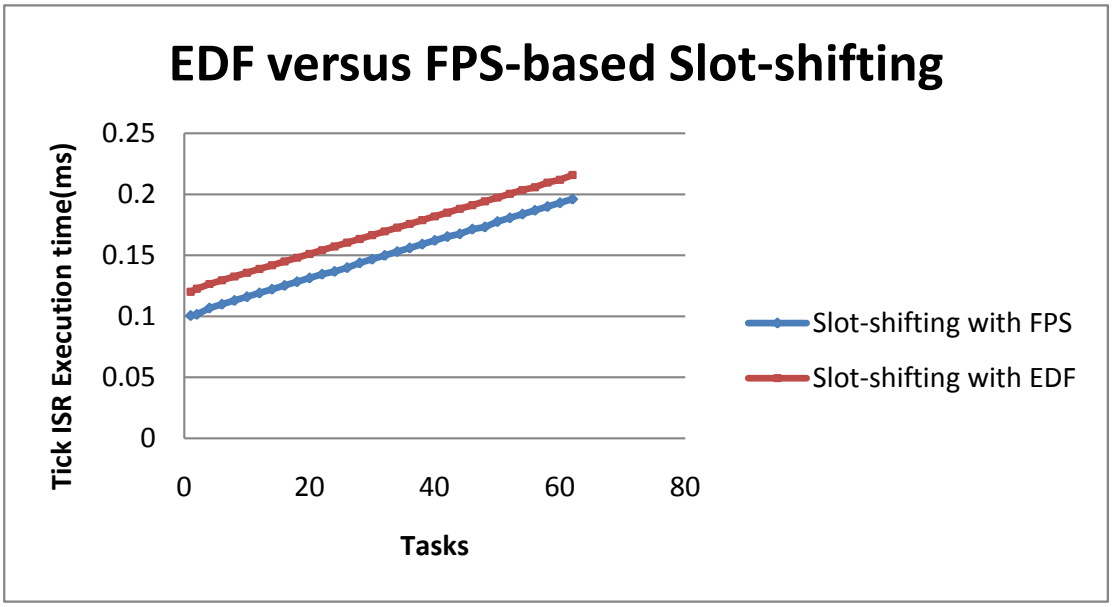

Figure 7.12: Event handling overheads of Slot-shifting for tasks with relative arrival offsets: Slot-shifting with EDF versus Slot-shifting with FPS.

prove the performance of the slot-shifting scheduling algorithm. However, we consider such improvements as our future work.

Furthermore, the slot-shifting scheduling algorithm uses EDF for the scheduling of tasks based on the deadline. Section 7.5 presents the scheduling overhead of EDF scheduler. The

scheduling overhead of slot-shifting scheduling algorithm is same as EDF, as the operations involved in the selection of a task for the execution is same. So, we refer the reader to Section 7.5.2 to know about the scheduling overhead of EDF scheduler.

7.6.2 Memory complexity

The slot-shifting scheduling algorithm requires new run-time mechanisms to be implemented in μC/OS-II. New data structures are added to maintain task and interval attributes. To evaluate the memory complexity of our implementation, we manually evaluated our extensions. We present the memory overhead of slot-shifting scheduling algorithm in Table 7.13.

Extensions	**Memory overhead (in bytes)**
ICB	56
TCB	64

Table 7.13: Memory complexity of slot-shifting.

The memory overhead of the slot-shifting scheduling algorithm constitutes of ICBs and the extensions to the TCB. Table 7.13 presents the memory requirement for a single ICB. The memory overhead presents an ICB with a single task. The memory overhead of a ICB increases with the increase in number of tasks in a ICB. The higher the number of intervals in the application, the higher the memory overhead. In addition, the TCB of a task is extended with new attributes for slot-shifting. The memory overhead of the slot-shifted scheduler increases with the number of tasks in the application. The maximum number of ICBs is $\mathcal{O}(N)$, where N denotes the number of intervals (See Section 3.4). The size of the linked list of ICBs is therefore $\mathcal{O}(N)$. The TCB extensions are needed irrespective of the task type and therefore the memory overhead is $\mathcal{O}(n)$, where n is the total number of tasks (See Section 3.3). The total overhead is therefore $\mathcal{O}(N + n)$;

Note that the spare capacity is dynamically updated during run-time. We only need to know the spare capacity of the current interval to be updated and this value can be saved in a single variable stored in memory. In the ICB, however, we can make all data static, so that it can be stored in Read-Only Memory (ROM) rather than in Random Access Memory (RAM). In this work, we do not store the static parameters in ROM, however, such an improvement is considered as a future work.

7.6.3 Memory complexity of EDF

The implementation of EDF requires deadline events, pointer to a deadline event and a ready-queue. Each deadline event requires 16 bytes and it increases linearly with the increase in number of tasks. In addition, we maintain a pointer to the ready-queue event with a TCBReadyQueueEvent pointer to keep track of a ready queue event. The memory complexity of a deadline event and a pointer to the ready-queue event increases with the increases in the total number of tasks, hence it is $\mathcal{O}$(n), where n is the total number of tasks (See Section 3.3).

Extensions	**Memory overhead (in bytes)**
Deadline event	16
TCBReadyQueueEvent	4
EDFGlobalReadyQueue	16

Table 7.14: Memory complexity of EDF.

The EDF ready-queue is used to store deadline events and this is the basis for the scheduling of tasks based on the deadline. The memory complexity of the EDF ready-queue depends on the number of active events. During the simultaneous arrival of tasks, the memory complexity of the EDF ready-queue depends on the number of arrivals, which in turn depends on the number of tasks. The worst-case memory complexity of the EDF ready-queue is $\mathcal{O}$(n).

7.7 Discussions

In this section, we present our findings during the design and implementation of slot-shifting scheduling algorithm in μC/OS-II.

7.7.1 On the complexity of online acceptance test

In our work, we implemented the slot-shifting scheduling algorithm with the support for handling periodic and sporadic tasks. In addition, the support for resource reclaiming and the borrowing & lending mechanisms are integrated in μC/OS-II. The original slot-shifting scheduling algorithm in [15] deals with periodic, aperiodic and sporadic tasks and aperiodic tasks. Aperiodic tasks are guaranteed with a online guarantee test. A guarantee algorithm is not implemented for the reasons discussed in Section 6.7. After passing this test, tasks are executed in the spare capacity in the same way as sporadic tasks.

The tick ISR can be seen as a periodically arriving task at the highest priority. In EDF, this means that the computation time is equal to its deadline, i.e. $C_{\text{ISR}} = D_{\text{ISR}}$. In order to minimize the interference by ISR executions, we aim to (*i*) minimize the computation time; (*ii*) minimize the fluctuations in the computation time, since these fluctuations may lead to activation jitter for other tasks.

On the one hand, the execution of guarantee test in the context of tick ISR increases the overhead and it is dangerous to perform a guarantee test within the context of tick ISR in a resource-constrained system.

On the other hand, the execution of a guarantee test within the context of a task delays the acceptance notification and requires to reserve resources explicitly during the off-line phase to perform a guarantee test online. This limits the granularity of the deadlines of tasks that can be serviced by a system. The online guarantee test of an aperiodic task should therefore be improved to work efficiently in a resource-constrained systems and in turn to improve predictability. We consider the implementation of a online guarantee algorithm as our future work. Moreover, we opt for a reservation-based approach to limit the worst-case ISR overheads.

However, a guarantee algorithm needs to be executed on arrival of aperiodic events. The execution of a guarantee test within an ISR affects the predictability, as the computation complexity of a guarantee test depends on the previously guaranteed aperiodic tasks. We propose a approach based on a sporadic task to perform a guarantee test to improve the predictability of a slot-shifting algorithm, while preventing aperiodic tasks from overloading the system. A sporadic task with a minimum inter-arrival time can be used to perform a guarantee test in the slot-shifting scheduling algorithm. The downside of this technique is the delayed response to aperiodic events, as the guarantee (sporadic) task arrives only after its minimum inter-arrival time has elapsed.

Furthermore, an improved guarantee algorithm can be developed with a less resource requirement, to improve the performance. Isovic and Fohler [15] propose a online guarantee algorithm, which considers the last activations of sporadic tasks. By considering the worst-case arrival behavior of sporadic tasks, the performance of the online guarantee algorithm can be improved. In addition, the impact of a sporadic task is needed for the acceptance of an aperiodic task. By storing the pre-computed sporadic tasks impact in a table, the run-time complexity of a guarantee algorithm can be minimized.

7.7.2 On the absence of critical slots during run-time

The slot-shifting scheduling algorithm introduces the notion of intervals defined by the arrival-time and the deadline of a task. A task with a deadline at the end of the interval, has a guaranteed resource in that interval. The critical slot of an interval is defined as the slot where the execution of the dynamically arriving tasks are delayed due to the execution of off-line guaranteed periodic tasks.

In our work on slot-shifting, the critical slot of an interval is not tracked during run-time, as the guaranteed tasks will meet their deadline, in the absence of task misbehavior. Note that such a deadline violation is detected by our WCET-monitoring mechanism, i.e. the deadline timer expires, so that an appropriate action can be taken. Once a task is guaranteed to execute with the slot-shifting scheduling algorithm, then it should meet its deadline. A task misbehavior, a system malfunction or an incorrect guarantee test can only lead to a task missing its deadline. The critical slot of an interval is therefore not tracked during run-time, but merely used for analysis purposes during the off-line scheduling phase [15].

7.7.3 On the management of misbehaving tasks

With slot-shifting scheduling algorithm, we use the WCET of a task to track the execution of a task during run-time. When a task completes its execution before its WCET, then the

unused resources are reclaimed and added with a spare capacity to deal with dynamically arriving tasks. On the contrary, when a task executes longer than its WCET, we terminate the application with a task misbehavior message. Although, we may allow a misbehaving task to execute with a spare capacity of an interval, but such an extension might affect the execution of other off-line guaranteed periodic tasks in the system. With an improvement in the WCET and Spare capacity monitoring mechanism, we might allow misbehaving task to execute with a spare capacity of an interval. However, such extensions are outside the scope of this work.

7.8 Summary

The execution trace of the slot-shifted scheduler is visualized using the logging mechanisms in μC/OS-II via Grasp visualization tool. Using this tooling, the run-time behavior of the slot-shifted scheduler is presented with examples comprising periodic tasks and sporadic tasks.

Next, we evaluated the performance of our implementation. The performance of the EDF scheduler is compared with the de-facto FPS scheduler of μC/OS-II. The tick ISR execution time is measured, since the arrival-events of tasks are handled in the context of ISR. With the increase in the number of arrivals, the tick ISR execution time increases quadratically. The dispatching overhead of the EDF and FPS schedulers are compared, and the result indicates a marginal increase in the scheduling overhead for EDF. The event handling overhead of EDF scheduler is approximately 15% high compared to FPS, for distributed arrivals of tasks.

The performance of a slot-shifted scheduler is evaluated and the results are presented. From the evaluation results, it is apparent that a task-set with distributed arrival times performs better as compared to a task with simultaneous arrival times. The event handling overhead of slot-shifting scheduler is approximately 26% high compared to FPS, for distributed arrivals of tasks. A nice feature of slot-shifting is that it is likely that arrivals are distributed by design. The findings of our work are presented in Section 7.7.

Chapter 8

Conclusions and future Work

8.1 Conclusions

In this work, we carried out the integration and evaluation of the slot shifting scheduling strategy in μC/OS-II. We implemented the slot-shifting scheduling strategy with minimum modification to the μC/OS-II kernel.

In chapter 4, we presented the run-time mechanisms required for the integration of slot-shifting scheduling algorithm. We identified the run-time mechanisms by applying a slot-shifting scheduling concept presented in [15], to case-studies. Furthermore, we presented the static system properties in chapter 3. With the help of the static system properties, we carried out the design of the slot-shifting scheduling algorithm.

The slot-shifting scheduling algorithm requires tracking of intervals and spare capacities during run-time. We presented the interval-tracking feature in Section 4.5.1. The design of the interval-tracking mechanism is presented in Section 6.1. The notion of an interval is used in the slot-shifting scheduling strategy to handle dynamically arriving tasks. An interval is defined by the arrival-time and the deadline of a task. Additionally, the end-time of the previous interval is considered as the start time of an interval. The slot-shifting scheduling strategy proposes borrowing and lending mechanisms to deal with early arriving tasks. We implemented the borrowing and lending mechanisms by using a WCET monitoring mechanism presented in Section 4.5.

In the slot-shifting scheduling algorithm, the dynamically arriving tasks are executed with the unused resources in the intervals, also known as the spare capacity of an interval. Isovic and Fohler [15] proposed a resource-reclaiming mechanism to reclaim the unused resources allocated to aperiodic tasks to reclaim unused resources during run-time. In our work, we extended the resource reclaiming mechanism to deal with the early completion of periodic and sporadic tasks. We used the WCET monitoring feature to reclaim unused resources during run-time. During run-time, we dynamically update the spare capacity of an interval to deal with the dynamically arriving tasks.

Furthermore, tasks are scheduled based on their deadline in slot-shifting. Since many RTOSes, including μC/OS-II do not implement deadline driven scheduling, i.e. EDF, we designed and implemented an EDF scheduler, for μC/OS-II. We carried out our implementation on top of

the RELTEQ timer management mechanisms.

To analyze the performance of our extensions in μC/OS-II, we evaluated the performance of the EDF scheduler and the slot-shifting scheduling algorithm. To better understand the performance of the EDF scheduler, we compared the performance of the EDF scheduler with the de-facto FPS of μC/OS-II. We presented the results of the EDF versus FPS comparison in Section 7.5. In addition, we evaluated the performance of slot-shifting scheduling algorithm and presented our results in Section 7.6. The tick ISR execution of the EDF scheduler is high compared to FPS, due to the additional operations performed in the arrival event handler for EDF scheduling. With FPS, a new arrival event is created and inserted into the system queue. But, for the EDF scheduler, the deadline event is created and inserted into the ready queue, hence the increase in tick overhead. The scheduling overhead of the EDF scheduler is also marginally high compared to FPS. The event handling overhead of EDF scheduler is approximately 15% high compared to FPS, for distributed arrivals of tasks.

The performance of slot-shifting is also evaluated. Typically, the slot-shifting scheduling strategy uses task-set with a varying arrival times. Tasks with simultaneous arrival times have shown to increase the run-time overhead, due to the execution of a burst of arrival-event handlers. In slot-shifting, the intervals regulates a distribution of arrivals, which reduces the run-time overhead as the arrival events are distributed over more than slot. The event handling overhead of slot-shifting scheduler is approximately 26% high compared to FPS, for distributed arrivals of tasks. The distributed arrivals of tasks does not reduce the absolute overhead; it reduces the worst-case periodic interference and therefore it is more predictable. The impact of the arrival-time of a task can therefore be considered as a system constraint, and can be resolved off-line, to improve the run-time performance.

The slot-shifting scheduling algorithm is suitable for applications with a distributed arrival times of tasks, as the performance of the slot-shifted scheduler with offsets behaves better than tasks with simultaneous arrival times. With offsets, the run-time overhead of the slot-shifted scheduler is distributed over number of time slots, which minimizes the tick ISR execution time. On the other hand, the simultaneous arrival of tasks handles all arrival events in one time slot, which results in large tick ISR execution time. The simultaneous arrival of tasks are therefore bound to increase the run-time overhead irrespective of the scheduling algorithm used, unless the efficiency of the arrival event handler is improved.

8.2 Future work

In this section, we suggest future work on integrating slot-shifting in μC/OS-II.

8.2.1 Increasing the RTOS predictability

In the slot-shifting scheduling algorithm, a guarantee algorithm is needed to accept an aperiodic task during run-time. Isovic and Fohler [15] proposes a guarantee algorithm, which takes into consideration the previously guaranteed periodic, aperiodic and sporadic tasks along with the newly arrived aperiodic task.

Typically, aperiodic tasks are activated by an external hardware interrupts. The interrupt-driven nature of cheap hardwares creates numerous challenges in real-time systems. A faulty

hardware device can overload the system by activating a number of aperiodic tasks. This could lead to the execution of a guarantee test for each of the newly arrived aperiodic tasks. A system overload due to an interrupt burst can therefore lead to the system failure in the worst-case.

In Section 6.3.2, we have already discussed about the enforcement of inter-arrival times for sporadic tasks. The arrival pattern of such sporadic tasks are uniform, but the dynamic arrival pattern of interrupt-driven aperiodic tasks requires dedicated run-time mechanisms to prevent system overload due to the *top-half execution* of interrupt handler. Regehr and Duongsaa [26] propose methods to prevent such interrupt overload problems in embedded systems. Methods are based on temporarily disabling faulty interrupt sources, i.e. they assume that each interrupt source can be masked individually.

The enforcement of interrupt inter-arrival times is one of the solution proposed by Regehr and Duongsaa [26], which is similar to the minimum inter-arrival time enforcement of Rajkumar [25]. Alternatively, they propose a method based on a reservation to handle bursty interrupt sources is also presented in [26].

Finally, both approaches by Regehr and Duongsaa are considered as viable complementary techniques to our implementation. However, an implementation is considered beyond the scope of this work.

8.2.2 Performance enhancement of RELTEQ

With RELTEQ framework, tasks arrival events are handled within the context of tick ISR. The tick ISR execution time increases quadratically with the increase in the number of tasks in the system. The performance can be improved by handling arrival events outside the context of tick ISR. An arrival event can be handled within a task. Upon arrival of a task, the arrival time of a task can be stored. On the task completion, insert a timer for the next activation based on the stored arrival time.

Alternatively, an implementation of RELTEQ based on the binary tree improves the performance, as it reduces the complexity of deletions and insertions from linear to logarithmic.

8.2.3 Resource sharing between aperiodic and sporadic tasks

In the slot-shifting scheduling algorithm, all task dependencies are resolved off-line, which leads to (relatively) independent execution of tasks during run-time. If sporadic or aperiodic tasks are converted to periodic tasks, then the resource is over-provisioned, which leads to less spare capacity for the execution of dynamic events. The synchronization protocol for sharing resources between aperiodic tasks and sporadic tasks can therefore be an interesting solution for limited form of dependencies, i.e. mutual exclusive execution of so-called critical sections.

8.2.4 Slot-shifting on distributed nodes

Slot-shifting scheduling can also be employed in distributed nodes. In this work, we presented the scheduling of tasks in a single node. The slot-shifting can be extended to support multiple nodes, as the run-time mechanisms are same for each node in the network. In a dis-

tributed setting, the slot-shifted scheduler requires synchronization of time slots. The choice of time synchronization may depend on the degree in which a design considers an autonomous operation of nodes in the system [17].

The time synchronization between nodes can be achieved at various granularity. Based on the application requirement, a time synchronization can be performed at slots, intervals or hyper-periods. For the synchronization of time at slots, the timer-tick interrupt occurrence on multiple nodes should be synchronized with a single global reference. Aoun et al. [4] describes technique to perform tick alignment in a distributed setting. Aoun et al. tested their technique in FreeRTOS, which is similar to our target RTOS, μC/OS-II. With an extension to perform tick alignment on distributed nodes at suitable granularity, the slot-shifted scheduler can be employed in a distributed setting.

References

[1] *VxWorks Reference Manual - 5.3.1*, April 1998.

[2] *OpenRISC Architecture Manual, Rev 1.1*, July 2004.

[3] Opencores. http://opencores.org/or1k/Main_Page, 2011.

[4] Marc Aoun, Julien Catalano, and Peter Stok. Distributed task synchronization in wireless sensor networks. In *Proceedings of the 6th European Conference on Wireless Sensor Networks*, EWSN '09, pages 150–165, 2009.

[5] Arcticus Systems. *Rubus OS - Reference Manual*, June 2004.

[6] Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proc. Real-Time Systems Symposium (RTSS)*, pages 182–190, 1990.

[7] Giorgio C. Buttazzo. Rate monotonic vs. EDF: judgment day. *Real-Time Syst.*, Volume 29, Issue 1:5–26, January 2005.

[8] Wim Cools. Extending μC/OS-II with FPDS and reservations. Master's thesis, Eindhoven University of Technology, July 2010.

[9] A. Damm, J. Reisinger, W. Schwabl, and H. Kopetz. The real-time operating system of MARS. *SIGOPS Oper. Syst. Rev.*, Volume 23, Issue 3:141–157, July 1989.

[10] Radu Dobrin. *Combining Off-line Schedule Construction and Fixed Priority Scheduling in Real-Time Computer Systems.* PhD thesis, Mälardalen University, September 2005.

[11] Gerhard Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proceedings of the 16th Real-Time Systems Symposium*, Pisa, Italy, December 1995.

[12] Mike Holenderski, Wim Cools, Reinder J. Bril, and Johan J. Lukkien. Extending an open-source real-time operating system with hierarchical scheduling. Technical report, Eindhoven University of Technology, October 2010.

[13] Mike Holenderski, Martijn M. H. P. van den Heuvel, Reinder J. Bril, and Johan J. Lukkien. Grasp: Tracing, visualizing and measuring the behavior of real-time systems. In *Proc. 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pages 37–42, July 2010.

[14] Damir Isovic. *Flexible Scheduling for Media Processing in Resource Constrained Real-Time Systems.* PhD thesis, Mälardalen University, Sweden, November 2004.

[15] Damir Isovic and Gerhard Fohler. Handling mixed task sets in combined offline and online scheduled real-time systems. *Real-Time Systems Journal*, Volume 43, Issue 3:296–325, December 2009.

[16] Hermann Kopetz. Event-triggered versus time-triggered real-time systems. In *Proceedings of the International Workshop on Operating Systems of the 90s and Beyond*, pages 87–101, London, UK, 1991. Springer-Verlag.

[17] Hermann Kopetz and Gnther Bauer. The time-triggered architecture. In *PROCEEDINGS OF THE IEEE*, pages 112–126, 2003.

[18] Jean J. Labrosse. *MicroC/OS-II.* R & D Books, 2nd edition, 2002.

[19] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *proceeding of Real Time Systems Symposium*, pages 166–171, 1989.

[20] J. Y. T Leung and J. Whitehaed. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, Volume 2, Issue 4:237–250, 1982.

[21] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, January 1973.

[22] OSEK. *OSEK/VDX,Operating System Manual, Version 2.2.1*, February 2005.

[23] OSEK. *OSEK/VDX, Time-Triggered Operating System, Version 1.0*, July 2010.

[24] Michael J. Pont. *Reducing the time taken to test your next embedded system.* TTE systems, 2010.

[25] Ragunathan Rajkumar. Dealing with suspending periodic tasks. IBM T.J. Watson Research Center, 1991.

[26] John Regehr and Usit Duongsaa. Preventing interrupt overload. *SIGPLAN*, Volume 40, Issue 7:50–58, June 2005.

[27] Fabian Scheler and Wolfgang Schroeder-Preikschat. Time-triggered vs. event-triggered: A matter of configuration? *Model-Based Testing, ITGA FA 6.2 Workshop on and GI/ITG Workshop on Non-Functional Properties of Embedded Systems, 2006 13th GI/ITG Conference -Measuring, Modelling and Evaluation of Computer and Communication (MMB Workshop)*, pages 1–6, March 2006.

[28] Martijn M. H. P. van den Heuvel, Mike Holenderski, Reinder J. Bril, and Johan J. Lukkien. Constant-bandwidth supply for priority processing. In *IEEE Transactions on Consumer Electronics (TCE)*, volume 57, pages 873–881, May 2011.

Appendix A

A case study : slot-shifting with periodic tasks

In this section, we will analyze the static system properties with an example task-set. Let us consider a task set presented in Table A.1. This task-set is converted into set of intervals using the static system properties presented in Section 3.4.

Task	**Arrival Time**	**WCET**	**Deadline**	**Period**
P_1	0	2	4	20
P_2	1	6	16	20
P_3	4	2	8	20
P_4	10	2	14	20
P_5	15	1	20	20

Table A.1: An example task-set.

Based on Equation (3.3), (3.4) and (3.6), the task-set is divided into intervals. Intervals for the task-set presented in Table A.1 are given below in Table A.2.

Intervals(I_m)	**Start time** (st_m)	**End time**(e_m)	**Spare capacity**(sc_m)	**Task-set**(τ_m)
I_0	0	4	2	$\{P_1\}$
I_1	4	8	2	$\{P_3\}$
I_2	8	14	0	$\{P_4\}$
I_3	14	16	-4	$\{P_2\}$
I_4	16	20	3	$\{P_5\}$

Table A.2: Intervals for the task set presented in Table 4.1.

The hyper-period of intervals,$H_{\mathcal{I}}$ (see Equation (3.6)) is,

$$H_{\mathcal{I}} = LCM(T1, T2, T3, T4, T5) = 20 \tag{A.1}$$

The length of an interval, I_m, at the start of an interval st_m is calculated based on Equation

(3.8).

$$Length(I_0) = e_0 - st_0 = 4 - 0 = 4 \tag{A.2}$$

$$Length(I_1) = e_1 - st_1 = 8 - 4 = 4 \tag{A.3}$$

$$Length(I_2) = e_2 - st_2 = 14 - 8 = 6 \tag{A.4}$$

$$Length(I_3) = e_3 - st_3 = 16 - 14 = 2 \tag{A.5}$$

$$Length(I_4) = e_4 - st_4 = 20 - 16 = 4 \tag{A.6}$$

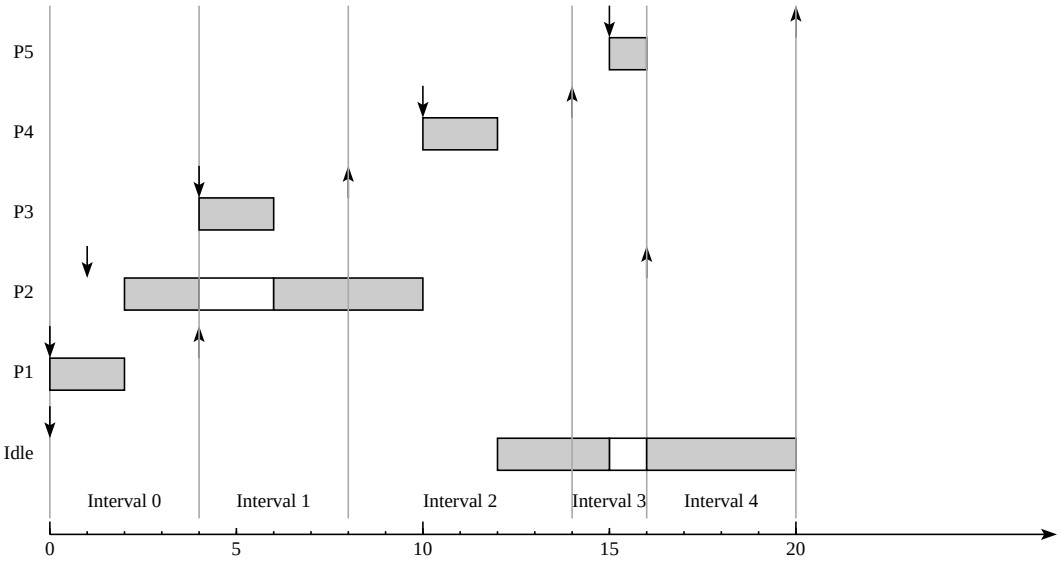

Figure A.1: Execution scenario of task-set presented in Table A.1. Only periodic tasks are considered in this scenario.

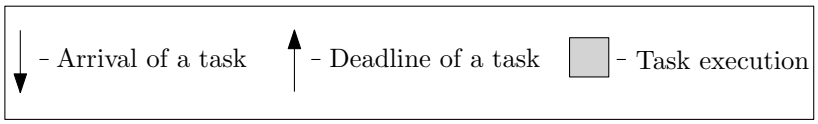

Figure A.2: Legend for Grasp trace presented in Figure A.1.

Let us assume the execution scenario without dynamic tasks, to understand the spare capacity variation during run-time. For the execution sequence presented in Figure , the variation of state variables for each time instance is presented below.

The following holds at the start(st_0) of an interval I_0:

- **Start of an interval I_0**
- $Q(0) = \{P_{1,1}\}$
- $C_1^{P,completed}(0) = 0$
- $PW(P_1, 0) = 2$ (see Equation (3.9))
- $C_{res}^0(0) = 2$ (see Equation (3.11))
- $sc_0(0) = 2$ (see Equation (3.13))
- Assume: Periodic task, $P_{1,1}$, execution at time 0

Time-slot 1. At time slot 1, the following holds:

- $Q(1) = \{P_{1,1}, P_{2,1}\}$
- $C_1^{p,completed}(1) = 1$
- $PW(P_1, 1) = 1$
- $C_2^{p,completed}(1) = 0$
- $PW(P_2, 1) = 6$
- $C_{res}^0(1) = 1$
- $sc_0(1) = 2$
- Assume: Periodic task, $P_{1,1}$, execution at time 1

Time-slot 2. At time slot 2, the following holds:

- Periodic task $P_{1,1}$ completes its execution
- $Q(2) = \{P_{2,1}\}$
- $C_2^{p,completed}(2) = 0$
- $PW(P_2, 2) = 6$
- $C_{res}^0(2) = 0$
- $sc_0(2) = 2$
- Assume: Periodic task, $P_{2,1}$, execution at time 2

Time-slot 3. At time slot 3, the following holds:

- $Q(3) = \{P_{2,1}\}$
- $C_2^{p,completed}(3) = 1$
- $PW(P_2, 3) = 5$
- $C_{res}^0(3) = 0$
- $sc_0(3) = 1$
- Assume: Periodic task, $P_{2,1}$, execution at time 3

Time-slot 4. At time slot 4, the following holds:

- **Start of an interval I_1**
- $Q(4) = \{P_{3,1}, P_{2,1}\}$
- $C_3^{p,completed}(4) = 0$
- $PW(P_3, 4) = 0$
- $C_2^{p,completed}(4) = 2$
- $PW(P_2, 4) = 4$
- $C_{res}^1(4) = 2$
- $sc_0(4) = 0$
- $sc_1(4) = 2$
- Assume: Periodic task, $P_{3,1}$, execution at time 4

Time-slot 5. At time slot 5, the following holds:

- $Q(5) = \{P_{3,1}, P_{2,1}\}$
- $C_3^{p,completed}(5) = 1$
- $PW(P_3, 5) = 1$
- $C_2^{p,completed}(5) = 2$
- $PW(P_2, 5) = 4$
- $C_{res}^1(5) = 1$
- $sc_1(5) = 2$
- Assume: Periodic task, $P_{3,1}$, execution at time 5

Time-slot 6. At time slot 6, the following holds:

- Periodic task $P_{3,1}$ completes its execution
- $Q(6) = \{P_{2,1}\}$
- $C_3^{p,completed}(6) = 2$
- $PW(P_3, 6) = 0$
- $C_2^{p,completed}(6) = 2$
- $PW(P_2, 6) = 4$
- $C_{res}^1(6) = 0$
- $sc_1(6) = 2$
- Assume: Periodic task, $P_{2,1}$, execution at time 6

Time-slot 7. At time slot 7, the following holds:

- $Q(7) = \{P_{2,1}\}$
- $C_2^{p,completed}(7) = 3$
- $PW(P_2, 7) = 3$
- $C_{res}^1(7) = 0$

- $sc_1(7) = 1$
- Assume: Periodic task, $P_{2,1}$, execution at time 7

Time-slot 8. At time slot 8, the following holds:

- **Start of an interval** I_2
- $Q(8) = \{P_{2,1}\}$
- $C_2^{p,completed}(8) = 4$
- $PW(P_2, 8) = 2$
- $C_{res}^2(8) = 4$
- $sc_1(8) = 0$
- $sc_2(8) = 2$
- Assume: Periodic task, $P_{2,1}$, execution at time 8

Time-slot 9. At time slot 9, the following holds:

- $Q(9) = \{P_{2,1}\}$
- $C_2^{p,completed}(9) = 5$
- $PW(P_2, 9) = 1$
- $C_{res}^2(9) = 3$
- $sc_2(9) = 2$
- Assume: Periodic task, $P_{2,1}$, execution at time 9

Time-slot 10. At time slot 10, the following holds:

- Periodic task $P_{2,1}$ completes its execution
- $Q(10) = \{P_{4,1}\}$
- $C_4^{p,completed}(10) = 0$
- $PW(P_4, 10) = 2$
- $C_{res}^2(10) = 2$
- $sc_2(10) = 2$
- Assume: Periodic task, $P_{4,1}$, execution at time 10

Time-slot 11. At time slot 11, the following holds:

- $Q(11) = \{P_{4,1}\}$
- $C_4^{p,completed}(11) = 1$
- $PW(P_4, 11) = 1$
- $C_{res}^2(11) = 1$
- $sc_2(11) = 2$
- Assume: Periodic task, $P_{4,1}$, execution at time 11

Time-slot 12. At time slot 12, the following holds:

- Periodic task $P_{4,1}$ completes its execution
- $Q(12) = \{\emptyset\}$
- $C_4^{p,completed}(11) = 2$
- $PW(P_4, 11) = 0$
- $C_{res}^2(11) = 0$
- $sc_2(12) = 2$
- Assume: Idle task execution at time 12

Time-slot 13. At time slot 13, the following holds:

- $Q(13) = \{\emptyset\}$
- $C_{res}^2(13) = 0$
- $sc_2(13) = 1$
- Assume: Idle task execution at time 13

Time-slot 14. At time slot 14, the following holds:

- **Start of an interval** I_3
- $Q(14) = \{\emptyset\}$
- $C_{res}^3(14) = 0$
- $sc_3(14) = 2$
- Assume: Idle task execution at time 14

Time-slot 15. At time slot 15, the following holds:

- $Q(15) = \{P_{5,1}\}$
- $C_5^{p,completed}(16) = 0$
- $PW(P_5, 16) = 1$
- $C_{res}^3(15) = 0$
- $sc_3(15) = 1$
- Assume: Periodic task, $P_{5,1}$, execution at time 15

Time-slot 16. At time slot 16, the following holds:

- Periodic task $P_{5,1}$ completes its execution
- **Start of an interval** I_4
- $Q(16) = \{\emptyset\}$
- $C_5^{p,completed}(16) = 1$
- $PW(P_5, 16) = 0$
- $C_{res}^4(16) = 0$
- $sc_4(16) = 4$
- Assume: Idle task execution at time 16

Time-slot 17. At time slot 17, the following holds:

- $Q(17) = \{\emptyset\}$
- $C^4_{res}(17) = 0$
- $sc_4(17) = 3$
- Assume: Idle task execution at time 17

Time-slot 18. At time slot 18, the following holds:

- $Q(18) = \{\emptyset\}$
- $C^4_{res}(18) = 0$
- $sc_4(18) = 2$
- Assume: Idle task execution at time 18

Time-slot 19. At time slot 19, the following holds:

- $Q(19) = \{\emptyset\}$
- $C^4_{res}(19) = 0$
- $sc_4(19) = 1$
- Assume: Idle task execution at time 19

Time-slot 20. End of hyper-period

The execution scenario above presents the variation of system parameters during run-time. This sequence also presents how the spare capacity becomes positive from a initially negative. Since, the periodic task P_2 belongs to an interval I_3, the spare capacity of an interval is updated during run-time, when a task executes outside its own interval I_3. The off-line computed spare capacity changes during run-time, as the tasks are dynamically executing outside their intervals. The spare capacities and the reserved capacities need to be computed dynamically at the start of each interval.

For all intervals, the start-time, the end-time and the task-set belongs to intervals are the static parameters. With these static parameters, the spare capacity and the reserved capacity is calculated. In order to compute the actual reserved capacity of an interval, we need to keep track of the pending work of a periodic task. The execution of each task must therefore be tracked in order to obtain the reserved capacity, and derived from that, the spare capacity. For example, see the variation in parameter from time-slot 7 to time-slot 8 in the execution sequence presented above. At time-slot 7, the spare capacity of the interval I_2, sc_2 is 3. While, at time-slot 8, at the start of an interval I_2, it is computed as 2 using Equation (3.13). This is a side-effect of the borrowing mechanism, which updates the spare capacity at the start of each interval.

Appendix B

RELTEQ Revisited

RELTEQ, an efficient time representation mechanism, developed for different kinds of timed events such as period event, deadline event and so on. RELTEQ stores the arrival times of future events relative to each other, by expressing their time relative to their previous event. The arrival time of the head event is relative to the current time [12], as shown in Figure B.1.

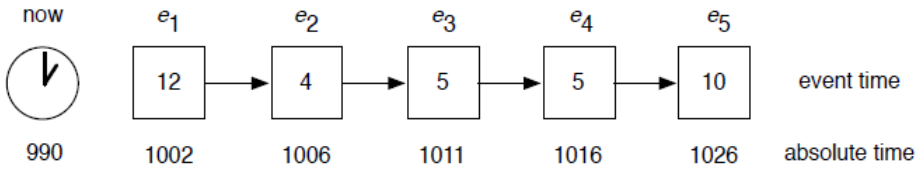

Figure B.1: Example of the RELTEQ event queue.

A RELTEQ event is specified by the tuble (kind, time, handler). The kind represents the event kind, e.g. a delay or the arrival of a periodic task. time is the event time. handler points to additional data that may be required to handle the event and depends on the event kind. For instance, a delay event will point to the task which is to be resumed after the delay event expires.

More detailed description on RELTEQ can be found in [12]. In this appendix, the purpose of RELTEQ mechanisms and its functionality is described briefly. This appendix can be considered as a cook book to implement functionalities on top of RELTEQ.

B.1 How to create a new RELTEQ queue?

A RELTEQ queue is a list of RELTEQ events. When the event at the head of the queue expires, the event handler is invoked. The event handler depends on the kind of event. For example, a delay event will resume the delayed task. In order to create a new RELTEQ, the following mechanisms should be considered.

- RelteqQueueCreate() function is used to create a new RELTEQ queue. The new RELTEQ queue is created from the free list of queues. This function is defined in os_relteq_core.c.
- In the configuration file, os_cfg.h, the number of queues supported by the RELTEQ is defined by the OS_RELTEQ_NUM_CUSTOM_QUEUES preprocessor directive. This parameter must be properly defined to create additional RELTEQ queues. By default, the activation of RELTEQ creates the RELTEQ system queue to deal with delay events. The integration of new mechanism on top of RELTEQ might require dealing with different events and queues. In such circumstances, this parameter must be configured properly to create new RELTEQ queues.

B.2 How to activate/deactivate a new RELTEQ queue?

A RELTEQ queue is used to store events. Each event has an expiration time associated with it. Upon expiration, the event handler is invoked to perform actions, based on the event type. In order to deal with a RELTEQ event and to keep track of event expirations, the RELTEQ queue must be activated. The queue will not be updated with the deactivation of a RELTEQ queue.

- RelteqQueueActivate() function activates the queue to keep track of event expirations. This function is defined in os_relteq_core.c.
- RelteqQueueDeactivate() function deactivates the queue and the events in the queue will not be updated, after the deactivation. This function is defined in os_relteq_core.c.

B.3 How to create a new RELTEQ event?

A RELTEQ event is created with the type and the handler. The type of an event can be period, delay, deadline etc. Each event must have an handler associated with it. On the expiration of a event, the event is handled by performing the actions described in the handler.

- RelteqEventCreate() function is used to create a RELTEQ event. The creation of a event requires the event type and the pointer to the handler. This function is defined in os_relteq_core.c.
- RelteqEventDel() function is used to delete a event. After handling the event, it is always safe to delete the expired event. This function is often invoked in the event handler to delete the event. This function is defined in os_relteq_core.c.

B.4 How to insert and delete an event from a queue?

After the creation of an event, they are inserted into the queue. Once the event becomes part of the queue, the RELTEQ manages the event by keeping of expiration of events in the queue. In the same way, the event can be deleted from the queue.

- RelteqQueueInsertEvent() function is used to insert a RELTEQ event into a RELTEQ queue. This function takes a RELTEQ event and a RELTEQ queue as an argument. The incorrect arguments would lead to system termination via ASSERT function. This function is defined in os_relteq_core.c.
- RelteqQueueDeleteEvent() function is used to delete a RELTEQ event from a RELTEQ queue. This function takes a RELTEQ event and a RELTEQ queue as an argument. The event passed as an argument is deleted from the queue specified in the argument. This function is defined is os_relteq_core.c.

B.5 RelteqTimeTick function

The RelteqTimeTick function is invoked from the tick handler of the μC/OS-II [18]. The RelteqTimeTick function is responsible for handling the timed events. The head of active RELTEQ queues are updated in each time tick. As a consequence, events may expire in RELTEQ queues. All the expired events are handled as part of the RelteqTimeTick function.

- RelteqSyncWithCurrentTime function is responsible for updating all the active queues in the system. In addition, the expired events are handled as part of this function. The overhead of the RelteqTick handler comes from the execution of the RelteqSyncWithCurrentTime. The overhead increases with the increase in the number of active queues and the number of expiring events in each of these active queues. This function must be used carefully to reduce the overhead. While developing extension on top of the RELTEQ framework, a special care must be taken to deal with the overhead caused by the RelteqSyncWithCurrentTime function.

B.6 Discussions

The purpose of this Appendix is to provide an high-level overview of the RELTEQ framework. For the integration of new run-time mechanisms, the function calls presented above are highly essential. To properly understand the dependencies involved with the RELTEQ framework implementation, the careful analysis of the os_relteq_core.c, os_relteq_port.c, os_relteq_port.h and os_relteq.h functions are essential.

The RELTEQ framework provides very useful mechanisms to add new rum-time features. The proper understanding of the framework is essential to develop efficient run-time mechanisms with less run-time overhead.

Appendix C

Slot-shifting cook book

In this Appendix, we present the programming APIs and the configuration details of slot-shifting scheduling in μC/OS-II.

C.1 Programming API

We present the programming APIs for the EDF scheduler and the slot-shifting scheduling in μC/OS-II. All parameters are of type INT8U. The OSTCBPrioTbl gives the pointer to a TCB of a task with the help of a task priority.

C.1.1 EDF scheduler

The EDF scheduler is implemented on top of the RELTEQ framework. For scheduling of tasks based on the absolute deadline, the deadline of a task is inserted into the ready-queue. During run-time, deadlines of tasks must be maintained in RTOS. We introduced a new API to store task deadline in the TCB of a task.

```
OSTaskSetDeadline(OSTCBPrioTbl[TASK_PRIORITY], TASK_DEADLINE );
```

The EDF scheduling in μC/OS-II works with the RELTEQ and the OS_RELTEQ_EDF option enabled in OS_CFG.h. In order to store the deadline of a task in its TCB, we created the OSTaskSetDeadline function. The OSTCBPrioTbl[TASK_PRIORITY] acts as the pointer to the TCB of a task. TASK_DEADLINE parameter passed as the second argument is the relative deadline of a task.

C.1.2 Slot-shifting scheduler

For the scheduling of tasks using the slot-shifting scheduling technique, an RTOS should maintain information regarding tasks in the system and the set of intervals associated with the task set. We developed APIs for storing the task attributes in TCB and the inter-

val attributes in ICB. For these APIs to work and to enable slot-shifting in μC/OS-II, OS_SLOT_SHIFTING_EN should be enabled (set to 1) in OS_CFG.h.

- The slot-shifting run-time mechanisms requires the information regarding the type of a task and the WCET of a task. We developed a new API to store these informations in a TCB.

  OSTaskSetParam(OSTCBPrioTbl[TASK_PRIORITY], TASK_TYPE, TASK_COMP.TIME);

  The OSTCBPrioTbl[TASK_PRIORITY] acts as the pointer to the TCB of a task. The TASK_TYPE parameter indicates the type of a task. The *type* can be Periodic, Aperiodic, or Sporadic [1]. The TASK_COMP.TIME parameter takes the WCET of a task. With the help of this API, the **type** of a task and the **WCET** of a task is stored in a TCB.

- The slot-shifting scheduling algorithm requires tracking of intervals during run-time. The interval attributes must be maintained in RTOS to perform interval-tracking. We created ICBs to maintain interval attributes. In order to register intervals in a ICB, we created an API.

  OSIntervalCreate(START_TIME, END_TIME, SPARE_CAPACITY, NUMBER_OF_TASKS, POINTERS_TO_TCBs..);

  The interval associated information such as start time of an interval, end time of an interval , the spare capacity of an interval and the tasks belongs to the interval are stored in a ICB by using the OSIntervalCreate() function. This function is the variable argument function. Number of arguments of this function is decided by the number of tasks belonging to an interval. We present an example below to explain the use of OSIntervalCreate() function.

Interval	Start	End	Spare Capacity	Task set
I_0	0	3	2	{TASK1, TASK2}

Table C.1: Example - Interval Creation.

Interval can be created for the interval shown in Table C.1 as shown below:

OSIntervalCreate(0, 3, 2, 2, OSTCBPrioTbl[TASK1_PRIORITY], OSTCBPrioTbl[TASK2_PRIORITY]);

The number of intervals used in the application should be defined on OS_CFG.H. OS_MAX_NO_OF_INTERVALS should be configured with the number of intervals used in the application.

[1] We support only periodic and sporadic tasks currently with the slot-shifting, aperiodic tasks handling is considered as a future work.

C.2 Configuration details

In this section, we present the configuration details of the EDF scheduler and the slot-shifting scheduler. Functions presented below should be invoked for each task in the application.

C.2.1 Configuration of EDF scheduler

1. In the OS_CFG.h, we added the following flags:
   - The OS_RELTEQ_EN flag enables the RELTEQ framework.
   - The OS_RELTEQ_PERIODIC_EN is enabled to use the periodic task extension based on the RELTEQ framework.
   - The OS_RELTEQ_EDF_EN flag enables the EDF scheduler.
2. Create tasks with the OSTaskCreate() function.
3. Invoke OSTaskSetPeriodEX() function to set the period and the offset of a task.
4. Invoke OSTaskSetDeadline() function to store the deadline of a task in the TCB.

C.2.2 Configuration of slot-shifting scheduler

1. In addition to the flags discussed in Section C.2.1 for the configuration of EDF scheduler, the OS_SLOT_SHIFTING_EN flag should be enabled to activate slot shifted scheduling.
2. Create tasks with the OSTaskCreate() function.
3. Invoke OSTaskSetPeriodEX() function to set the period and the offset of a task.
4. Invoke OSTaskSetDeadline() function to store the deadline of a task in the TCB.
5. Invoke OSTaskSetParam() function to store the type and the WCET of a task in the TCB.
6. Invoke OSIntervalCreate() function to store interval associated information in an ICB.

Appendix D

Software Versions

In this appendix, we present the software and hardware tools used in our work.

- μC/OS-II RTOS-2.84
- GNU binutils-2.18.50
- GNU GCC-4.2.2
- GNU GDB-6.8
- or1ksim-0.3.0
- Grasp visualization tool-2011.07.13.1744