Temporal Partitioning of Flexible Real-Time Systems

Attila Zabos

PhD

2011

Abstract

In real-time systems it is important to make the best use of the available processor capacity, and to provide and ensure timing guarantees during the entire runtime of the system. Systems with safety aspects, like in the automotive and aerospace domain rely on the system's compliance to temporal specifications. In order to ensure that temporal specifications are respected and misbehaving applications do not jeopardise the execution of other applications in the systems, the execution of applications is constrained by temporal partitions. The usual mechanism to enforce temporal partitioning of the processor time is accomplished by execution-time servers. However, the choice of appropriate server parameters such that the application demand is satisfied but the processor allocation is kept at a minimum, is not trivial. In the past, for dynamic as well as for fixed-priority scheduled servers, time-consuming and approximation methods were presented, enabling the determination of the server's temporal parameters with the objective to minimise the processor time allocation.

The work in this thesis focuses on open fixed-priority scheduled systems since they still represent the majority of real-time systems used in the automotive or aerospace domain and explores methods to determine the minimal processor reservation for a given fixed-priority scheduled set of flexible tasks. It is assumed that the processor reservation is maintained and enforced by periodic execution-time servers that are also scheduled according to a fixed-priority scheduling scheme. Since we consider flexible tasks in the server parameter determination, the servers themselves will enable and support flexible temporal partitioning during runtime. An appropriate runtime algorithm is also presented to address this flexibility and to utilise spare processor capacity by maximising the processor usage.

The efficiency and effectiveness of the presented algorithms is examined and compared to existing methods by performing empirical evaluation on a desktop computer and an embedded hardware platform.

Contents

[1]The experiments in this thesis are consecutively numbered with a unique number assigned to each of them.

List of Figures

List of Tables

List of Algorithms

Acknowledgements

It takes a long time to finish a Ph.D. and I would like to express my deepest gratitude to people without whom this work would not have been possible.

I am deeply grateful to my supervisor Professor Alan Burns for his constant support, invaluable advice and feedback, encouragement throughout all the years of my research and giving me the opportunity to undertake research at the University of York. Without his support, this thesis would not have developed.

I also wish to express my particular thanks to my colleague Dr. Robert I. Davis, for his excellent guidance to conduct high quality research, the fruitful discussions about research problems and for always taking time to review my work and to provide valuable feedback.

I am very thankful to Dr. Guillem Bernat for the initial supervision of my research, for giving me an understanding of important aspects and practices of scientific research, and for introducing to me useful scientific tools and approaches.

I sincerely thank Professor Andy Wellings for his critical view and constructive feedback on my work.

Special thanks go to Rapita Ltd. as well, for providing the tools and hardware that allowed me to perform worst-case execution time (WCET) measurements and analysis on the developed algorithms.

I am also grateful to Professor Michael González Harbour for his input on flexible real-time system, Dr. Adam Betts for his help with the WCET analysis tools and his advice from a research student's point of view, Dr. Jack Whitham and Alexander Vajnov for the discussions about execution time measurement, Dr. Alexandros Zerzelidis for our discussions about various research topics during the long nights in the office, and all the members of the Real-Time Systems Group for the great working atmosphere at the University of York. Special thanks go also to Filo and Sue for taking care of a lot of administrative and organisational issues, and always helping me with their good advice.

I am very thankful to my nearest relatives, especially my partner Michaela, and my friends, especially Carsten and Sashi, for their support, encouragement and understanding during this stressful and exhaustive time. I am especially thankful to my dad for his never ending support and encouragement, and for taking a lot of load of my shoulders during all the years of my study. Thank you!

Declaration

I declare that, unless otherwise indicated in the text, the research work presented in this thesis is original work that I undertook between October 2005 and August 2011. As listed below, the content of this thesis has been partially drawn from work that was previously published as technical reports, conference proceedings or journal publications.

Chapter 1 contains material from a technical report [102], a paper [103] presented at the *International Conference Real-Time and Network Systems* and the author's Qualifying Dissertation.

Chapter 2 also contains general information about real-time systems from the author's Qualifying Dissertation, the conference paper [103] and some initial work done on server temporal parameter selection as part of the FRESCOR project [33].

Chapter 3 contains parts of the system model definition presented in the FRESCOR project deliverable [33]. The demand points based approach was also briefly mentioned in the deliverable [33].

Chapter 4 contains also content that was published in the FRESCOR deliverable [33].

Chapter 5 contains material that was published as joint work [32] in the *IEEE Transactions on Computers*. The author of this thesis mainly contributed to the publication by providing the evaluation of the different approaches to support the decision about the most valuable optimisation method for an efficient online schedulability test. The description of embedded hardware test set-up is based on the conference paper [103].

The main part of Chapter 6 is based on the conference paper [103].

The conference proceeding [103] and the sections of the FRESCOR deliverable [33] that were incorporated into this thesis were written by the author of this thesis with advice from Professor Alan Burns and Dr. Robert I. Davis.

Chapter

1

Introduction

The correctness of real-time system depends not only on the logical result of the computation but also on the time instant at which the result was produced. That means, the performed operations of the system need to comply with predefined temporal specifications.

During the last few decades, the complexity of real-time systems experienced a continuous increase [19] that can be also observed in the trend of research topics addressing problems in the area of temporal partitioning [57, 59, 88, 95–97], and hierarchical and flexible scheduling [5, 21, 22, 27, 35, 36, 53, 55, 64, 74, 87].

The classical design of complex real-time systems usually contains several Electronic Control Units (ECUs), each executing a single application. Applications are considered as a collection of real-time tasks that provide a certain system function. For example, in robotics there are dedicated applications responsible for path planning, motor control, image processing and communication. In the aerospace or automotive area the decomposition of the real-time system usually results in applications for engine, brake and attitude control, to name but a few. With this distributed architecture in mind, the continuously increasing feature set of modern real-time systems would result in complex distributed systems with a growing number of hardware components and ECUs, and more difficult design, verification and maintenance of these system.

To reduce the need for complex distributed systems, it is a common research approach to utilise integrated open and flexible real-time systems that combine multiple system properties:

- enabling the composition of multiple application by *integrating* them on the same platform,
- keeping the system *open*, enabling running applications to leave the system and also the submission of new applications into the system during runtime, and
- facilitating the *flexibility* of the system by supporting the specification of multiple rather than a single set of temporal parameters (i.e. worst-case execution time, period and deadline) for the tasks.

1.1 Integrated open real-time systems

The increasing processing power and flexibility of modern hardware counteracts the increasing number of required hardware components and ECUs in real-time systems by facilitating the integration of multiple real-time applications on a single uniprocessor platform. This approach allows the designer to address cost, space, weight and energy constraints of real-time systems by reducing the hardware costs, the number of ECUs in a system and the power consumption of the system. At the same time the analysis, design and development processes of these real-time systems do not become over complicated since complex distributed system architectures can be avoided.

Our focus is on modern *integrated open real-time systems* that enable the offline and online composition of multiple applications (including legacy applications that have been developed and verified on a dedicated platform and might already have been in production) with their own scheduling policy. In these systems multiple concurrently executed real-time applications are usually scheduled by a fixed-priority *two-level hierarchical scheduler*.

The two-level hierarchical scheduler consists of an operating system level or *global scheduler* and an application level or *local scheduler* for each application (see Figure 1). The global scheduler decides the processor time allocation to

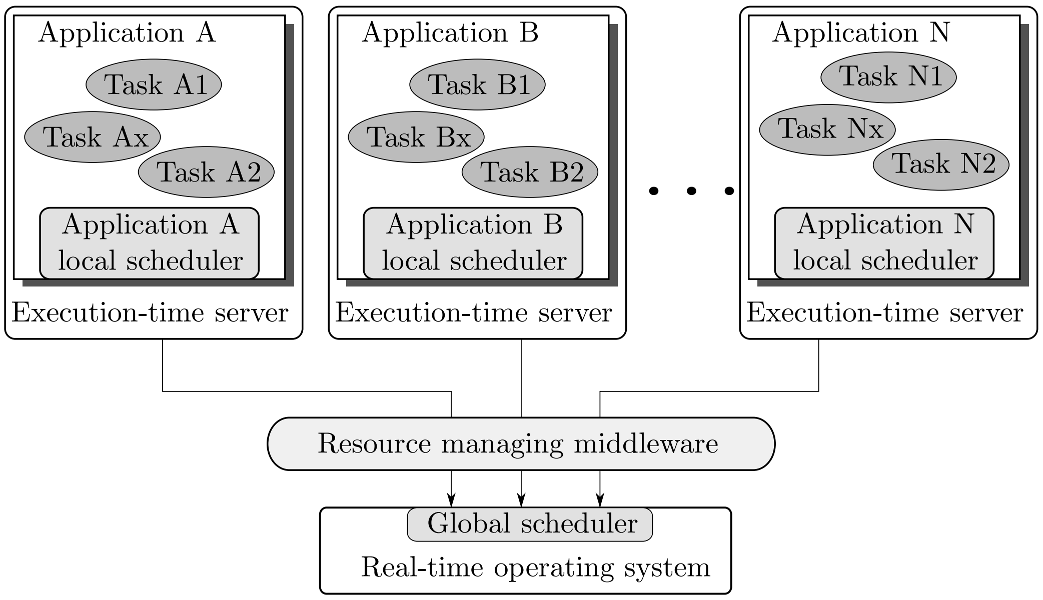

Figure 1: Two-level hierarchical open real-time system [36, 37, 64]

applications, and the local scheduler determines which task of the currently scheduled application shall execute.

In addition to legacy applications, the development and verification of new applications may still be performed by different developers, in isolation from the other applications. Since different applications are usually responsible for different system functions, fault containment has to be ensured. One major element of fault containment is temporal isolation of applications, in order to limit timing errors to the originating application.

Industry standards like ARINC 653 [2], AUTOSAR [10] or the research project DECOS [77] already specify the need for temporal partitioning among applications. Applying temporal partitioning in real-time systems must prevent timing faults caused by erroneous applications having a side-effect on other applications in the system and jeopardising their operation.

However, current real-time operating systems, that are compliant to the aforementioned standards (i.e. ARINC 653, AUTOSAR), provide only static temporal partitioning [60]. That means, the processor time partitioning is determined before runtime and remains unaltered during the lifetime of the system.

1.2 Flexible real-time systems

It is expected that in the next generation of real-time systems, the temporal properties and the number of applications will vary during runtime [21, 35, 37, 53, 70]. That means, the tasks of a *flexible application* provide implementations that can adapt their execution to the available processing resources. These tasks can execute at different frequencies (i.e. have variable period) and/or may demand variable processing time (i.e. by specifying variable worst-cases execution times) depending on the accuracy of the executed algorithm. This implies that the task's, and consequently the corresponding application's *Quality of Service* (QoS) depends on the amount of processing resource that is reserved for the application in certain time intervals. With the objective in mind, to maximize the processor utilisation and avoid unused processing time reservations (as would occur with static temporal partitioning), the applications are started on demand and terminate if their services are not required anymore. At these changing scenarios the available processor capacity has to be redistributed among the remaining applications in the system.

The dynamic behaviour of the envisaged open real-time systems, executing flexible applications, prevents the use of an efficient static temporal partitioning of the processor time. A static temporal partitioning that lasts over the entire lifetime of a system would result in an oversized system. Hence, in order to efficiently use the processing time, the flexibility of applications and possible demand changes during runtime have to be considered in the offline analysis as well as during runtime.

Despite the flexibility of the applications, it is desirable to maintain predictability of integrated open real-time systems. Partitioning of the processing time has to be accomplished such that the applications' temporal requirements are still respected and remain valid even if they are executed in an integrated open system. Similar to classical real-time systems, the schedulability of real-time tasks shall be guaranteed before they are executed.

To support the temporal partitioning in the system, the application's temporal requirements are mapped to an *execution-time server* that represents the processor reservation and manages the consumption of processor time by the tasks of the associated application [4, 47, 64, 71–73]. In the remainder of this thesis, the terms execution-time server and the simpler form *server* will be used interchangeably. Our focus is in particular on periodic servers, that are a common means for resource management in fixed-priority scheduled real-time systems. Periodic servers provide a certain amount of processing time in a periodic time interval to applications.

In order to achieve an efficient temporal partitioning, the temporal parameter values of a server are derived from the timing requirements of the associated application's tasks, such that the resource reservation for the application is minimal. The server parameter values leading to minimal resource reservation are considered as the *optimal* parameter values. Furthermore, for flexible applications there are multiple optimal temporal parameter values, depending on the flexibility of the application's timing requirements. The set of optimal parameter values can then be utilised during runtime. By varying the server parameter values within the predetermined limits or set of values, the temporal partitioning of the system can be affected in a way that the processor capacity distribution among the active applications in the system is maximized.

The distribution of the processor capacity is considered [47] to be a system function provided by a middleware layer responsible for resource management on top of a real-time operating system. Based on the server parameter values, the resource managing middleware determines, during runtime, the resource reservation for each application. Consequently, with the support of resource managing middleware, the applications can adapt their execution to the available processing resource.

Since an unaltered temporal partitioning of the processing time is not efficient for uniprocessor open real-time systems that execute flexible real-time applications, this thesis is concerned with temporal partitioning before and during runtime.

1.3 Motivation

The efficient use of processing time in an open real-time system, where the processor time is partitioned among the concurrently executed applications, starts with the selection of optimal server parameters.

The server parameters are referred to as *optimal* if a server supplies only as much processing time as required to finish the execution of the associated application's tasks before their deadlines, in other words satisfying the timing and demand requirements of the corresponding application task. Thus, the processor utilisation is considered as the cost function for the optimisation process.

Our aim is to reserve only the minimal required processing time for each application such that its associated tasks can finish their execution within their timing constraints. In addition, as much as possible of the processor capacity shall be utilised during runtime by the flexible real-time applications.

Flexible system behaviour can be supported by implementing algorithms to control devices or machines, but also incremental anytime algorithms enabling the progressive improvement of calculated results and permitting to interrupt their execution any time. For example, control algorithms can execute at variable frequency [70] consequently influencing the quality of the control output by changing the corresponding task period, or the accuracy of the results produced by anytime algorithms, like path planning, may vary with the granted execution time or in other words with the reserved processing time [104]. These flexible tasks provide support for open real-time systems, allowing to maximise application quality of service via maximising the processor utilisation during runtime. In contrast to classical real-time systems, the temporal specification of the corresponding tasks provide multiple or a range of timing constraints. Whereby, the variable task parameter values entail a variable quality of the produced results. Depending on the purpose of a real-time application, different QoS attributes might be used to represent the quality of the generated results.

Irrespective of the meaning of QoS attributes, for the optimal server parameter selection it is assumed that the more processing time an application receives the higher is its QoS. This implies that the QoS delivered by an application varies with the temporal parameter values of the associated server and the resulting processing time reservation.

In open real-time systems where the processor load changes with respect to the number of active applications, it is essential to determine multiple sets of server parameters for each of the flexible applications. This flexibility allows the resource managing middleware to adapt variable server parameter values to the prevalent processor load during runtime. The objective of such an online parameter adaptation is to maximise the processor utilisation by distributing the processor capacity among the active execution-time servers and consequently increasing the QoS while the created schedule for the active servers remains feasible.

The main concerns with existing runtime adaptation mechanisms are that they:

- leave the processing time and timing requirements of the adaptation mechanism in real-time systems out of consideration [86], hence do not make any statement about the execution time of such a task,
- limit the online adaptation to a single temporal parameter [21, 22] or,
- do not consider dynamic systems [5], like open real-time systems.

Therefore, with respect to the various timing constraints of real-time systems, there is a need for an efficient but also predictable online algorithm to adapt the server parameters to changing system state, as in open real-time systems.

1.4 Thesis proposition

The focus of this thesis lies on the efficient temporal partitioning of processing time, and the thesis proposition can be stated as follows:

Open flexible real-time systems can adapt their processing to the changing system load, facilitating the efficient utilisation of the available processing time. Offline and online optimal execution-time server parameter selection for flexible real-time applications is achievable and provides the foundation for the efficient usage of processing resources.

1.5 Thesis contribution and outline

The set of contributions that are presented in this thesis support the thesis proposition and are summarized in the following.

Chapter 2 provides the definition of various terms and notations used in this thesis. A critical review of relevant research topics in the area of temporal partitioning, server parameter selection, runtime parameter adaptation, and links between our work and these topics are also given. Finally design specific details for applications are provided. An application skeleton using the features of the envisaged middleware, is presented.

Chapter 3 starts with the introduction of the underlying system model. An improved representation, to express the application's processing requirements at certain time instants, referred to as *demand points*, is also introduced in Chapter 3. Additionally, a simple algorithm is defined that enables the calculation of a lower and upper bound on the optimal server period. This interval ensures the existence of the optimal server period value and limits the search space for the optimal value. It shall be noted that the optimal server period is not necessarily smaller than the smallest task period or its deadline. We also show that the server's temporal parameter values, using the server period upper bound value and the corresponding minimal server capacity, lead to sufficiently good parameter values though they might not always denote the optimal values.

Using the demand points, a method is presented to derive the partition parameters of a periodic *execution-time server* that is used as a resource reservation mechanism. The objective is to select server temporal parameters such that the reserved processor utilisation is minimised. In contrast to previous approaches, we utilise an exact server supply bound function and avoid inaccuracy due to a linear supply model. Furthermore, our analysis requires only the taskset's temporal specification, enabling the calculation of optimal server parameters without the intervention of the application designer. In Chapter 4 an iterative method, based on the server period upper bound, is defined to determine the optimal server temporal parameter values. The iterative method starts with the temporal parameter values at the server's period upper bound and consecutively examines possible optimal parameter candidates. Hence, the search space is significantly reduced in contrast to a brute-force approach.

The flexibility of the tasks, expressed by the specification of variable worst-case execution time and variable requirements for task period and deadline, enable the generation of demand point sets that capture the taskset demand for various situations. The methods to determine the server's optimal temporal parameters (as presented in Chapter 3 and Chapter 4) can be easily applied to flexible applications. An extended approach to derive the server parameters for flexible applications is also examined in Chapter 4.

Since in open real-time system the schedulability test represents a crucial component of the online acceptance test, various optimisation methods to improve the performance of an online test are presented and evaluated in Chapter 5.

In the envisaged systems, the schedulability test is, however, integrated into a more sophisticated component that exploits the flexible property of certain applications. Given the flexible applications and variety of possible temporal parameters, in Chapter 6 an online algorithm targeted for a middleware implementation is introduced to distribute the available processing resource among the servers in the system. The online algorithm, also denoted as *spare capacity distribution* (SCD) algorithm, incorporates the acceptance test for a dynamic open environment. The acceptance test of new applications arriving in the system is an atomic part of the SCD algorithm. The result of this test is implied in the updated spare processor capacity distribution for the prospective set of servers (including the servers of the new applications). Particularly, servers that fail the acceptance test, will not be part of the new server set and will not receive any processor capacity.

The efficiency of every step, from the offline server parameter selection to the online capacity distribution is evaluated at the end of each chapter that introduces the corresponding method. The evaluations ensure that the presented methods are applicable in real world applications.

The research work presented in this thesis is summarized in Chapter 7. Conclusions are drawn from the set of research contributions and prospective future work is considered.

Chapter 2

Real-time systems and temporal partitioning

The classical real-time multitasking model [67] was developed to verify the temporal correctness of tasksets executed on comparatively simple ECUs. Usually these simpler ECUs accomplish a single functionality of a more complex system. In these systems the concurrently executed tasks belong to a single application and share the processor as a resource.

With the increased performance of microprocessors and microcontrollers, developers had the option to integrate multiple tasksets that were previously executed on multiple ECUs onto a single uniprocessor platform [10, 82, 101]. In order to enable the verification of concurrently executed multi-application systems with real-time constraints the classical multitasking model was extended by the hierarchical system model [35, 36, 53, 87].

In hierarchically scheduled systems not only multiple tasks but also multiple applications share the processor as a resource. The hierarchical model provided the theoretical means for the verification of integrated real-time systems but in order to realise hierarchical scheduling, an appropriate resource management mechanism was required. Execution-time servers provide the necessary means for resource management and temporal partitioning in uniprocessor systems, and facilitate hierarchical scheduling.

Based on the classical multitasking model and the basic idea of hierarchical scheduling, the FRESCOR[2] project introduced an extended system model which enables a certain flexibility in the specification of temporal parameters. The results presented in this thesis are based on some of the fundamental specifications of the FRESCOR project.

One important aspect of multi-application multitasking real-time systems is temporal partitioning and the selection of appropriate execution-time server parameters. Temporal partitioning of the processor time such that all application tasks can satisfy their temporal requirement is not trivial. The review of previous work in the area of server parameter selection will reveal the opportunity for further optimisation and extension of existing approaches for flexible open real-time systems.

The content of the following sections in this chapter is twofold. First, we will introduce the necessary notation and fundamental concepts that are required for the understanding of the presented work. These concepts also form the basis for the assumed fixed-priority scheduled hierarchical open real-time system model. Second, a review of important works that address related problems or support the solution of the issues addressed by this thesis is presented. In order to cover the background knowledge required for the understanding of the approached challenges and to introduce the important aspects of the envisaged systems, the following topics will be reviewed in this chapter:

- classical *real-time tasking model*, introducing basic notations and definitions for real-time systems,
- *hierarchical and open real-time systems*, providing the foundation for integrated multitasking multi-application systems,
- *temporal partitioning*, enabling temporal containment and protection for applications via resource management mechanisms,
- *flexible real-time system*, introducing variable temporal parameters,
- uniprocessor fixed-priority *schedulability analysis*, to determine if a set of tasks and applications can be feasibly scheduled,

[2] *Framework for Real-time Embedded Systems based on COntRacts.* Available at: http://www.frescor.org [Accessed: 18 July 2011].

- *mode-change protocols*, managing the dynamic behaviour of flexible applications in open real-time systems,
- *parameter selection* for execution-time servers, used for the implementation of a resource management mechanism that enforces temporal partitioning,
- *compositional frameworks*, introducing an abstraction step for the design of multitasking and multi-application real-time systems, in order to hide unnecessary specification details at the appropriate design levels.

2.1 Real-time tasking model

In classical real-time theory an application A_q contains a set of n_q tasks, $A_q := \{\tau_1, \ldots, \tau_{n_q}\}$. However, the number of tasks in any two applications in the system may be different.

Each unit of work that is scheduled and executed by the system is referred to as a *job* and a set of related jobs that provide a system function is called a *task* [68].

The underlying real-time tasking model is based on the model that was introduced by Liu and Layland [67]. Within an application A_q, each task τ_i is defined by the tuple (C_i, T_i, D_i), with C_i denoting the task's worst-case execution time and D_i its relative deadline. Based on the task's arrival pattern, T_i denotes either the period of a *periodic task* or the minimum interarrival time of a *sporadic task*. The task's deadline may be less than or equal to its period (i.e. $D_i \leqslant T_i$). Furthermore task deadlines can be more specifically denoted as *hard* or *soft* deadlines. Tasks with a soft deadline are still able (at least partially) to contribute to the system provided functionality, although they have not finished execution by their deadline. On the other hand, tasks with a hard deadline are not able to generate any valuable results if they do not complete execution before their deadline. In this thesis we focus on tasks with a hard deadline due to the envisaged domain of automotive, aerospace and robotic applications.

To select and to assign processing time to the tasks in a given taskset, the use of a fixed-priority scheduler is assumed. The task priority is expressed by the subscript i, with the value 1 representing the highest, and n_q the lowest priority. A common method to determine the priority of each task, is the *deadline monotonic priority ordering (DMPO).* This method ensures unique priority assignment to each task within a single application taskset.

In uniprocessor real-time systems usually there are also various resources that have to be shared among the running tasks. In order to preserve the consistent state of these shared resources, they have to be accessed in a mutually exclusive way. The sequence of instructions performing the access is usually embedded in critical sections. A *critical section* is a sequence of instructions that must not be interrupted by other jobs if the interruption would lead to unpredictable or inconsistent data and system states. Since more than one task might want to use a certain resource at the same time, there has to be a policy in place that controls the access to the shared resources in a predetermined way.

Various protocols have been developed to serialise the access of competing real-time jobs to shared resources. These protocols are based on synchronization objects that provide a mechanism to guard concurrent access to the shared resources. In addition to access control, resource sharing protocols are also concerned with two common phenomena that may occur during the serialised access. One is, bounding the time of *priority inversions* [89] and the second is, to prevent *deadlocks.* Priority inversion occurs if the execution of a higher priority job is blocked by a lower priority job, and a deadlock emerges if two or more jobs wait for each other to release reserved shared resources before they can continue processing.

Sha et al. [89] introduced the *basic priority inheritance protocol (BPIP)* and the *priority ceiling protocol (PCP)* as a solution for the problems that might occur during concurrent access to shared resources. The main difference between the two protocols is that the PCP minimizes blocking time and prevents deadlocks without the need for total ordering of the access to guarded shared resources. However, PCP requires offline analysis of a given taskset to determine the proper priority ceiling value of each shared resource. But due to the offline analysis of the tasks and the used shared resources, the taskset schedulability can be ensured before runtime also in the presence of shared resources. In contrast, BPIP does not require offline analysis of the used shared resources since the priority of the executed jobs is adjusted during runtime according to the priority of jobs attempting to access shared resources guarded by synchronization objects.

Another resource sharing protocol is the *immediate priority ceiling protocol (IPCP)* [18]. It has its origins in the *stack resource policy (SRP)* [11]. In contrast to the SRP, IPCP was developed for fixed-priority scheduled system. Compared with the original PCP [89], the IPCP specifies that a job's dynamic priority during its execution is the maximum value of the corresponding task's static priority and the ceiling of any resource locked by that job at that time. Using the IPCP, jobs can be blocked only before their execution. It is guaranteed that once a job starts its execution, all resources that it needs are available until its completion. That means jobs might only be blocked at the beginning of their execution since the resource availability is ensured before they start execution.

Although jobs are blocked at the beginning of their execution if not all the required resources are available, the worst-case behaviour of the PCP and IPCP are identical [18]. Since blocking occurs only at the beginning of the job execution, the number of context switches is reduced in contrast to PCP.

The side effect of resource access policies, however, is that the response times (introduced in Section 2.5) of higher priority tasks might increase due to blocking caused by lower priority tasks.

2.2 Hierarchical and open real-time systems

The continuous development and improvement of microcontrollers and microprocessors provides engineers with steadily increasing processing power. The increasing processing power enables not just to increase the complexity of the executed jobs but also to integrate multiple concurrently executed applications on a single platform. The integration of multiple applications on a single processor platform can increase the efficiency of embedded systems because of e.g. reduced number of hardware components and reduced power consumption of the entire system, or reduced development effort due to reuse of existing applications.

However, the integration of multiple applications on the same platform, where each of them might be developed independently of the other applications, requires dedicated schedulability analysis. It has to be ensured that the temporal requirements of all real-time tasks can be met also in an integrated system where the processor time is shared not just among tasks but also among the applications.

The hierarchical system model as an extension to the classical real-time multitasking model provides the required means to analyse the schedulability of multi-application multitasking real-time systems.

The initial idea of a two-level hierarchical real-time system model was introduced by Deng et al. [36] in 1997 on the basis of an *earliest deadline first (EDF)* operating system level scheduler and priority-driven task scheduler at the application level for each application in the system. The constraints of this initial model, like application tasksets are limited to periodic tasks and that no global shared resources may be used, were removed in a further development [35] of the hierarchical model. However, the transition of the system level scheduler from EDF to a fixed-priority schema was implemented by Kuo and Li [53] in 1999, providing a hierarchical model that enables the assignment of processor time to applications and their associated real-time tasks by fixed-priority schedulers.

The concept of hierarchical scheduled real-time systems can be further extended by a dynamic property. In certain systems, the set of applications is either not known in advance or some of them need to be activated and deactivated during runtime in order to efficiently use the available processing resource. Hence, the operating system or a middleware sitting on top of the operating system has to enable during runtime the submission of new applications into a running system. These systems are referred to as *open systems*, since the set of active applications might change during the course of the system's lifetime. The composite model including both the concept of open real-time systems and the concept of hierarchical scheduling was also formalised by Kuo and Li [53].

The hierarchical and open real-time system model does not impose any limitation on the level of scheduled entities as the model (see Figure 2) defined by Saewong et al. [87] shows. Their model is able to define a hierarchy of schedulable entities up to an arbitrary depth. Despite the possibility to define arbitrarily deep scheduling hierarchies, Saewong et al. [87] also suggested that the depth of such systems never exceeds 3–4 levels. In the context of real-time systems we will associate one level with applications and a second level with real-time tasks. This view of the system allows us to limit our considerations to two-level hierarchical open system. Furthermore, our focus will be on applications consisting of periodic and sporadic real-time tasks that are largely applied in the automotive, aerospace and robotic domain.

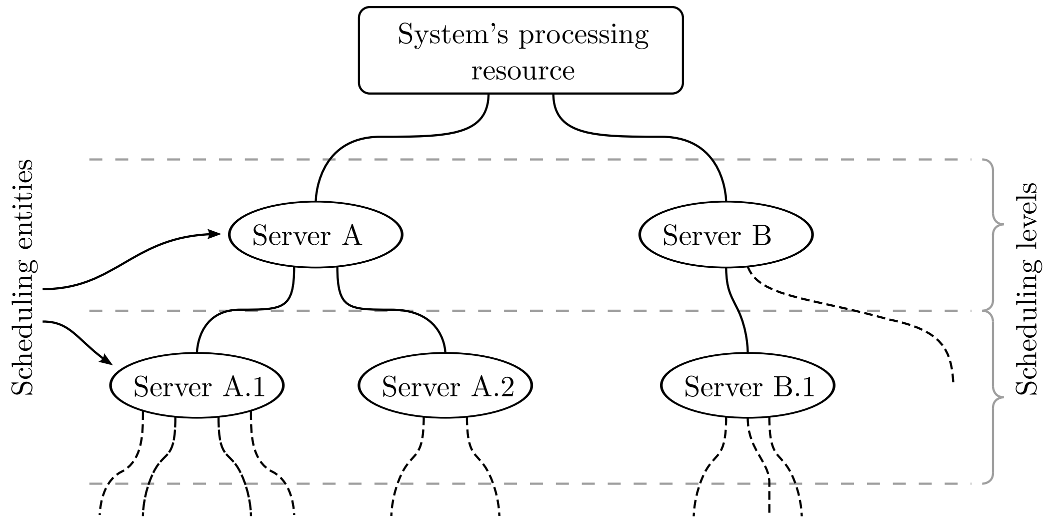

Figure 2: Specific example of the hierarchical model [87]

As it has been indicated in Section 2.1, some real-time tasks within an application usually need to access common resources. In order to solve the problem of concurrent access to these shared resources and to ensure temporal guarantees of real-time tasks, appropriate access policies were introduced. However, without any modification, the shared resource access policies presented in Section 2.1 are only applicable to tasksets of a single application. Therefore the analysis of hierarchically scheduled open real-time systems, where multiple applications are integrated on a single uniprocessor platform, requires additional considerations.

In the following we will distinguish between *local shared resources* that are used only by the taskset of a single application and *global shared resources* that may be used by multiple applications.

Concurrent access to local shared resources can be serialized according to one of the previously introduced resource sharing protocols and does not require any additional analysis for hierarchically scheduled real-time system.

Then again, the access to global shared resources needs a dedicated approach in order to keep the blocking time impact small on tasks in other applications.

One proposed approach is the use of budgeting mechanisms, like execution-time servers (further described in Section 2.3), to schedule the execution of critical sections accessing global shared resources. Supported by experiments in [34], the *multi-reserve Priority Ceiling Protocol* [34] was suggested as the best alternative among various server based approaches. The multi-reserve PCP requires a dedicated server for each global shared resource and the task accessing this resource. The execution capacity of the server is defined according to the expected length of the associated task's critical section. However, global shared resource protocols, relying on the support of dedicated servers, have the disadvantage of higher overhead due to server context switch every time a critical section is entered and left.

Another policy for global shared resource access in hierarchical fixed-priority preemptive real-time systems was introduced by Davis and Burns in [27]. The *Hierarchical Stack Resource Policy (HSRP)* extends the IPCP and specifies a policy for the serialization of concurrent access to global shared resources by utilising the overrun and payback mechanism [46] for servers. The *overrun and payback mechanism* enables the server to occasionally overrun its assigned capacity. If a server overruns its reserved capacity while one of its tasks has a lock on a global resource, the server continues to execute until the task releases the global resource. In the case of a server capacity overrun and the application of the overrun and payback mechanism, the server capacity is decreased by the amount of overrun at the beginning of the next server period.

Due to the availability of various competing global shared resource access policies, we consider shared resources in the context of the optimal server parameter selection problem to be part of the future work.

2.3 Temporal partitioning

The hierarchical open real-time system model provides the necessary foundation to analyse integrated real-time systems with multiple concurrently executed multitasking applications. In integrated real-time systems it is important to avoid temporal faults that might occur on the processor as a common computational resource for multiple applications.

The main cause of temporal fault is when one job delays the execution of other jobs and consequently extends the execution time of an application. *Temporal partitioning* restricts a temporal fault to its cause and avoids its propagation. The enforcement of temporal partitioning preserves the availability of the processor as predicted by the applied scheduling policy. Hence, the time instant and duration of processor availability is maintained and the predetermined temporal guarantees of correctly operating applications are preserved.

With regards to the implementation of temporal partitioning we distinguish between *static* and *mutable* methods [75].

Static temporal partitioning enables an easy implementation by defining before runtime the static time intervals that are assigned to unique applications for the execution of their tasks. The assignment of time intervals to applications represents a sequence of temporal partitions and this sequence is usually cyclically repeated. However, due to the rigid property of static temporal partitioning, the implementation of open real-time systems with a changing set of applications becomes more difficult with this method.

By contrast, the mutable temporal partitioning method facilitates the dynamic property of open real-time systems. In order to implement mutable temporal partitioning in real-time systems, usually *execution-time servers* are utilised. Execution-time servers provide the necessary means to enforce the temporal requirements of applications in a running system.

Servers represent a resource reservation mechanism and implement the processor time reservation for the running applications in a system. Each server defines by its execution capacity the fraction of processing time that is made available within a certain time interval to the associated application, for the execution of the application's real-time tasks. Although server concepts [68, 88, 96] have been introduced to improve the response time of aperiodic tasks, their application has been adapted to provide temporal partitioning among tasks [1].

At the operating system level we consider servers as resource reservation mechanisms that are scheduled by the operating system scheduler. The assignment of the processing time to servers and the associated applications in the system is determined according to a scheduling policy that is similar to scheduling policies for real-time tasks. When processing time is assigned to a server, its execution capacity is decreased at a uniform rate as the associated application's tasks are executed. A server is suspended until its next capacity replenishment in the case that the associated tasks consumed the server's initial execution capacity.

Various approaches were proposed in the literature to implement execution-time servers for real-time systems.

For single threaded applications Mercer et al. [73] and Rajkumar et al. [80] investigated the design and implementation of a server mechanism, denoted as *reserve* in their work. With the proposed framework [80], the responsibility of temporal parameter selection for servers is allocated to the associated applications instead of implementing it as part of the system functionality where information about the overall resource usage is known. The two publications [73, 80] also provided an extensive evaluation and investigated how certain server and system properties influence the behaviour of the executed applications. For example, the accounting mechanism for system calls that are executed outside the application's server or the achievable processor utilisation based on the policy for scheduling depleted servers. Our focus is however on multitasking applications requiring hierarchical scheduling.

For fixed-priority scheduled systems the periodic [88], sporadic [95, 96] or the deferrable [59, 97] server is the most common implementation scheme for temporal partitioning.

In contrast to other server types, *periodic or polling servers* require a low effort to implement temporal partitioning and in contrast to deferrable servers they cannot jeopardize the schedulability of the system due to back-to-back interference. For fixed-priority scheduled real-time systems Sha et al. [88] introduced an implementation of periodic servers based on periodic tasks with predefined worst-case execution times as the server capacity. The periodic task acts as an execution-time server by scheduling and executing the associated aperiodic tasks as long as there are aperiodic jobs ready to execute and the server's execution capacity is not completely consumed. As with periodic tasks, periodic servers also have a period. The server execution capacity is always replenished at the beginning of the next server period.

In a system with multiple servers, like an open real-time system, the processing time assignment to applications is accomplished by the usage of a fixed-priority scheduling scheme for the corresponding servers. Each server has a fixed unique priority that is used analogous to task priorities for server level scheduling decisions, i.e. which server's taskset is provided the processing resource next.

From the analysis point of view [96] a periodic server behaves like a periodic task. This enables the application of already existing analysis techniques for schedulability tests of a temporal partitioned fixed-priority scheduled real-time system.

The disadvantage of the initial periodic server implementation is that it looses its execution capacity if there are no aperiodic jobs to process. Using an alternative implementation for periodic servers it can be avoided that the server's execution capacity is instantaneously exhausted if there are no aperiodic jobs in the ready queue waiting for execution when the processor is assigned to the server, i.e. when the server is invoked. In that case the periodic server's execution capacity is continuously consumed by an idle task until it is exhausted [26]. This modification of the original periodic server model enables the execution of aperiodic jobs even if they arrive after the server invocation.

Depending on the period and execution capacity of the periodic server, aperiodic jobs might still arrive while the server is waiting for its capacity replenishment. In this case the responsiveness of aperiodic tasks can be considerably decreased. The *deferrable server* [59, 97] extends the properties of the periodic server in order to preserve the server execution capacity for aperiodic job arrivals after the server invocation. The deferrable server preserves the execution capacity until its next period where it is replenished to its maximal value. Aperiodic jobs can be served by the deferrable server as long as its priority is the highest amongst the ready servers in the system and as long as its capacity is not exhausted. Like with the periodic server, the deferrable server's execution capacity is replenished at the beginning of its next replenishment period.

However, since the deferrable server's execution capacity is preserved during the entire server period, lower priority servers can suffer *back-to-back* interference. This situation occurs if the server's entire execution capacity is consumed right before its next replenishment period and the consumption of the replenished execution capacity continues right after the replenishment period. The deferrable server then causes on lower priority servers a higher interference than initially assumed by the schedulability analysis for periodic tasks.

Considering the back-to-back hit of the deferrable server, Saewong et al. [87] presented a dedicated schedulability test for tasks. Since the analysis captures the worst-case situation of a deferrable server's execution capacity consumption and its impact on lower priority servers, the analysis reflects a very pessimistic situation for the majority of cases.

Sporadic servers [95, 96] also belong to the category of execution capacity preserving servers. In contrast to other servers that periodically replenish their capacity, the sporadic server's capacity is only replenished if it has been consumed by aperiodic or sporadic jobs. The magnitude of capacity replenishment is equal to the capacity that has been consumed during the last server invocation. In contrast to the deferrable server, the processor load generated by a sporadic server is equal [95, 96] to the load that is produced by a periodic task with the same period and worst-case execution time. Therefore in the schedulability analysis of a real-time system containing sporadic servers, sporadic servers can be handled as periodic tasks.

Although the sporadic server has some advantage from the schedulability analysis point of view and a similar performance as the deferrable server, the implementation complexity is not negligible since the system has to manage all the replenishment times.

In this thesis, therefore, we will focus only on the periodic server model. The general periodic execution-time server S_x is characterised by its *execution capacity* Θ_x and *replenishment period* Π_x.

The server's execution capacity Θ_x denotes the maximal processing time that can be consumed by the associated tasks before the server is suspended. The execution capacity is replenished to its full amount after every server period Π_x.

In a multi-application real-time system with more than one server, the processor time assignment to servers is scheduled according to a fixed-priority scheduling scheme. Hence, the priority P_x of a server S_x is determined according to a fixed-priority assignment policy. Due to the similarities between periodic servers and periodic tasks, we utilise the rate monotonic priority ordering scheme [67] to determine the server priorities.

In the schedulability test of periodic servers, the term *schedulable* describes that a server is able to supply its entire execution capacity to the associated application's tasks before the server's next capacity replenishment.

The attributes of this server model will be extended later in this chapter in order to support the solution of certain temporal partitioning problems that are addressed by the undertaken research.

In the envisaged two level hierarchical real-time system, the real-time tasks of each application are mapped to and are executed by one or more servers, in order to enforce temporal partitioning. In the former case the application's taskset is mapped to one server, where in the latter case the taskset is divided into subsets and each subset is mapped to a server. Unless a one-to-one mapping of tasks to servers is used, an application level scheduler determines the execution order of tasks on the same server.

Furthermore, to ensure the temporal protection among the various applications, the tasks of one application must not be mixed with tasks of other applications on the same server.

2.4 Flexible real-time systems

Task specifications of modern real-time systems increasingly exhibit flexible behaviour that is hard to describe only with the classical hierarchical real-time system model. The analysis [47] carried out within the FRESCOR project highlighted the need for additional temporal attributes in order to capture the temporal requirements of modern real-time systems. Selected fundamental extensions proposed [47] within the FRESCOR project are also utilised by the research presented in this thesis.

The analysis of modern real-time systems revealed the requirements [44, 47] that state-of-the-art real-time systems need to address. This thesis builds on the fundamental requirements [44, 47] that were identified by the FRESCOR project for a scheduling framework in order to facilitate the development of current and future real-time systems. The following list represents the selected high level FRESCOR requirements that will be addressed in the presented research work:

- support the composition of independently developed applications, and to control and to enforce the predetermined resource usage of the applications;
- support the handling of periodic and sporadic events with variable periodicity and resource requirements. The change of periodicity and

resource requirement can be triggered either by changes of system internal states or events in the environment of the system;

- support new attributes, denoted as *importance* and *weight*, as a means to influence the exposed *Quality of Service (QoS)* of applications after they are integrated on a uniprocessor platform;
- support the system wide optimization goal to maximize the processing resource usage;
- support *contract* based acceptance testing at runtime for new applications.

These requirements guide on the one hand the realization of improved analytical tools and on the other hand the implementation of these tools in order to provide efficient temporal partitioning on uniprocessor systems.

2.4.1 Resource dependent Quality of Service

Flexible applications come in many different forms. For instance, multimedia systems need to process different kinds of video and audio streams that have highly variable computation times but require real-time processing and rendering. It is common that classic industrial control applications, such as a robot control, get mixed together with multimedia activities when the process in which the robot is working requires image capture and analysis, or remote video monitoring.

Not all the applications running in a real-time system are capable of adapting or adapting equally to the available processing resource [47]. Of those that may adapt to the available processing resource the level of adaptation may be different. For instance, a video player may upgrade itself to a higher frame rate if more processing time is available for the corresponding application. The application then changes the rate of the rendering task in a continuous way, up to a maximum level after which there is no perceived increase in the quality of the output. These type of applications are referred to as *continuous*. A control algorithm on the other hand may have two versions: a fast one with a low quality output and one requiring more computation time and providing a high quality output. In this case the system should allocate resources to run either one version or the other. Applications with this type of behaviour are referred to as *discrete*. In general we find application tasks that can operate and generate valid results at different frequencies (i.e. having a variable period), and/or handle different processing time assignments (i.e. having a variable execution time). Hence, the flexible real-time system model provides a further complementary extension to the classical, hierarchical and open real-time system model.

The output generated by flexible real-time systems is usually also associated with certain *Quality of Service (QoS)* values. However, the notion of QoS and QoS attributes is very domain specific. Even in a certain domain, like in our case the domain of real-time systems, there is a wide difference in the QoS definition for results generated by tasks or systems [3, 50, 54, 63, 76, 81]. For example, QoS is considered in [81] as the sum of the application utilities in the system where the application utility is a function of the allocated processing resource. Others [54] use the present task period or sampling rate as a direct indicator for the QoS where a shorter period is equivalent to a higher quality of service. With another approach [63] each application specifies a finite set of performance levels at which it can operate, the corresponding resource requirements and how much benefit it provides to the system at that level. The benefit provided by the applications is subsumed as the total system utility and the QoS manager aims at maximising this value by selecting the appropriate performance level for each application. An approach [76], introduced in the context of distributed and open real-time systems, describes the QoS output of tasks as a function of the reserved computation time. For soft real-time tasks, a specific function calculating a QoS value based on the completion ratio of tasks [50] was defined, i.e. by considering the number of tasks that completed on time, missed their deadline or were completely abandoned due to insufficient amount of processing resource.

Due to the lack of unity in the literature about the QoS attributes and the exposed QoS of real-time tasks and systems, we simply consider that the QoS of tasks is monotonically non-decreasing with decreasing period and/or with increasing amount of processing time provided to the tasks.

2.4.2 Flexible tasks

In the past, a few aspects of flexible real-time systems were addressed in the literature by utilising the adaptability of flexible tasks to the available processing resources. In order to exploit the adaptability of flexible tasks, various resource allocation mechanisms were proposed. Former research work is analysed in the following, which serves as a motivation for the proposal of a more general tasking model that is able to capture the temporal properties of a wide variate of flexible tasks.

An attempt to address various system load conditions at the tasking model level was undertaken by Buttazzo et al. [21, 22]. In the domain of multimedia or adaptive control systems, certain algorithms can handle different task period settings and allow the system to adjust these settings according to the present system load. The proposed elastic tasking model enabled the adjustment of task periods in order to adapt the processing to different load conditions. The category of tasks addressed by the work in [21, 22] represent a subset of tasks (i.e. providing variable task period) that will be further examined in this work.

A QoS based resource allocation model addressing the variability of either the computation time or the periodicity of applications in real-time systems was investigated by Rajkumar et al. [81]. The presented approach, however, requires a significant application involvement in the determination process of feasible server parameters. Resource allocation to applications is made in terms of resource utilisation, but it is the application's responsibility to choose the appropriate parameters for the temporal partitioning, i.e. the replenishment period and execution capacity of the associated server. Moreover, the algorithm that determines the resource allocation, requires the implementation of QoS functions that represent resource dependent changes of the application's contribution to the system utility. Unfortunately the definition of such QoS functions is usually not easy and straightforward. Furthermore, the involvement of the applications in temporal parameter determination blurs the responsibilities of applications and resource allocation and management mechanisms.

In contrast to the approach of Rajkumar et al. [81] with blurred functional boundaries between applications and resource management, Rosu et al. [86] suggested an adaptive resource allocation mechanism for distributed real-time systems with a strong separation between application and resource adaptation functions. The expected resource needs of applications are specified by configurations. The choice of the appropriate configuration and the resulting resource allocations depends on environmental states, availability of resources in the system and the achievable system performance. The resource allocation is carried out as a response to events in the environment and changes in the processing demand of a complex distributed application.

For soft real-time tasks, a resource adaptation mechanism using a heuristic algorithm to increase the overall benefit by iteratively adjusting the QoS level of the adaptive soft real-time tasks was examined by Lin et al. [62]. As the resource demand varies with the QoS levels, the processing of the adaptive tasks is adjusted so that they can be accommodated on the available processing resource. The processing time reservation is accomplished by a periodic budgeting mechanism similar to periodic servers. The system reacts to various load conditions by adjusting the period in which a fixed execution capacity is made available to a soft real-time task.

Real-time systems operating in a predefined set of modes with each mode of operation corresponding to a QoS level of the system, Almeida et al. [5] presented an approach to adapt the temporal parameters of flexible periodic tasks during runtime. Each task's execution time and period is expressed by a finite set of n-tuple vectors, each corresponding to one of the n different QoS levels. From the set of all possible combinations of task parameters, a set of schedulable configurations is deduced by an offline method. This set is used by the online QoS manager to adapt the task parameters according to changes of the system load.

Considering the various types of flexible tasks with either discrete or continuous ranges of temporal parameters, the general flexible real-time tasking model capturing all the aforementioned task properties can be defined as follows.

The temporal specifications of real-time tasks implementing for example N-version or imprecise computation algorithms can be best described by *discrete tasks* [47]. The different implementations of these tasks, either by providing multiple versions or state dependent execution paths, is accompanied by different temporal specifications. Each implementation version would be associated with a different task period, deadline and worst-case execution time value. Formally, the finite set of temporal parameter values of a *discrete* periodic task τ_i with m different execution behaviours can be described [33] as $\left(C_{i(v)}, T_{i(v)}, D_{i(v)}\right) \in \left\{\left(C_{i(1)}, T_{i(1)}, D_{i(1)}\right), \cdots, \left(C_{i(m)}, T_{i(m)}, D_{i(m)}\right)\right\}$. The definition of the discrete task temporal parameters, expressed by a tuple $\left(C_{i(v)}, T_{i(v)}, D_{i(v)}\right)$, represent the temporal properties of the v-th version of a task τ_i and implies a link between the task's period, deadline and assumed worst-case execution time.

Real-time tasks that implement, for example, anytime algorithms belong to the second group of flexible tasks also addressed in this thesis. Tasks in this category are denoted as *continuous tasks* [47]. If anytime algorithms are executed at a higher rate or they are given longer time for execution, the quality of the results they generate improves. The implementation of anytime algorithms allows to utilise any task parameter value selected from within a predefined interval for each temporal parameter. That means, the actual task period T_i is selected from a range of possible values with integer granularity defined by the interval $\left[T_{i(\min)}, T_{i(\max)}\right]$. The same applies to the task's worst-case execution time C_i that is in the range $\left[C_{i(\min)}, C_{i(\max)}\right]$. The task deadline D_i is constrained by the task period T_i. Since the addressed system model does not permit multiple task releases before its deadline, it leads to the implicit definition of $D_i \leqslant T_i$. In contrast to a discrete task, the period, the deadline and the allocated execution capacity of a continuous task can be adjusted independent of each other.

2.4.3 Flexible execution-time servers

The presented adaptability of flexible tasks to system load, state and processing resource availability enables a more sophisticated temporal partitioning on uniprocessor systems, including a higher flexibility than the usual static partitioning approach would provide. The flexibility of the presented tasking model can be utilised in order to determine various server parameters such that the processing resource provided by the server under different circumstances ensures the schedulability of the corresponding flexible taskset. That means, by using different server parameter values, different amount of processing resources are reserved for an application. Depending on the processing resource that is reserved for an application, the quality of the results generated by the associated flexible tasks varies. In this thesis the interpretation of this correlation is that the various server parameters and the corresponding resource reservations reflect various application QoS levels.

An execution-time server, that is able to realise different temporal partitions for an application with flexible real-time tasks, will be denoted as a *flexible* *server*. Analogous to flexible tasks, we also classify flexible servers according to the domain of their temporal parameters either as continuous or discrete [47]:

- For a continuous server S_x, the operational ranges of period and execution capacity is defined by a lower and upper bound, $[\Pi_{x(\min)}, \Pi_{x(\max)}]$ and $[\Theta_{x(\min)}, \Theta_{x(\max)}]$, respectively. The actual value assigned to a server's temporal parameter can take any value from within its corresponding operational range. The execution capacity and period of continuous servers are, like the parameters of flexible tasks, independent of each other and therefore can be adjusted independently.
- For a discrete server S_x a finite set of tuples, $\{(\Theta_1, \Pi_1), \cdots, (\Theta_v, \Pi_v)\}$, is defined. Only values from this set of temporal parameter tuples can be assigned to the temporal parameters of a discrete server. This definition implies that defined temporal parameter values within a tuple are linked to each other.

At the moment, the association of flexible tasks to flexible servers is constrained such that discrete servers contain only discrete tasks, and continuous servers contain only continuous tasks. The mixture of discrete and continuous tasks is considered as an interesting topic for future research work.

2.4.4 Contract model

The server parameter values for a given application taskset represent implementation specific values, like the period and execution capacity of periodic servers. However, in open real-time systems, where applications are launched and terminated during runtime, it is preferable to define the application's resource requirements at a higher abstraction level in order to facilitate the independence from system specific implementation details. Hence, an application's resource requirements defined in a contract would remain the same regardless of the specific server implementation (fixed or dynamic priority scheduled servers).

The FRESCOR project, as the origin of the research work presented in the following chapters, defined the contract model [48] that enables the description of application resource requirements in an implementation independent way. The resource requirements are defined in a contract for each application and they are expressed by timing values that specify the required processing time capacity per time interval. With a focus on flexible application tasks, the resource requirements for an application, specified in a *contract*, address the application's minimum and maximum resource requirements.

First, each application specifies in the contract the minimum required processing time and the corresponding maximum time interval (in the following referred to as period) in which the processing time is provided to the application. As far as the application is capable of utilising higher processing resources than specified by the minimum resource requirements the contract also defines the maximum utilisable processing time and the minimum period. Two additional contract attributes that will be addressed and explained in a later chapter, denoted as importance and weight, allow during runtime to influence to a certain extant the distribution of spare processing capacity in the system among the running flexible applications.

When an application is launched, the contract is negotiated with the resource managing middleware in the system. The negotiation process ensures that the minimum resource requirements of the application can be satisfied while the already running applications in the system remain schedulable and their contracts are not violated. If the contract negotiation succeeds, i.e. there are sufficient processing resources to satisfy at least the application's minimum resource requirements, an appropriate server is created to manage the application taskset's resource consumption. This step is denoted as the *online acceptance test* for applications in an open real-time system.

As an extension to the initial contract negotiation process, the resource managing middleware will attempt to distribute any spare capacity of the processing resource among the flexible applications. This task needs to be accomplished during runtime when the running applications in the system and the current system load is known. The spare capacity in the system will be distributed by utilising the information in the contract about the maximum resource requirements of each application, and the importance and weight attributes.

An efficient online spare capacity distribution algorithm including the acceptance test for applications will be examined in detail and the results will be presented in Section 5 and Chapter 6.

2.4.5 Application design blueprint

The implementation and design of flexible applications and tasks goes beyond the classical real-time task design. This section presents a design proposal for flexible tasks under consideration of the envisaged middleware functionality [48] as defined in the FRESCOR project. It is assumed that in addition to the functions provided by a standard real-time operating system, the system functions that wait for certain events or modify task parameters are implemented in a resource managing middleware sitting on top of the real-time operating system.

The general application structure that we propose consists of a high priority main task that is responsible for application specific management functions, and one or more real-time task.

Slightly different implementations are proposed for continuous and discrete tasksets due to their intended fields of operation. Discrete tasks usually provide solutions for problems that can be decomposed into a finite number of operational modes. While the set of continuous tasks facilitate implementation of solutions that support an almost seamless improvement or degradation of the provided application services.

Algorithms 1 and 2 illustrate an exemplary implementation of the application's main task (utilising an OS or middleware API as described in [48]) and further mode based periodic tasks within the same application. For discrete applications the primary purpose of the main task (Algorithm 1) is to detect and define the application's current mode of operation based on the processing resource reserved for it, and to create and start the tasks for the active mode of operation.

In the case of the server parameter change, a discrete application waits for an *idle instant* before it suspends all the child tasks. The idle instant and the mode change protocol based on this concept will be further explained in Section 2.6. After all tasks that are associated with the application have been suspended, and the new server parameters are put into effect, the tasks in the new application's mode of operation are resumed and scheduled according to the application specific fixed-priority scheduler.

When the discrete periodic tasks (see Algorithm 2) are executed, they query the application's mode of operation and carry out predefined work according to the currently active application's mode. After a task completes the job for the

```
1 while no application shutdown do
2     Determine application's mode of operation based on assigned server
      parameters;
3     Set application's mode of operation;
4     Create and start application tasks using task parameters that are
      designated for the current application's mode of operation;
5     Suspend main task and wait for intended server parameter change
      notification or application shutdown event;
6     Wait for application's taskset idle instant and suspend all associated
      tasks;
7     Signal application readiness for server parameter change and wait for
      the system to activate new server parameters;
8 end
```

Algorithm 1: Discrete application's main task

```
1  while true do
2      switch associated application's mode of operation do
3          case mode A
4              Execute work appropriate for mode A;
5          end
6          case mode B
7              Execute work appropriate for mode B;
8          end
9      end
10     Wait for next periodic task activation;
11 end
```

Algorithm 2: Discrete periodic task [24]

active application's mode, it suspends itself until the next periodic activation and allows lower-priority tasks in the same application to execute.

A common usage of discrete applications is to solve various control problems (e.g. cruise, speed, attitude or process control). In the domain of control applications, the system is usually in a certain state that represents a mode of operation and enables an easy mapping to discrete tasks.

```
1 while no application shutdown do
2     Adjust execution capacity and period of continuous tasks according
      prevalent server parameters;
3     Create and start application tasks using task parameters that were
      determined for the current server parameter setting;
4     Suspend main task and wait for intended server parameter change
      notification or application shutdown event;
5     Wait for application's taskset idle instant and suspend all associated
      tasks;
6     Signal application readiness for server parameter change and wait for
      the system to activate new server parameters;
7 end
```

Algorithm 3: Continuous application's main task

```
1 while true do
2     while task's execution capacity sufficient for an additional iteration
      do
3         Improve end result;
4         Update globally accessible result;
5     end
6     Wait for next periodic task activation;
7 end
```

Algorithm 4: Continuous periodic task [104]

A different approach to realise mode dependent task execution is to provide mode specific task implementations, i.e. one task implementation for each mode of operation.

As indicated, there is a minor difference in the implementation of continuous and discrete applications and their tasks. The main task of a continuous application (Algorithm 3, also utilising an API as described in [48]) has a similar structure to the main task of a discrete application. However, instead of selecting and setting mode specific task parameters values, in the case of continuous applications, the task parameters like period and granted execution time are adjusted within defined ranges.

Nevertheless, continuous applications also use the idle instant mode change protocol to modify the associated flexible real-time task parameters.

In contrast to discrete tasks, continuous tasks (Algorithm 4) usually implement anytime algorithms that can continuously refine the end result (as long as they do not reach the exact solution) and provide improved results the longer they run.

A common application for continuous tasks is in the area of robotics. In this domain, algorithms implementing, for example, path planning, or video processing can provide better results the longer the corresponding task can execute.

In summary, continuous and discrete tasks allow developers to cover a wide variety of real-time application types and provide a solution for problems that are less rigid than those encountered in classical real-time systems.

2.5 Schedulability analysis

Real-time systems have to comply with temporal requirements that are specified before a system is deployed. A central component of the real-time theory is the offline analysis of envisaged real-time systems. That means, given a set of temporal parameters for a taskset, the schedulability analysis determines whether the predefined temporal requirements can be met or not.

Schedulability analysis, however, is also applied on more complex real-time systems, like hierarchically scheduled systems with multiple applications. In the case of multi-application multitasking hierarchical real-time systems, the purpose of the schedulability analysis is twofold. Additionally to the verification that the temporal requirement of each task can be met, the processing time supply to each application via its associated execution-time server is analysed.

The schedulability techniques applied on execution-time servers provide the means to verify that the server's execution capacity can be completely supplied to the associated application's taskset before the end of the server's replenishment period is reached. Hence, concerning the schedulability analysis of servers, the term *schedulable* defines if a server can supply its complete execution capacity to the associated application taskset before the occurrence of the server's next replenishment event.

A task is denoted as *schedulable* if the *schedulability analysis* determines beforehand that a task's temporal requirements can be ensured during runtime. For fixed-priority scheduled periodic and sporadic tasks there are three common approaches [8, 58, 67] to determine the schedulability.

The considered schedulability analysis techniques assume that task priorities are assigned according to a fixed-priority scheme, like the rate or deadline monotonic priority assignment. These schemas assign higher priority to tasks with shorter period or shorter deadline according to the rate or deadline monotonic priority assignment [61, 67].

Based on the accuracy of the schedulability test results for fixed-priority scheduled real-time system, a test can be specified as sufficient or exact. Although sufficient schedulability tests are efficient and usually have a time-complexity of $O(n)$ with n denoting the number of tasks in the given taskset, the test may provide false negative responses. That means, a taskset might be schedulable but using the utilisation based test, it is determined as not schedulable.

A sufficient schedulability test proposed by Liu and Layland [67] determines a taskset's schedulability based on the corresponding taskset utilisation. Since the determined utilisation bound (Equation 2.1) represents an upper bound for a schedulable taskset utilisation, it can be only used as a sufficient test.

$$\sum_{i=1}^{n} \frac{C_i}{Ti} \leq n(2^{1/n} - 1) \tag{2.1}$$

Equation 2.1 indicates that the higher the number of tasks in the systems the lower the upper bound of the processor utilization will be where a feasible schedule can be guaranteed. The upper bound decreases as the number of tasks n increases, and it converges towards 0.69 [67] representing a maximum processor utilization of 69%.

Although there were improvements [23, 54] to the initial Liu and Layland [67] based utilisation test, the accuracy of exact tests has not been reached yet.

Contrary to pessimistic estimation of sufficient schedulability tests, exact schedulability tests always determine accurately whether a taskset is schedulable or not.

Exact schedulability analysis approaches [8, 58] assume the worst-case situation. The worst-case situation occurs when a task has its *worst-case response time* [67], i.e. the longest time from the release of a task until its completion. This situation occurs when the task is released at the same time instant as all the higher priority tasks and it is denoted as the *critical instant*.

The demand based exact schedulability analysis introduced by Lehoczky et al. [58] for rate monotonic scheduled tasksets examines the taskset demand over all priority levels at specific time instants.

It is sufficient to verify that the cumulative demand of a taskset at a number of finite points [58] is less than or equal to the available processing capacity. For priority level i only the time instants defined by the set E_i have to be examined. The set E_i defines the relevant time instants (Equation 2.2) that are multiples of all higher priority task periods, between the release time and deadline of task τ_i. These time instants are also referred to as *scheduling points* of the given taskset.

$$E_i = \{lT_k \mid k = 1, \ldots, i;\ l = 1, \ldots, \lfloor T_i/T_k \rfloor\} \tag{2.2}$$

Given the set of scheduling points E_i, the processor demand W_i at priority level i can be determined by Equation 2.3. The processor demand is caused by tasks at and higher priority than i.

$$W_i(t) = \sum_{j=1}^{i} \left(C_j \left\lceil \frac{t}{T_j} \right\rceil \right) \tag{2.3}$$

Based on the processor demand W_i for priority level i, the schedulability of a rate or deadline monotonic ordered taskset can be determined by Equation 2.4 [58]. The equation determines the smallest processor demand at each priority level and examines whether the maximum demand over all priority levels is smaller than the available processing time. Hence, for a schedulable taskset there has to be a time instant in E_i for all priority levels such that the processor demand of the taskset is smaller than the time elapsed by that time instant.

$$\forall i : \max_{1 \leq i \leq n} \left(\min_{t \in E_i} W_i(t) \right) \leqslant t \tag{2.4}$$

An alternative exact schedulability test is based on the examination of task response times. The response time based schedulability test verifies that a task's worst-case response time is less than or equal to its deadline, $\omega_i \leqslant D_i$. The worst-case response or completion time ω_i of a task τ_i is defined as the end of the level-i busy period. The *level-i busy period* denotes the time interval $(t_0, t]$ during which the processor is busy all the time executing jobs with priority i or higher [56, 68].

The worst-case response time of a task τ_i can be also considered as a *computational window* [8] that starts with the release of task τ_i and ends when the task completes its execution. The length of the computational window or worst-case response time (see Equation 2.5) is composed [8, 51] of the task's worst-case execution time C_i and a certain amount of interference I_i caused by higher priority tasks during the execution of task τ_i.

$$\omega_i = C_i + I_i \tag{2.5}$$

The higher priority interference I_i can be determined by analysing the number of higher priority task releases within the computational window of task τ_i. The maximum number of releases of a higher priority task τ_j within the computation window ω_i is calculated as $\lceil \omega_i / T_j \rceil$. Consequently, the demand of this task is equal to $\lceil \omega_i / T_j \rceil C_j$. Ultimately the higher priority interference is the sum of all higher priority task demands. The set of tasks with priority higher than i is denoted as $hp(i)$. Applying the substitution for the higher priority interference I_i in Equation 2.5, leads to a recurrence relation (Equation 2.6) that can iteratively determine the worst-case response time of a task τ_i.

$$\omega_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{\omega_i}{T_j} \right\rceil C_j \tag{2.6}$$

Equation 2.6 can be iteratively solved by applying $\omega_i^0 = 0$ for the first iteration. Successive values ω_i^{n+1} are calculated by using the previous value ω_i^n until the recurrence relation (see Equation 2.7) converges to a limit that is denoted as the task's worst-case response time. Alternatively, the iteration also stops, if the intermediate value of worst-case response time exceeds the task's deadline, i.e. $\omega_i > D_i$, in which case the task is determined to be not schedulable.

$$\omega_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{\omega_i^n}{T_j} \right\rceil C_j \tag{2.7}$$

The initial form of the recurrence relation, calculating a task's response time (see Equation 2.7), was extended by Audsley [9] in order to account for blocking. The time interval, while the execution of task i is blocked by an other task with priority lower than i due to concurrent access to a shared resource, is denoted as B_i (see also Section 2.1 for shared resource access policies). Hence, in the worst-case, the response time of task i is increased by the duration of the blocking time B_i. The adapted equation for the response time calculation, considering the blocking time B_i, is defined as depicted in Equation 2.8.

$$\omega_i^{n+1} = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{\omega_i^n}{T_j} \right\rceil C_j \tag{2.8}$$

In order to determine a task's response time, it is not required to start the recurrence relation with the initial value $\omega_i^0 = 0$. Though, it needs to be ensured that the initial value is smaller than the actual task's response time, i.e. representing a lower bound on the task's response time. Otherwise, if the initial value is too large, the recurrence relation might converge towards a value that is larger than the actual response time of the examined task.

Sjödin and Hansson [94] proposed a safe initial value that speeds up the convergence of the recurrence relation for the response time calculation. They reasoned that the worst-case response time of a task τ_i is at least as long as the response time of the next highest priority task ω_{i-1} plus the worst-case execution time C_i of the task τ_i. Hence, they proposed to set the initial value to $\omega_i^0 = \omega_{i-1} + C_i$. This improvement implies that the response time of each task in a taskset is determined in a decreasing priority order. By using an initial value for ω_i^0 that is larger than 0, some values are skipped at the beginning of the iteration. Consequently this results in a faster convergence towards the definite worst-case response time value.

An alternative calculation of a response time initial value [17] is based on the task's worst-case execution time and the utilisation of higher priority tasks, i.e. $\omega_i^0 = C_i / \left(1 - \sum_{\forall j \in hp(i)} U_j\right)$. Hence, this approach does not rely on the evaluation of task response times in a decreasing priority order.

The efficiency of various initial values and further improvements of a task's worst-case response time calculation were considered in [29, 32]. The improvements include the calculation of an upper bound [29] on the worst-case response time and new initial values [32]. The combination of a necessary schedulability test based on the response-time upper bound and improved initial values provide highly efficient response-time based schedulability tests. The efficiency of the proposed schedulability test [29, 32] and the applicability for online schedulability analysis forms part of the results presented in this thesis and will be further examined in Section 5.

Additionally to the schedulability test of tasks, the response time analysis can be also used to determine if a server can supply its execution capacity before the end of its replenishment period. As already noted, from the schedulability point of view, periodic servers behave like periodic tasks [96], enabling the application of schedulability tests that were initially developed for tasksets also to servers. In order to use the response time analysis for servers, in Equation 2.7, C_i has to be replaced by the server's execution capacity Θ_i and T_i with the server's replenishment period Π_i. The recurrence relation determines under consideration of higher priority servers in the system whether the server's execution capacity can be fully supplied to the associated taskset or not. This implies that if the value determined by the recurrence relation is smaller than the server's replenishment period then it is ensured that the taskset associated with the server will receive the predefined and guaranteed processing time.

The verification that the server can supply its full capacity can also be considered as part of the online admission test in open real-time systems. If the online admission test determines that upon the arrival of a new application in an open real-time system the associated server is not able to supply its full capacity to the application taskset, then the new application cannot be accepted by the system. In other words, it is not possible to reserve via an execution-time server sufficient processing time in order to satisfy all temporal requirements of the corresponding application's taskset. Hence, the new application is rejected in order not to jeopardise already running applications, and to possibly violate the temporal requirements of the new application if it would have been accepted and executed.

In summary, the introduced schedulability analysis techniques will be applied in this thesis for various purposes. The demand based analysis will be used by the server parameter selection algorithm to determine the taskset demand at relevant time instances. The response time analysis will be utilised in conjunction with appropriate initial values for the recurrence relation by the online admission test and the algorithm to determine the processor's spare capacity distribution among active execution-time servers.

2.6 Mode change protocols

The admission test in a flexible open real-time system, determining whether a new application can be submitted for execution into the system or not, represents only the first stage of the online system reconfiguration. An online reconfiguration takes place either when a new application can be submitted into the system or an application intends to leave the system. In each case, the server parameter values of applications that continue their execution, have to be adjusted and the parameter values of the new application servers have to be determined. However, the server parameters cannot be immediately changed when newly calculated server parameter values are available. The altered processing resource allocation to all active applications in the system by setting the server parameters to appropriate values must only happen at a certain time instant. Contrary to an immediate change, by applying a certain protocol to modify the server parameters in a timely controlled way it can be ensured that the system's temporal specifications are not violated during such a transitional state.

Multitasking but single application systems operating in different modes also require a deterministic protocol to change the task parameters when the system changes its mode of operation. *Mode change protocols (MCPs)* were developed to enable, during runtime, a controlled and deterministic transition from one set of tasks to an other one [90].

In some real-time systems only a subset of the available tasks is required to be active during certain periods of time. In multi-mode system we denote the tasks that are eligible to be scheduled in the current mode of operation as *active tasks*.

In a multi-mode system many task attributes vary, especially task's worst-case execution time, depending on system states [84]. A common way to take this kind of variability into account is to define various modes of operations, in which the system exposes different behaviours. Furthermore, events triggered by the system environment or sporadic events might require a more complex processing that can only be handled efficiently by using modes of operations with the associated active tasksets.

Additionally to the change of the taskset content, i.e. terminating certain tasks and staring new ones, the mode change may include or contain only the modification of task parameters that are active in multiple modes. Hence a mode change may be used to merely change the task parameters.

The tasking model enabling different modes of operation contains the definition of the set of active tasks with their corresponding attributes for each mode of operation.

Since the load during the transition from one mode of operation to a different one might be a lot higher than during the steady state of each mode, the deterministic transition from one mode to another one is controlled by mode change protocols [42, 43, 78, 84, 90, 99]. One important requirement of multi-mode real-time systems is to preserve the temporal requirements of the system during the transitional states between two distinct modes of operation.

For cyclic executive real-time systems Fohler [42, 43] defined a mode change protocol that uses precedence graphs to describe the set of system modes and their possible transitions. The nodes of the precedence graph describe the schedule for the corresponding mode of operation. The mode transition itself is also considered as a self-contained mode. However, in contrast to the schedule of regular system modes of operation, where the predefined schedule is executed periodically, mode transition schedules are carried out only once. The advantage of the proposed protocol in contrast to other MCPs [78, 84, 90, 99] is that new mode changes can be initiated while the system is in a mode transition.

Mode changes in cyclic executive systems, however, can last as long as the major cycle of the schedule. Fixed-priority real-time systems have a higher degree of flexibility than cyclic executive systems and usually provide better means to perform mode transitions in a more efficient way.

All mode change protocols introduced for fixed-priority scheduled real-time systems constrain mode changes to steady states of the system. That means, the mode change protocols for these systems do not allow a new mode to start while there is a mode transition in progress.

With respect to the multi-mode system behaviour there are three different states distinguished [78]. Before the system receives a *mode change request (MCR)* event, it is in the *old mode* steady state and executes *old mode* tasks. Upon the arrival of a MCR event, the system enters the *mode transition* state where old mode tasks are terminated according to predefined rules and *new mode* tasks are released. It is considered that the system enters its *new mode* steady state when after the MCR event all old mode tasks completed their last job and new mode tasks completed their first execution.

Furthermore, depending on the schedule of old and new mode tasks during a mode transition, mode change protocols can be classified [84] as:

- *synchronous protocols* that separate the execution of old and new mode tasks. New mode tasks are released only after all old mode tasks have completed their execution and,
- *asynchronous protocols* that enable the old and new mode task to execute concurrently during the transitional state.

Although asynchronous protocols can generally accomplish the mode transition faster then synchronous protocols, they have a higher complexity especially if shared resources are used in the system.

A simple asynchronous mode change protocol for fixed-priority preemptive scheduling was first introduced by Sha et al. [90]. The protocol uses the taskset utilisation to determine a feasible mode change. This mode change protocol, however, is based on a theorem that only provides a necessary condition for the feasibility test of the modified taskset. That means, certain tasksets might not be schedulable in the presence of mode changes although they were classified as such by the feasibility test of the simple mode change protocol. Tindell et al. [99]

provided a counter example and presented a new solution for a mode change protocol in fixed-priority scheduled real-time systems. Instead of using the notion of processor utilisation to derive the feasibility test for mode changes, Tindell et al. [99] used task categories (i.e. old mode, new mode or wholly new tasks) to predict their processing demand during a mode transition and incorporated this information into the response time analysis to determine a feasible mode change.

A mode change protocol proposed by Pedro and Burns [78] extends the capabilities of the protocol developed by Tindell et al. [99]. It allows the release of new tasks during mode transition to be delayed. Consequently, if the old and new mode of operation are both schedulable in their steady state, then a sufficiently large delay of new task releases during mode transition can also create a feasible transition between the two modes. That means, the release of new mode tasks is delayed so that they do not overlap with old mode tasks.

The mode change protocol presented by Real and Crespo [83, 84] utilizes the protocols developed by Tindell et al. [99], and by Pedro and Burns [78] with a focus on four objectives during a mode transition. The objectives are:

- to maintain the *schedulability* of the system in the old and new mode but especially during mode transition,
- to maintain the *periodicity* of unchanged tasks during mode transitions,
- to maintain the *consistency* of shared resources during mode transitions, and
- by completing the mode transition before a given time instant, also referred to as *Mode Change Deadline (MCD)*, facilitating the *promptness* of mode change handling.

The presented protocol has to find a balance between the four objectives since they represent contradicting requirements [84].

In contrast to the more complex asynchronous protocols, a simple synchronous protocol keeping the implementation effort as well as the complexity of the analysis low, was presented by Tindell and Alonso [98]. The objectives of this mode change protocol were the following:

- the protocol must not lead to violation of the temporal requirements of tasks,

- mode transitions have to be accomplished within a bounded time interval,
- the protocol should not require that old mode tasks are aborted upon the arrival of an MCR event, and
- the protocol should preserve the activation pattern of unchanged task.

The *idle period* has been identified to fulfil the aforementioned objectives for a mode transition. The idle period is the time interval where no real-time tasks with temporal requirements are ready to execute. In a 100% utilised system the idle period can represent the *idle instant* at the end of the major cycle of the taskset.

Since the tasksets of the old and new mode do not overlap, one of the classic schedulability analysis techniques [8, 58] for the steady system states can be applied. Also in the presence of shared resources and the application of the priority ceiling protocol, no additional rules are required [84]. The resource ceilings can be also safely adjusted during the idle period due to the non-overlapping schedule of the old and new mode tasks. The disadvantage of this simple synchronous protocol is that the worst case latency of the mode change is equal to the worst-case response time of the lowest priority old mode task.

For the envisaged real-time systems with flexible behaviour, we also utilise the simple idle instant mode change protocol to change the application's server parameters by a resource managing middleware and the corresponding taskset setting by the corresponding application's main task (see Algorithm 1).

The time required for a mode change operation is usually not negligible. Since old tasks have to be stopped, new tasks have to be started and perhaps shared resource related adjustments need to be carried out as well, overhead of the mode change operation needs to be considered in the schedulability analysis. For fixed-priority scheduled real-time systems with the simple idle instant mode change protocol there is an easy solution [98]. Subsequent to a mode change request event, right after the idle instant, a critical section with the highest ceiling priority in the new mode is considered to carry out all the necessary mode change operations.

The use of shared resources in multi-mode real-time systems requires also a special attention, since resource access policies developed for systems with only a single mode of operation cannot be reused without modification. Due to changes of task temporal parameters, the task priorities might change as well. Furthermore, the deletion of old tasks and the release of new tasks between two distinct modes usually also affect the resource ceiling [85].

An approach to determine the priority ceiling of shared resources in multi-mode systems, was proposed by Real and Wellings [85]. This approach, also referred to as *ceiling of ceilings*, first determines the priority ceiling of a shared resource in each mode of the system. In the next step, each resource is assigned the highest priority ceiling that it may have in the various system modes of operation. The general *ceiling of ceilings* approach, however, has to be complemented by a task priority re-scaling step [85] in order to avoid the bounded, but in the worst-case excessive, priority inversion.

A simpler solution was presented by Real and Crespo [84] pointing out that the IPCP can be applied to determine the priority ceiling of shared resources in the case of synchronous mode change protocols, i.e. if old mode and new mode tasks do not overlap their execution. In this case the priority ceiling of shared resources can be adjusted right after the idle instant without requiring any additional adaptation.

Mode change protocols have been extensively studied in the context of single application multi-mode systems. Existing investigations showed that asynchronous MCPs might reduce the mode transition phase and initiate the tasks of a new mode of operation earlier than synchronous protocols would do. However, they are more complex to implement and shared resource access policies require special attention if old and new mode tasks overlap their execution. In contrast, synchronous mode change protocols like the idle instant MCP require only a controlled termination of old tasks before new tasks are started. Hence, the handling of shared resources and the classical shared resource access policies do not require any special treatment or modification.

Since the focus of this thesis is on the development of efficient server parameter selection algorithms, we will utilise the idle instant mode change protocol due to its simplicity.

2.7 Server parameter selection

A crucial step in the process of temporal partitioning and integration of flexible applications in an open real-time system, is the selection of appropriate server parameter values that determine the resource reservation for the associated application and enforce predetermined temporal specifications.

As noted in the introduction, in fixed-priority scheduled systems the selected server parameter values can be considered either as globally or locally optimal [28]. The determined server parameter values are denoted as *globally optimal* if the minimum system utilisation, including all the other servers in the system, is achieved. The server parameter values of a single server, considered in isolation, are *locally optimal* if the obtained server utilisation is the smallest value compatible with scheduling the associated application's tasks. We note that the distinction of server parameter optimality is specific to fixed-priority scheduled systems and in the case of Earliest Deadline First scheduling, local and global optimality would be the same since the system can be 100% utilised.

In this thesis our main focus is on locally optimal server parameter values. The main challenge in the selection of locally optimal server parameters is that the demand requirements of the associated application are satisfied while the processing time reservation for the application and the application context switch time is kept to a minimum. The server execution capacity and period parameters leading to this minimal resource reservation (i.e. minimum server utilisation) are considered as the *optimal server execution capacity* and *optimal server period.* Moreover, the optimal execution capacity and period is subsumed under the term *bandwidth optimal parameters* or simply just *optimal parameters.*

A usual approach [6, 38, 39, 64, 65] in the calculation of server parameters is to utilise information about the processing demand produced by an application's taskset and the execution capacity supply of the server. Generally there are two models available to describe a server's execution capacity supply. First, the exact model can be expressed by the supply bound function, characterising the exact capacity supply pattern of the server. Second, a simpler but also less accurate linear supply model characterising a continuous capacity supply of a slower processor. Using either of the server capacity supply models, some approaches [38, 64] further simplify the server parameter calculation by setting certain server parameters to a specific value and determining the optimal value for the variable parameters (e.g. calculating the minimal server capacity for a constant server period value) within a given search space [39, 65].

However, various deficiencies of existing approaches inhibit the calculation of locally optimal server parameters. The origin of these deficiencies are:

- the application of a linear server supply model [6, 64] for server parameter calculation,
- the requirement of additional server parameter constraints (e.g. the initial delay of the server supply function [64] or the specification of the server period [38]) that need to be specified by the application designer,
- the definition of a search space that might not contain the optimal server parameters [65], and
- the definition of an optimal server period upper bound [39] that is generally too large to efficiently constrain the search space.

Using a *linear supply function* defined by an availability factor and a partition delay, Mok et al. [75] presented the bounded-delay resource partition model for *Earliest Deadline First (EDF)* scheduled systems. Feng and Mok [40] extended the bounded-delay model by an additional constraint that specifies the smallest scheduling quantum of a server. This constraint limits the number of server context switches and reduces the resulting context switch overhead. The application of the bounded-delay resource model [75] on the selection of periodic server parameters was investigated by Shin and Lee [91] in EDF and fixed-priority scheduled systems, but without the option to determine the optimal parameters. Since the bounded-delay model is a general resource model, there may exist many periodic servers that satisfy a given linear supply function.

A linear model representing the server supply is also used by Lipari and Bini [64]. The server model is characterised by an initial delay that occurs before the server supply starts at a uniform rate. This model has the same disadvantage as the bounded-delay model. Lipari and Bini extended their linear model in [65] and provided a recurrence relation to determine the optimal parameters of a periodic server. However, both approaches require extensive optimization of a cost function in order to obtain the optimal server parameters.

Using an *Explicit Deadline Periodic* model, Easwaran et al. [38] presented an algorithm to determine the optimal server execution capacity for a given server period. Though, the calculation of the optimal server period was not explored.

Exact schedulability conditions (based on the server supply function and the demand bound function) necessary to determine whether an application's taskset is schedulable or not is specified by Lipari and Bini [65] and Easwaran et al. [39]. The use of exact schedulability conditions, to determine the optimal server parameters, implies a brute-force search over the interval of possible server periods. To limit the search space of possible server periods, the least common multiple of task periods and the smallest task deadlines were suggested by Easwaran et al. [39] and by Lipari and Bini [65], respectively, as an upper bound on the server period search space containing a sufficiently good server period value. In a later chapter we will show by an example that the upper bound on the optimal server period may be larger than the smallest task period or deadline, and also that it is significantly smaller than the least common multiple of task periods.

The selection of periodic server parameters based on the task demand function was analysed by Almeida and Pedreiras [6]. They examined the demand at each priority level and determined relevant demand points that were used as input for the server parameter calculation algorithm. The information about the taskset demand at the time instant of task deadlines was used in order to determine the required processing time of the corresponding taskset. This is a pessimistic assumption since there might be a lower taskset demand at an earlier time instant than at a task deadline. Furthermore, their approach also uses a linear server supply function to determine the final server execution capacity and period.

The calculation of the optimal *Time Division Multiple Access (TDMA)* slot and cycle length for hard real-time load was analysed by Wandeler and Thiele [100]. However, the presented model has restrictions that assume non-preemptible slots, and single slot reservation for each communicating node or task within a TDMA cycle. Considering these restrictions of the model in the context of real-time scheduling, the proposed approach could be best applied only to systems with a cyclic scheduler specifying static schedules.

The selection of optimal server parameters was also investigated by Davis and Burns [28]. Though, they took a slightly different approach. For a set of periodic servers, the parameter selection was investigated if two out of the three server parameters (i.e. execution capacity, period or priority) were given, and the interrelation of server period values was examined in detail.

Fisher [41] proposed a fully polynomial-time approximation scheme for the server parameter selection. His algorithm determines the server parameters for an EDF scheduled sporadic taskset, searching for the server period within an interval bounded by a lower and upper bound. The resulting server utilisation is at most $(1 + \epsilon)$ times larger than the optimal value, with the accuracy parameter variable in the range $0 < \epsilon \leqslant 1$. However, the server period lower and upper bound was not specified [41].

In real systems the server context switch overhead usually consumes a significant amount of time and it is an important aspect in the selection of appropriate server parameters [20]. For fixed-priority scheduled systems this overhead was considered in the parameter selection by Lipari and Bini [64], and Almeida and Pedreiras [6] as a cost function. However, the server context switch considerations were embedded in the context of the less accurate linear server model. For dynamic priority servers (i.e. Constant Bandwidth Server) a different approach, incorporating the context switch time into the server capacity, was taken by Buttazzo and Bini [20]. That means, the execution capacity provided by a server to the associated tasks is reduced by the time required for the execution of the server context switch.

Finally, in the past various papers [26, 35, 53, 87] provided in-depth analysis of hierarchical real-time systems, leaving the question of efficient server parameter selection open by implying the availability of the necessary algorithms.

In a multi-application fixed-priority scheduled systems where multiple servers manage the resource reservation for different application tasksets, the combination of servers with locally optimal parameters might result in an unschedulable system. In such a scenario the server scheduler inhibits at least one of the servers from supplying its full execution capacity within the replenishment period to the associated taskset. Therefore, at least one task in the corresponding tasksets cannot satisfy its timing requirements.

In such a situation it might, however, be possible to create a schedulable system by using locally non-optimal parameter values for one or more server. Hence, server parameter values that result in a schedulable system are globally optimal if the total system utilisation, calculated as the sum of all server utilisations, is the lowest for all possible server parameter allocations.

Ultimately, the locally and globally optimal server parameter values may be equivalent. If the combination of servers with locally optimal parameters is feasible on a fixed-priority scheduled system, then the locally optimal values are also globally optimal. Conversely, globally optimal values may either be locally optimal or non-optimal.

2.8 Compositional real-time frameworks

The challenge of server parameter selection is taken to the next level in the context of compositional frameworks where the main focus is on abstraction and interface generation. The problems addressed by compositional frameworks, however, have a strong link to server parameter selection.

The approach with compositional frameworks [38, 66, 91–93] is to abstract the resource requirements of a taskset and generate an interface description describing the taskset's resource needs. The notion of interface description has certain similarities to the FRESCOR contract model.

Interface descriptions enable an efficient online test to determine if a new set of real-time applications is schedulable or not. In compositional frameworks the objective is also the selection of appropriate interface descriptions with minimal wasted processing resources. Finding appropriate interface descriptions can be considered as the search for locally optimal server parameters.

A compositional framework based on the periodic resource model, i.e. periodic server, was presented by Shin and Lee [91–93]. They used the *period multiple relationship* between the tasks and the corresponding server in order to solve the server parameter selection problem. Additionally to this restriction, it is also assumed that the server period value is already given, and only the execution capacity needs to be determined. Similarly, Easwaran et al. [38] derived the equations that enable the calculation of the smallest required server capacity for a predetermined server period. However, again the server period must be specified manually.

The *FIRST scheduling framework (FSF)* [4] addressed the need for a scheduling framework that could handle applications with varying processing demand. An online algorithm integrated into a middleware implementation of the proposed framework adapted the flexible server parameters using a utilisation-based schedulability test. Hence, within the range of given server parameters, the best suitable values could be chosen. However, since the utilisation-based schedulability test is not exact, the processing resources wasted by each application are higher than what could be achieved by the application of an exact schedulability test.

In contrast to the FSF, an improved online server parameter selection was developed and also integrated into the FRESCOR framework. The proposed improvements, to obtain an efficient online parameter selection algorithm for open real-time systems, facilitating the composition of real-time application during runtime, will be presented in Chapter 6.

2.9 Summary

The necessary theoretical foundation for flexible open real-time systems was provided by the introduction of the classical real-time system model and the analysis of these systems. Extensions and complementary work to the classical model, like hierarchical system model, scheduling frameworks, mode-change protocols and server parameter selection, provided the necessary tools to design real-time systems that reflect the current state-of-the-art of flexible real-time systems in robotics, and the automotive and aerospace industries. However, the link between scheduling schemes considering flexible task parameters and the appropriate server parameter selection was not addressed in previous literature.

This thesis builds on previous work (i.e. hierarchical systems, the FRESCOR project and server parameter selection), utilises and merges certain properties of those models to define a simpler framework than that envisaged by the FRESCOR project. A framework with a primary focus on the processor as a shared resource, is expected to enable efficient taskset analysis and server parameter selection for temporal partitioning. That means, the focus is not just on the server parameter selection but also on the investigation of efficient implementations for small embedded systems of offline and online optimal server parameter selection algorithms. In addition, the flexibility of tasks is considered in our approach starting at the taskset specification level and the parameter selection of the associated servers up to runtime processing time adaptation for these tasksets.

In summary, the research results presented in the following chapters explore methods that enable efficient realisation and implementation of solutions that satisfy the subset of the FRESCOR requirements for flexible open real-time systems, listed in Section 2.4.

Chapter 3

Optimal server period interval

The previous chapter gave an overview of topics that are related to and linked with the efficient server parameter selection problem. The review showed that most server parameter selection approaches either assume a given server period and calculate only the corresponding execution capacity, or provide only theoretical approaches without presenting any efficient implementation.

In order to determine the optimal server parameters, we divide this complex problem into smaller problems. Assuming that the temporal specification of a taskset is given (i.e. WCET, period and the deadline of tasks), first the processing resource requirements of the corresponding tasks is expressed as demand points to simplify the subsequent steps of the optimal server parameter determination process. Using the demand points, next, the optimal server period interval is determined for a given taskset. We argue that the complexity of the optimal server parameter selection problem can be significantly reduced by first determining the optimal server period interval (i.e. an interval containing the optimal server period value). A few pragmatic solutions were proposed in the past as an upper bound for the optimal or a sufficiently good server period value. However, in previous work the proposed optimal server period interval upper bounds are either unsafe [65] or a lot larger [39] than the value that we determine as an upper bound. Finally, the knowledge about the optimal server period interval enables the calculation of the optimal or near-optimal server period value, depending on the applied server period determination method.

In the remainder of this chapter, we summarise the system model of an open flexible real-time system and recall definitions that were already mentioned in Chapter 2. Then we provide insight into the server utilisation behaviour as a function of server parameters. As a starting point for the optimal server parameter determination algorithm, the optimal server period interval is derived. The methods to determine the optimal or near optimal server period value will be presented in Chapter 4.

3.1 System model

The real-time systems on which we focus our attention, are mainly located in the domain of automotive, aerospace or robotics, performing various real-time control or processing tasks.

The general architecture of these systems usually utilises a two-level hierarchical scheduling approach, with multitasking applications directly scheduled by the operating system and the tasks within each application's taskset by an application level scheduler. Hence, the task of the operating system level scheduler, also referred to as the *global scheduler*, is to determine the processor time allocation to the flexible real-time multitasking applications. Whenever the processor is allocated to an application, the application level scheduler, denoted as *local scheduler*, determines the execution schedule of the corresponding taskset's ready tasks.

Formally each application A_q represents a set of n_q tasks, i.e. $A_q := \left\{\tau_1, \ldots, \tau_{n_q}\right\}$. The cardinality of each taskset may be different, i.e. the number of tasks in an application is not related to or limited by the number of tasks in other applications in the system.

Within an application, each task τ_i is defined by the temporal parameters (C_i, T_i, D_i). The parameter C_i denotes the task's worst-case execution time, D_i its deadline and T_i the period of a periodic task or minimum interarrival time of a sporadic task. The task's deadline may only be less than or equal to its period, $D_i \leqslant T_i$, enabling only a single task release within the task period. This implies that the system does not execute two or more concurrent jobs of the same task.

In order to capture the temporal specifications of tasks with flexible processing, the classical tasking model requires certain extensions [33]. Based on the flexibility that a task can handle, the supported set of tasks is grouped into two categories, i.e. discrete and continuous tasks.

The *discrete* periodic task model supports the specification of a finite number of implementations, also referred to as *version*, for a certain task. Each implementation may have a different temporal specification. A task τ_i with m different implementations and execution behaviours is specified by $\left(C_{i(v)}, T_{i(v)}, D_{i(v)}\right) \in \left\{\left(C_{i(1)}, T_{i(1)}, D_{i(1)}\right), \cdots, \left(C_{i(m)}, T_{i(m)}, D_{i(m)}\right)\right\}$ [33]. The v-th version of a task τ_i is specified by the tuple $\left(C_{i(v)}, T_{i(v)}, D_{i(v)}\right)$ implying a link between the task's period, deadline and assumed worst-case execution time.

The *continuous* task model enables the specification of intervals for each temporal parameter. That means, continuous tasks can operate with temporal parameter values selected from specified intervals. We limit ourselves to intervals with integer granularity since the majority of real-time systems use integer values for the specification of temporal parameters. Hence, for a continuous task τ_i the task period and worst-case execution time are defined as $T_i \in \left[T_{i(\min)}, T_{i(\max)}\right]$ and $C_i \in \left[C_{i(\min)}, C_{i(\max)}\right]$, respectively.

Despite the extension to flexible parameters the deadline constraint of the classical tasking model, $D_i \leqslant T_i$, also applies to the flexible tasking model.

Due to the assumption of a fixed-priority scheduled system, each task has a unique priority within its associated application. The task priority within an application is expressed by the subscript i, with the value 1 representing the highest and n_q the lowest priority. For ease of presentation the number of tasks n_q in an application A_q will be simply depicted as n. The task priority i is determined according to the deadline monotonic priority assignment policy.

Temporal partitioning among flexible applications in the system is enforced by periodic execution-time servers that implement the appropriate resource reservation mechanism. An execution-time server is responsible for the management of the predefined execution capacity reserved for the associated application's taskset.

Without loss of generality, in the remainder of this thesis we assume that an *application* A_x is associated with only one *execution-time server* S_x, implying a one-to-one mapping between applications and servers. However, the decision that an application's taskset is mapped to one or more servers is considered as a design decision that does not impact the envisaged approach for server parameter selection. If an application's taskset was mapped to multiple servers, then the design decision could be to divide the application's taskset into multiple smaller sub-tasksets. The server parameter selection approach would consider each of these sub-tasksets as a self-contained application. We assume that the application or system designer defines the separation of the application's taskset into multiple sub-tasksets. After the separation of the application tasks, each of the sub-tasksets can be mapped to a dedicated server and the server parameters can be determined using the methods presented in this and the following chapter.

Although the separation of a single application's taskset into multiple smaller tasksets does not require a special consideration in the server parameter selection algorithm, it has a minor impact on the equations calculating the optimal server parameters if shared resources are used. This issue will be briefly elaborated in Chapter 7.

The type of execution-time server that we utilise in this research is the periodic server model [6, 41, 64, 65, 91, 93]. An execution-time server S_x is described by its execution capacity Θ_x that the server supplies within its replenishment period Π_x to the associated taskset.

The execution capacity Θ_x is decreased by the same amount that the executed tasks of the corresponding application A_x consumed. If the server's execution capacity is exhausted, the server (and hence the associated application) is suspended until the execution capacity is again replenished to its full capacity. The time instants of server capacity replenishment are determined by the server's replenishment period Π_x.

In contrast to the initial specification of the periodic server behaviour, we utilise a minor extension. The periodic server's execution capacity will not be instantaneously exhausted if the ready queue of the associated application's taskset is empty, i.e. there are no ready tasks in the application waiting for execution. In this case the periodic server's execution capacity is continuously consumed by an idle task until it is exhausted [26].

Analogous to the flexible tasking model, the temporal parameters of the classical periodic server model are also extended. Hence, execution-time servers are also classified as continuous or discrete servers [47] (also see Section 2.4.3).

The temporal parameters of a *discrete* execution-time server S_x are defined as $(\Theta_x, \Pi_x) \in \{(\Theta_1, \Pi_1), \cdots, (\Theta_v, \Pi_v)\}$ with Θ_x denoting the selected server execution capacity and Π_x the corresponding replenishment period. Similarly, the parameters of a *continuous* execution-time server S_x are defined as $(\Theta_x, \Pi_x) \in \big\{\big([\Theta_{x(\min)}, \Theta_{x(\max)}], [\Pi_{x(\min)}, \Pi_{x(\max)}]\big)\big\}$.

With respect to the temporal partitioning among tasksets, the current system model enables only to link discrete tasks with discrete servers, and continuous tasks with continuous server. However, tasks with fixed temporal parameters can be mixed with either continuous or discrete tasks. In such a configuration, the appropriate task model is used and for a fixed task τ_i each flexible temporal parameter is set to the same value, i.e. $C_{i(\min)} = C_{i(\max)}$ and $T_{i(\min)} = T_{i(\max)}$ using the continuous task model, and $\forall_{v,w} \big(C_{i(v)}, T_{i(v)}, D_{i(v)}\big) = \big(C_{i(w)}, T_{i(w)}, D_{i(w)}\big)$ using the discrete task model.

In a system with multiple execution-time servers, each server has a unique fixed-priority denoted by P_x. Based on the server priorities, a fixed-priority scheduling scheme is used to determine the processor time assignment to a server and the associated taskset. Although execution-time servers are only resource reservation mechanisms and do not represent executable entities, we will still use a similar terminology as it is common for task scheduling. In other words, a server starts executing if the scheduling scheme determined that processor time shall be assigned to it; it will be suspended if the scheduling scheme determines that processing time shall be assigned to an other server, or it terminates if the system shuts down or a task within the associated taskset requests the termination of the entire application.

Although deadline monotonic scheduling is an optimal fixed-priority scheduling scheme for sporadic tasks with $D_i \leqslant T_i$, it was shown [28] that this optimality no longer holds for servers, once server context switch overhead and task schedulability are taken into account. Therefore the server parameter selection approach relies on a fixed-priority scheduling scheme for tasks but it is not limited to the deadline or rate monotonic priority assignment.

3.2 Execution-time server properties

An important aspect that requires special attention in the analysis of tasksets and the determination of bandwidth optimal server parameters is the server context switch overhead [20]. In uniprocessor systems that enforce temporal partitioning among different tasksets by applying execution-time server mechanisms, the execution context needs to be stored and restored. Execution contexts usually contain taskset specific memory ranges and state information that needs to be saved when the associated execution-time server is suspended. Subsequently the execution context of the newly scheduled server and the associated taskset has to be restored in order to ensure that the tasks of the current taskset have access to the data that was stored before the taskset's suspension.

The server context switch overhead involved in the scheduling of execution-time servers (represented by C_0) is considered as a system specific value provided by the system designer or integrator. The context switch time that might be consumed at the beginning and at the end of the server execution is subsumed into a single value (C_0) and in the schedulability analysis the worst-case timing is assumed for it. Furthermore, for the schedulability analysis of tasks the critical instant of τ_i is considered to occur in the following situation (see Figure 3):

1. The release time of tasks is not synchronised with the associated server's replenishment period.
2. The server's capacity has been consumed at the beginning of the server period by tasks with priority lower than task τ_i.
3. Task τ_i and all tasks with priority higher τ_i are released just after the server's capacity has been exhausted. At the same time, the server context switch C_0 occurs as well.
4. In the following server periods, first, the server context switch C_0 precedes the server capacity supply and second, the server capacity supply experiences the maximum possible delay due to higher priority servers. That means, during these server periods, the server supply occurs right at the end of the server period and furthermore, the server supply is immediately preceded by the server context switch.

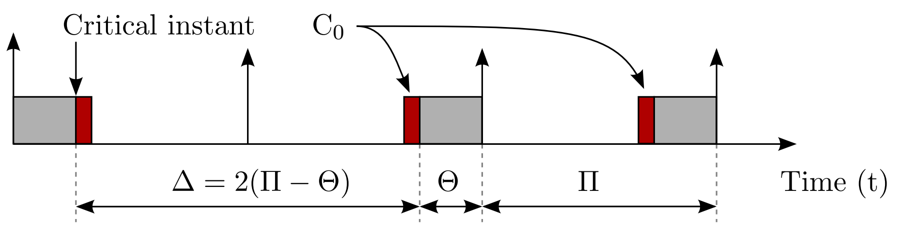

Figure 3: Server supply including context switch overhead

We note that the described worst-case occurrence of server context switch and task release time, represents a pessimistic model giving the maximum server supply delay, that usually will not occur in real systems.

Figure 3 illustrates the worst-case server supply scenario including the critical instant, server context switch overhead and the maximal server supply delay.

The time reserved for the server context switch C_0 is not subtracted from the server capacity since it is considered as a system function. The effect of the server context switch time can be considered identical to the interference caused by higher priority servers. Hence, the entire server capacity is available to the associated application's taskset.

Based on the parameters of a periodic server, the server supply bound function $s(t)$ can determine the cumulative execution capacity that a server supplies to the associated taskset within a time interval t. However, within a time interval between two server execution capacity replenishment events, there is a time interval without execution capacity supply. This inactive time interval arises from the fact that the server capacity is usually smaller than it replenishment period and therefore the server cannot provide a continuous execution capacity supply to its taskset. The maximum inactive time of the server can be expressed as $\delta = \Pi - \Theta$. Due to the server's inactive time interval, the server supply bound function for periodic servers is best described by a piecewise defined function (see Equation 3.1 [6]).

$$s(t) = \begin{cases} 0 & : t < \Delta \\ t - (\Delta + m\delta) & : \Delta + m\Pi \leqslant t < \Delta + m\Pi + \Theta \\ (m+1)\Theta & : \Delta + m\Pi + \Theta \leqslant t < \Delta + (m+1)\Pi \end{cases} \tag{3.1}$$

$$m = \lfloor (t - \Delta) / \Pi \rfloor$$
$$\Delta = 2\delta = 2(\Pi - \Theta)$$

The first case of Equation 3.1 specifies the initial delay of the server supply. During this time the cumulative processing time provided by the server is 0. The second case defines the continuous supply of processing capacity to the tasks, until the server capacity is exhausted. Hence, the cumulative processing time continuously increases with time. Finally, the third case specifies the supply gaps within the server period, i.e. time intervals without capacity supply.

The main objective in the selection of optimal server parameters is the minimisation of the server supply provided that the associated taskset remains schedulable. In order to facilitate the optimal server parameter selection, the information about the server supply can be summarised in a more abstract form as the server utilisation.

As the server context switch is assumed to have a constant execution time for a given system, the overhead caused by the context switch operations varies with the selected server period value. The server context switch overhead, expressed in terms of processor utilisation, increases with decreasing server period since in that case the context switch operations with a fixed duration are carried out more frequently. This also implies that for the selection of the optimal server parameters the server context switch overhead as a function of the server period has to be considered as well.

Since the server execution capacity and server context switch are considered separately, both terms have to be specifically considered in the calculation of the server utilisation. The server utilisation with incorporated context switch overhead ($\hat{U}$) can be determined as defined by Equation 3.2.

$$\hat{U} = U + \frac{C_0}{\Pi} = \frac{\Theta}{\Pi} + \frac{C_0}{\Pi} = \frac{\Theta + C_0}{\Pi} \tag{3.2}$$

For the sake of simplicity, in the remainder of this thesis we will refer to *locally optimal* parameter values and servers simply by the term *optimal*. Furthermore, the server execution capacity and period parameters are defined as the *optimal server execution capacity* and *optimal server period*, or generally the *optimal parameters*, if the resulting resource reservation (i.e. the server utilisation) for this particular server is the minimum reservation required to schedule the associated application's taskset.

3.3 Analysis of optimal server resource needs

In order to efficiently determine the optimal server temporal parameters, we analyse for a given taskset the minimal server utilisation with server replenishment period as the independent variable and approach the problem by dividing it into two sub-problems. Various solutions [33, 38] were presented in the literature for the sub-problem that is concerned with the calculation of the optimal server execution capacity. The second sub-problem is more complex and it is concerned with the determination of the optimal server period. However, in order to efficiently calculate the optimal server period, we first determine a formally proven safe interval that contains the optimal server period value.

3.3.1 Taskset demand and demand points

In this section we are concerned with the demand analysis of a single application's taskset facilitating the calculation of the optimal server period interval and the actual optimal server period value.

Given the temporal specification of each task in a taskset, the demand caused by the taskset at different priority levels can be determined by the *demand bound function* [58]. In order to simplify the analysis of the task requirements and the server parameter calculation, the demand requirements for each priority level i (expressed by the demand bound function) are mapped to a representative *demand point* G_i. A demand point G_i is defined as a pair (q_i, t_i), containing information about the demand q_i that occurs at time instant t_i and at priority level i.

The demand bound function for a priority level i is defined in Equation 3.3 with $hep(i)$ denoting priority levels equal to or higher than i.

$$dbf_i(t) = \sum_{j \in hep(i)} \left\lceil \frac{t}{T_j} \right\rceil C_j \tag{3.3}$$

The analysis of the expression $f(t) = dbf_i(t)/t$ shows that the priority level i demand per time interval, has local minima at time instants that are multiples of task periods. This was also observed by Lehoczky et al. [58] and Easwaran et al. [38]. Lehoczky et al. [58] defined these time instants, denoted as scheduling points, for the priority level i as $Q_i = \{kT_j | j = 1, ..., i; k = 1, ..., \lfloor T_i/T_j \rfloor\}$. Bini and Buttazzo [13] introduced a method to create a reduced set of time instants that are still sufficient to find the local minima of $f(t) = dbf_i(t)/t$. Considering the worst-case situation for a taskset, where all tasks are released simultaneously, Equation 3.4 determines the last possible idle instant at the priority level of each examined task within the time interval specified by the parameter of the recurrent equation. For the time instants, recursively defined by $P_i(t)$ (Equation 3.4), Bini and Buttazzo furthermore showed that $P_{i-1}(T_i) \subseteq Q_i$ [13]. We note that the subscript i of $P_i(t)$ is only an index and does not correspond to the priority level of the examined task in the equation.

$$\begin{aligned} P_0(t) &= \{t\} \\ P_i(t) &= P_{i-1}\left(\left\lfloor \frac{t}{T_i} \right\rfloor T_i\right) \cup P_{i-1}(t) \end{aligned} \tag{3.4}$$

Exploiting the aforementioned facts about local minima of the function $f(t) = dbf_i(t)/t$ and determining a specific demand point G_i for each priority level i, allows us to represent the demand requirements of an entire taskset by a set of demand points $\mathbb{G}$.

We present an approach in Algorithm 5 to determine the demand points that represent the minimum demand requirements of a given taskset. Since the runtime of the algorithm depends not only on the number of tasks but also on the value of the task periods, its time complexity is pseudo-polynomial. However, measurements of the algorithm's actual runtime for various tasksets, that will be presented in the evaluation section of Chapter 4, confirm that the algorithm is still applicable for real systems.

```
    In  : Priority ordered taskset
    Out : Set of demand points
1 $\mathbb{G} = \{\}$;
2 foreach priority level $i$ do
3     At each priority level $i$ determine a set of $(q_m, t_m)$ pairs. Each
      $(q_m, t_m)$ pair represents a demand point $G_m$ with the cumulative
      demand $q_m$ of the tasks $\tau_j : j \in hep(i)$ at a time instant
      $t_m \in P_{i-1}(D_i)$;
4     For priority level $i$ let $G_i$ denote $(q_i, t_i) = (q_m, t_m) : \min_m (q_m/t_m)$
      with the smallest ratio $(q_m/t_m)$. If multiple $(q_m, t_m)$ pairs satisfy this
      condition, then choose the one with the largest $t_m$;
5     $\mathbb{G} = \mathbb{G} \cup (q_i, t_i)$;
6 end
7 Remove redundant demand points from $\mathbb{G}$;
```

Algorithm 5: Demand point calculation

In summary, the algorithm determines for each priority level i a set of time instants $P_{i-1}(D_i)$. At the examined priority level i, for each time instant $t_m \in P_{i-1}(D_i)$ the corresponding demand q_m is calculated as $dbf(t_m)$. From the set of (q_m, t_m) pairs, the one with the lowest ratio is selected and denoted as the representative demand point G_i for priority level i. Before the algorithm terminates, redundant elements are removed from the set of demand points. A demand point (q_r, t_r) is considered to be a redundant element if there is another element (q_i, t_i) such that $t_r = t_i$ and $q_r < q_i$. This means, for a specific time instant t only the demand point with the largest demand is retained. The removal of redundant points from the set of demand points (last step in Algorithm 5) also implies that every time instant t_i is associated with only one demand requirement q_i.

The rationale to discard redundant elements is linked to the analysis of fixed-priority scheduled systems. In order for a taskset to be schedulable by an execution-time server, the server always has to supply processing capacity that is equal to or larger than the taskset demand. Hence, when multiple tasks are released at a certain time instant, only the maximum demand is important for the schedulability analysis and for the server parameter selection.

With respect to the number of demand points in $\mathbb{G}$, the cardinality of $\mathbb{G}$ can be determined as follows. The system model defines a unique priority assignment for a given taskset. Therefore, the number of priority levels and tasks in a taskset is identical. Algorithm 5 determines a single representative demand point for each priority level, and at the end, redundant demand points are removed from the set $\mathbb{G}$. Hence, the number of demand points in $\mathbb{G}$ is less than or equal to the number of tasks in the given taskset.

3.3.2 Example

The following short example demonstrates the application of Algorithm 5 on the taskset defined in Table 1. The deadline D_i of each task is set equal to its period T_i.

Table 1: Taskset example

Task	Priority		C_i	T_i
A	1	(high)	400	1300
B	2	(medium)	800	4600
C	3	(low)	1000	6800

The presented values are expressed in time ticks. We use *time tick* as the unit for all temporal parameters and it represents the granularity of the temporal parameter values. Given the taskset specification (i.e. the worst-case execution time, period and deadline of tasks), for each priority level a representative demand point is determined. The time instants and the corresponding workload are presented in Table 2. For each priority level, in the highlighted row, the time instant with the lowest workload defines the representative demand point (i.e. $G_1 = (400, 1300)$, $G_2 = (2000, 3900)$ and $G_3 = (4600, 6500)$).

Additionally, for this example, the priority level task demand, the determined demand points and the slowest processor speed required to satisfy the priority level demand are depicted in Figures 4–6.

Table 2: Demand points

Priority level	$t \in P_{i-1}(D_i)$	Demand dbf(t)	Demand time ratio
1 (high)	1300	400	0.3077
2 (medium)	3900	2000	0.5128
	4600	2400	0.5217
3 (low)	3900	3000	0.7692
	4600	3400	0.7391
	6500	4600	0.7077
	6800	5000	0.7353

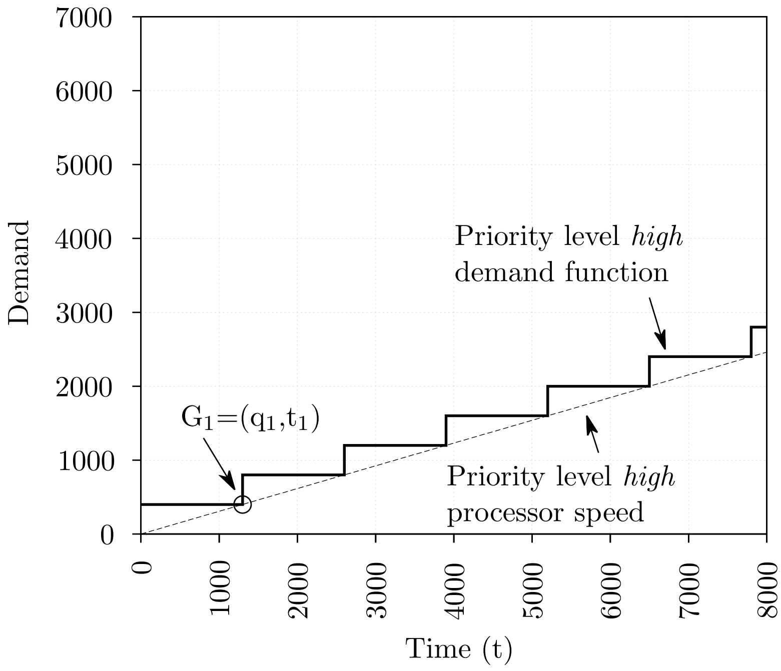

Figure 4: Example of taskset demand, demand points and server supply for highest priority level

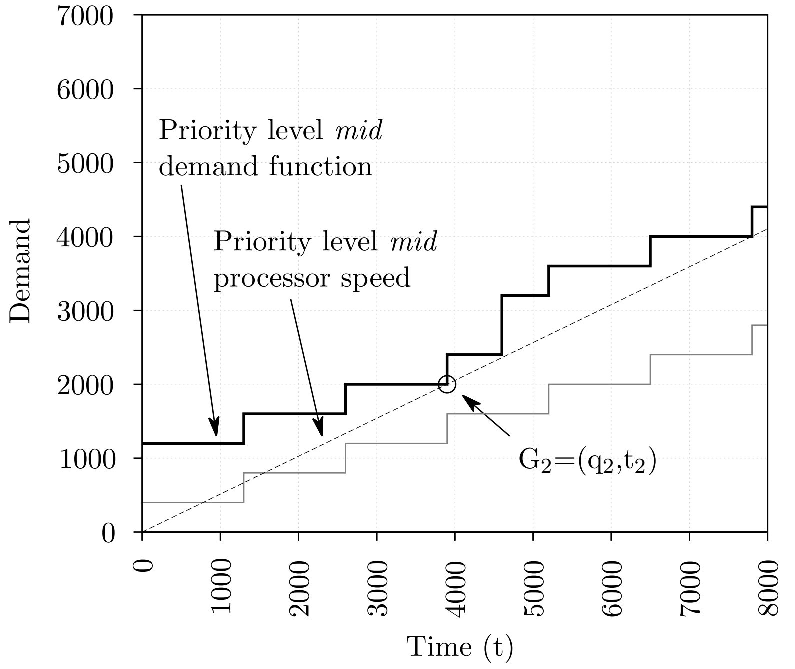

Figure 5: Example of taskset demand, demand points and server supply for middle priority level

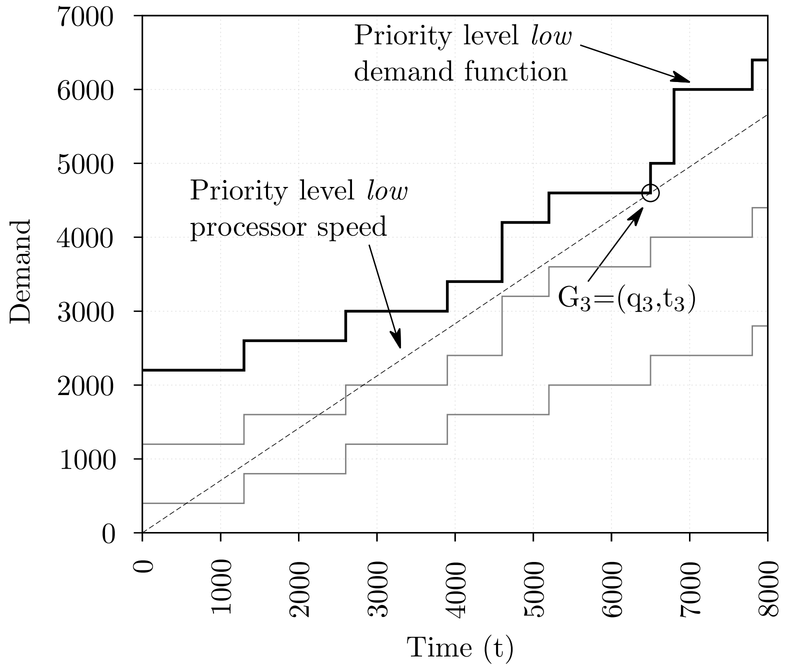

Figure 6: Example of taskset demand, demand points and server supply for lowest priority level

3.3.3 Optimal server utilisation

For a schedulable taskset, plotting the corresponding minimum server utilisation as a function of the server period, reveals a sawtooth shape graph (see Figure 7). The minimum server utilisation for a server period (further referred to as the optimal server utilisation) is determined by calculating the smallest server execution capacity such that the associated taskset is still schedulable.

Figure 7 shows for the taskset in Table 1 the minimum server utilisation as a function of the server period. The minimal server utilisation values were produced by an exhaustive enumeration. For the lower line on the graph, the server utilisation values were determined without considering server context switch overhead, and for the upper line on the graph, a context switch overhead of 100 ticks was specified.

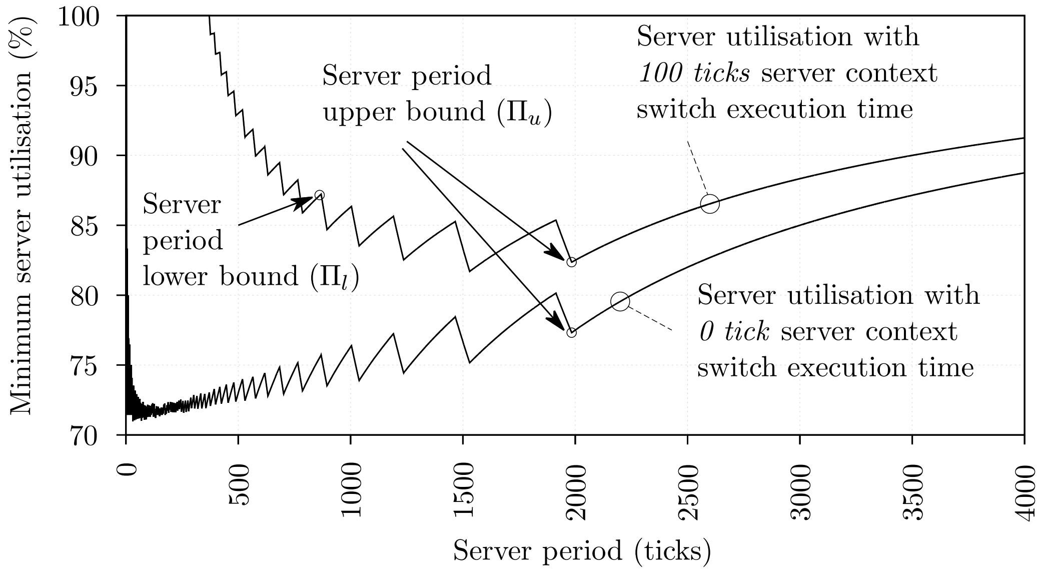

Figure 7: Minimum server utilisation

We recall that the server capacity and period values are limited to the integer domain and this is also reflected in the diagram. The server utilisation increases (even if there is no context switch overhead) towards and reaches 100% processor utilisation as the server period value gets very small and approaches 1.

Since the server parameter values belong to the integer domain and the minimal server utilisation graph shows a sawtooth shape, a sophisticated method is required to determine the server parameters that lead to the lowest server utilisation. Such a method will be successively introduced in the following sections and chapter.

3.4 Determination of the optimal server period interval

The prerequisite for the optimal server period calculation is the determination of the server period upper and lower bound. The *server period upper bound* Π_u defines a bound such that the optimal server period value must lie at or below this value. In addition, the server period lower bound Π_l specifies the smallest possible value for the optimal server period.

Due to the application of mainly integer arithmetic for the processing of temporal values in current real-time operating systems, we will use a discrete time space for the server parameters. Hence in further equations, the server period is mapped to an integer value by applying the floor function. For the same reason, the server capacity is mapped to an integer by applying the ceiling function. Generally, these discretisation steps are not limited to integer values but could be carried out to an arbitrary precision.

3.4.1 Server context switch excluded

First, we consider the scenario with no server context switch overhead. In the domain of real numbers, the optimal server period would approach the value zero if there is no server context switch. However, in real systems the time values have only a finite granularity. This assumption and the investigation of the server utilisation in Figure 7 indicates that a dedicated algorithm is required to determine the server parameters even if server context switch is not present. In Section 3.4.2 we will further examine the approach presented in this section and take into account the effect of the server context switch time.

To derive an appropriate algorithm for the server period upper bound calculation, the task and server properties have to be examined in the first place.

Since the task release times are not synchronised with the server's replenishment period, the worst-case task execution delay occurs under the following

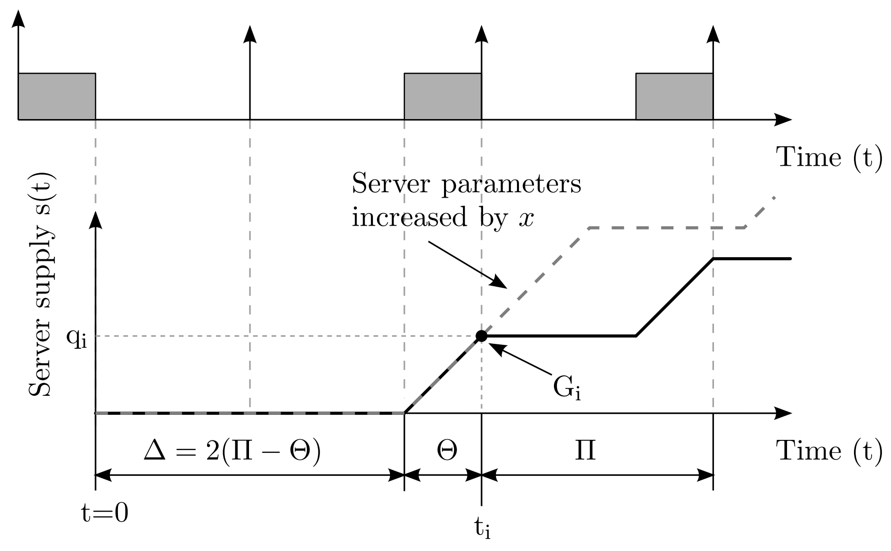

Figure 8: Server supply

circumstances. The server capacity is consumed by lower priority tasks starting immediately after the server's replenishment period. In the following server period, the execution capacity supply experiences the maximum possible delay due to higher priority servers. Therefore, after a task is released, its execution can be delayed by $\Delta = 2\delta = 2\,(\Pi - \Theta)$ in the worst-case situation (see Figure 8).

In addition, the examination of the minimum server utilisation graphs in Figure 7 indicates that there is a server period Π_u such that for all other server period values $\Pi \geqslant \Pi_u$, the server utilisation $U(\Pi)$ is monotonically increasing. Therefore, the optimal server period must lie at or below this value that we denote as the *server period upper bound* Π_u.

The following two lemmas and the subsequent theorem (Lemma 1, Lemma 2 and Theorem 1) provide the necessary statements that can be used to define the algorithm for the server period upper bound calculation. First, Lemma 1 provides an indication for an upper bound on the optimal server period. This lemma defines for a demand point G_i, that above a certain server period value, the server utilisation increases monotonically with increasing server period. This property of the server utilisation considered in Lemma 1 for a single demand point, is extended to the entire set $\mathbb{G}$ in Theorem 1. Lemma 2 enables the calculation of the longest feasible server supply delay that is maintained until the optimal server period is determined for a set of demand points. Based on the insight of the two lemmas, Theorem 1 provides the means to determine an optimal server period upper bound. Finally, in Theorem 2 we claim, the optimal server period upper bound is also an optimal server period candidate and prove that the server utilisation has a local minima at this value.

Lemma 1. (Monotonic increase of the server utilisation) *Given a demand point $G_i = (q_i, t_i)$, the optimal server utilisation $U(\Pi)$ (with a server supply satisfying the demand specified by G_i) monotonically increases for the server period $\Pi \geqslant (t_i + q_i)/2$.*

Proof. With an optimal server utilisation, the server is supplying only as much processing time as dictated by the demand point G_i. Hence, the server supply stops at G_i, implying $s(t_i) = q_i$ (see Figure 8).

Additionally, examination of the second case of the piecewise defined server supply bound function and the corresponding graph (see Figure 8) shows that a strict monotonic increase of the server utilisation occurs if the demand point G_i experiences a single server replenishment period.

Assuming $\Pi \geqslant (t_i + q_i)/2$, $s(t_i) = q_i$ and the worst-case delay $2(\Pi - \Theta)$ for the server supply, we can show that the time instant t_i is within the server's first replenishment period. Π can be also written as $\Pi \geqslant t_i - (t_i - q_i)/2$ where the second term $(t_i - q_i)/2$ represents in the critical instant the time interval without server capacity supply of the previous server period, i.e. $\Pi - \Theta$.

In order for t_i to lie within a the first server replenishment period, the condition $t_i \leqslant \Delta/2 + \Pi$ has to hold. First, we assume $\Pi = t_i + q_i/2$:

$$
\begin{aligned}
t_i &\leqslant \frac{\Delta}{2} + \Pi = \frac{2(\Pi - \Theta)}{2} + \Pi \\
t_i &\leqslant \frac{t_i - q_i}{2} + \frac{t_i + q_i}{2} \\
t_i &\leqslant t_i
\end{aligned}
$$

In the second case Π is assumed to be larger than $t_i + q_i/2$ by an arbitrary positive integer number x:

$$\begin{aligned}
t_i &\leqslant \frac{\Delta}{2} + \Pi = \frac{2\left(\Pi - \Theta\right)}{2} + \Pi \\
t_i &\leqslant \frac{t_i - q_i}{2} + \frac{t_i + q_i}{2} + x \\
t_i &\leqslant t_i + x
\end{aligned}$$

The previous two equations show that for $\Pi \geqslant \left(t_i + q_i\right)/2$, the demand point G_i experiences one server replenishment period.

A single server replenishment period up to t_i also implies that the server capacity Θ is equal to the demand requirement q_i. Furthermore, considering a single replenishment period, the supply bound function can be simplified to $s(t) = t - \Delta$ and the corresponding server period can be derived:

$$\begin{aligned}
s(t_i) &= q_i \\
t_i - \Delta &= q_i \\
t_i - 2(\Pi - \Theta) &= q_i \\
t_i - 2(\Pi - q_i) &= q_i \\
\Pi &= \frac{t_i + q_i}{2}
\end{aligned}$$

Given a single server replenishment up to t_i, we show that if the server period Π is increased by an arbitrary positive value x, i.e. $\Pi = \left(\left(t_i + q_i\right)/2\right) + x$, the server capacity needs to be increased by the same value x in order to preserve $s(t_i) = q_i$.

$$\begin{aligned}
s(t_i) &= q_i \\
t_i - \Delta &= t_i - 2(\Pi - \Theta) = q_i \\
\Theta &= \frac{q_i - t_i}{2} + \Pi \\
\Theta &= \frac{q_i - t_i}{2} + \frac{t_i + q_i}{2} + x \\
\Theta &= q_i + x
\end{aligned}$$

Additionally, the uniform increase of server capacity and period by x implies that the initial delay Δ remains constant:

$$\begin{aligned}
\Delta &= 2\left((\Pi + x) - (\Theta + x)\right) \\
\Delta &= 2(\Pi - \Theta) \\
\Delta &= 2\left(\frac{t_i + q_i}{2} - q_i\right) = t_i - q_i
\end{aligned}$$

To show that the server utilisation $U(\Pi)$ is a monotonically increasing function for $\Pi \geqslant (t_i + q_i)/2$, we define $U = \Theta/\Pi$. Considering that $\Pi - \Theta = \Delta/2 = \delta$ is constant in this case, the server utilisation can be rewritten as $U(\Pi) = (\Pi - \delta)/\Pi$. In order to prove that $U(\Pi)$ is monotonically increasing, we have to show that $\Pi_1 < \Pi_2 \Rightarrow U(\Pi_1) < U(\Pi_2)$.

$$\begin{gathered}
\Pi_1 < \Pi_2 \\
-\delta/\Pi_1 < -\delta/\Pi_2 \\
1 - (\delta/\Pi_1) < 1 - (\delta/\Pi_2) \\
(\Pi_1 - \delta)/\Pi_1 < (\Pi_2 - \delta)/\Pi_2 \\
U(\Pi_1) < U(\Pi_2)
\end{gathered}$$

□

Furthermore, it can be observed that for optimal periodic servers the demand point with the smallest difference $(t_i - q_i)$ has a significant influence on the server period upper bound calculation. It defines the maximal server initial delay $\Delta = 2(\Pi - \Theta)$ (see Figure 9). A supporting argument is presented in Lemma 2.

Lemma 2. (Maximal server initial delay) *The demand point* $G_s = (q_s, t_s) = (q_i, t_i) : \min_i(t_i - q_i)$ *defines the upper bound on the server's maximal initial delay.*

Proof. For an optimal server, satisfying the demand at G_i, Lemma 1 specifies that the maximal server supply delay Δ is equal to $(t_i - q_i)$ and the capacity supply is equal to the demanded processing time, $s(t_i) = q_i$.

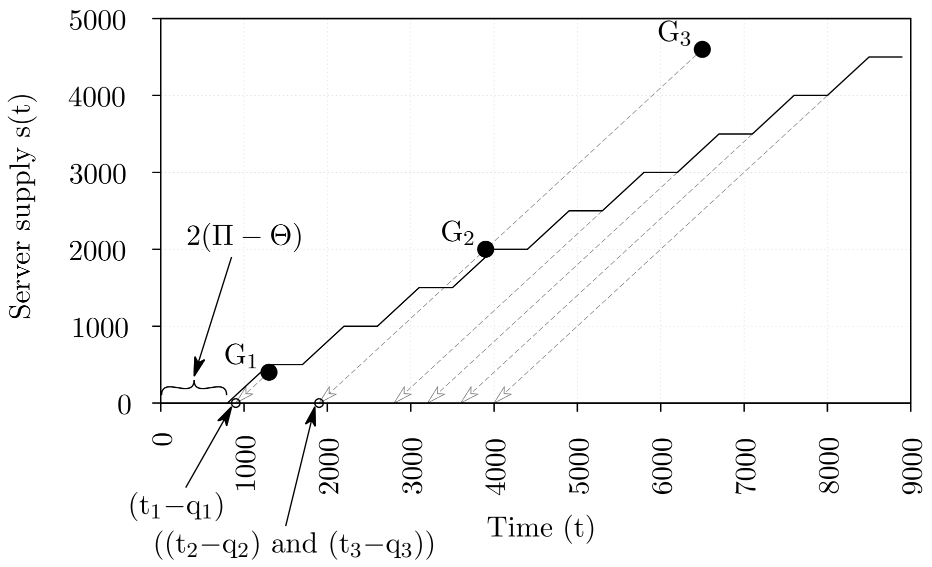

Figure 9: Mapping of demand points

Examining all demand points in isolation, it can be concluded that among all the demand points determined by Algorithm 5, $G_s = (q_s, t_s) = (q_i, t_i) : \min_i(t_i - q_i)$ specifies the maximal feasible server initial delay of $t_s - q_s$.

For a longer server initial delay than $(t_s - q_s)$, the server could not supply sufficient processing time to satisfy the requirement of the demand point G_s. Therefore the demand point G_s (see Equation 3.5) determines the maximal initial delay of a server.

$$G_s = (q_s, t_s) = (q_i, t_i) : \min_i(t_i - q_i) \tag{3.5}$$

□

Given the demand point G_s, we know by Lemma 1 that the server utilisation monotonically increases for server period values $\Pi \geqslant (t_s + q_s)/2$. Considering only the demand point G_s, the smallest server period upper bound can be calculated as $\Pi = (t_s + q_s)/2$. Since the temporal server parameters are only defined in the integer domain, the initial server period value Π_s derived for demand point G_s needs to be converted into an integer value by applying the floor function (see Equations 3.6). The corresponding initial server capacity Θ_s is equal to q_s (see Equations 3.7) because the maximal initial server supply, specified as $t_s - q_s$, necessitates a server capacity supply that is equal to the demand.

$$\Pi_s = \left\lfloor \frac{t_s + q_s}{2} \right\rfloor \tag{3.6}$$

$$\Theta_s = q_s \tag{3.7}$$

However, we are interested in the largest possible server period and the corresponding capacity (Θ_u, Π_u) that might enable us the calculation of the optimal server parameters. A further constraint for the server parameters (Θ_u, Π_u) is that the corresponding server capacity supply satisfies all demand points. Theorem 1 provides the essential means for an algorithm to determine a server period upper bound for bandwidth optimal server.

Theorem 1. (Server period upper bound) *The server period upper bound is $\Pi_u = \Pi_s + \iota_s$ and the corresponding minimal server capacity $\Theta_u = \Theta_s + \iota_s$, with ι_s being a positive integer number such that for all demand points G_i the condition $s(t_i) \geqslant q_i$ holds.*

Proof. Lemma 2 provided $(q_s, t_s) = (q_i, t_i) : \min_i(t_i - q_i)$, which also determines the largest possible server initial delay, i.e. $\Delta = t_s - q_s$. Lemma 1 showed that for $\Pi \geqslant (t_s + q_s)/2$ the server utilisation $U(\Pi)$ monotonically increases with the server period Π. Assuming $\Pi = \Pi_s + \iota_s$, with ι_s being an arbitrary positive integer value, we can alternatively write the aforementioned inequality for Π as:

$$\Pi = \Pi_s + \iota_s \geqslant \Pi_s = \left\lfloor \frac{t_s + q_s}{2} \right\rfloor$$

The increment ι_s of the initial server parameters (Θ_s, Π_s) can be determined by first calculating the required increase for the server capacity (Θ_s) in order to meet all task demand requirements.

We note that in further steps, references to *all demand points* imply the exclusion of $G_s = (q_s, t_s)$ from the set of demand points. This represents an optimisation step since the server parameters (Θ_s, Π_s) were determined such that the resulting server capacity supply satisfies the demand requirement of the demand point G_s.

In order to determine the value ι_s by which the initial server capacity and period has to be increased, we first map the demand points onto the X-axis, followed by the calculation of the number of server replenishment periods.

A simple geometric transformation applied to the demand points (see Equation 3.8 and Figure 9) maps each demand point (q_i, t_i) onto a point $(t_i - q_i)$ on the X-axis.

$$G'_i = \begin{pmatrix} t'_i \\ q'_i \end{pmatrix} = \begin{pmatrix} 1 & -1 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} t_i \\ q_i \end{pmatrix} = \begin{pmatrix} t_i - q_i \\ 0 \end{pmatrix} \tag{3.8}$$

Applying the same geometric transformation to the end points of the server supply, reveals that the distance between these points remains constant (i.e. $\delta = \Pi_s - \Theta_s$), due to a uniform increase of both server parameters. This mapping facilitates the calculation of the number of *full server capacity supply periods h_i* that a demand point G_i experiences, in the considered scenario. In contrast to the server replenishment period, a full server capacity supply period starts as the server begins supplying execution time capacity.

Following the transformation of each demand point G_i to G'_i, the number of complete server capacity supplies h_i up to the time instant t_i can be calculated as specified in Equation 3.9. The time interval taken into account to determine h_i is $[(\Pi_s - \Theta_s), t_i]$, i.e. excluding the inactive server time that originates from the server period starting before $t = 0$, (see Figure 8).

$$h_i = \left\lfloor \frac{(t_i - q_i) - (\Pi_s - \Theta_s)}{\Pi_s - \Theta_s} \right\rfloor \tag{3.9}$$

In order to maintain an optimal server utilisation, the condition $s(t_s) = q_s$ must remain true even after the server parameters are increased in order to satisfy the required higher demand of other demand points than G_s. From the maximum initial server delay Δ (determined by the demand point G_s) and the property of monotonic server utilisation increase derived in Lemma 1, it follows that only a uniform increase of the initial server parameters Θ_s and Π_s can be carried out without violating the demand requirement of G_s. The uniform increase of both server parameters can be considered as a scaling of the server supply function along the vector $(x, y) = (1, 1)$. Thus, Equation 3.9 can determine the number of server capacity supplies for each demand point before the final server parameter values are known.

Now we take a counter intuitive step and first calculate the minimal server capacity Θ_u for the server period Π_u that is yet to be determined. Θ_u is the smallest server capacity that is required to satisfy the condition $\forall i : s(t_i) \geqslant q_i$. Given the demand q_i and the number of server capacity supplies h_i for each demand point in the set $\mathbb{G}$, we can determine the required server capacity Θ_u as depicted in Equation 3.10.

$$\Theta_u = \max_i \left(\left\lceil \frac{q_i}{h_i} \right\rceil \right) \tag{3.10}$$

Knowing the server capacity Θ_u, the capacity increase ι_s over the server's initial value Θ_s can be easily determined (see Equation 3.11). The constraint that the scaling of the server supply function is performed only along the vector $(x, y) = (1, 1)$ implies that both server parameters have to be changed equally. Hence, the server capacity increase ι_s entails an increase of the initial server period Π_s by the same value (see Lemma 1). Finally, the server period upper bound can be calculated as shown in Equation 3.12.

$$\begin{aligned} \Theta_u &= \Theta_s + \iota_s \\ \iota_s &= \Theta_u - \Theta_s \end{aligned} \tag{3.11}$$

$$\Pi_u = \Pi_s + \iota_s \tag{3.12}$$

Increasing the initial server parameters (Θ_s, Π_s) by ι_s ensures that the server supply satisfies the demand requirement expressed by the set $\mathbb{G}$. $\square$

In summary, given Π_u and Θ_u we can conclude that the server utilisation $U(\Pi)$ monotonically increases for $\Pi \geqslant \Pi_u$.

Theorem 2. (Local minima at server period upper bound) *For a server utilisation $U < 1$, Π_u and Θ_u represent a local minimum on the optimal server utilisation graph.*

Proof. Lemma 1 and Theorem 1 prove that the server utilisation $U(\Pi)$ monotonically increases for $\Pi \geqslant \Pi_u$. In the following we show that by decreasing the server period, for $\Pi \leqslant \Pi_u$, the server utilisation $U(\Pi)$ again monotonically increases.

$\Theta_u = \Theta_s + \iota_s$ denotes the smallest possible server capacity for $\Pi_u = \Pi_s + \iota_s$. The server parameters (Θ_u, Π_u) cannot be further uniformly decreased by an arbitrary small positive integer value $\mu < \iota_s$, otherwise the solution for the server period upper bound and the corresponding capacity would be $\Pi_u = \Pi_s + \iota_s - \mu$ and $\Theta_u = \Theta_s + \iota_s - \mu$, respectively. Without violating the demand requirements, a reduction of the server period Π_u is still possible, since from the schedulability point of view $(\Pi_u - \mu)$ has two positive effects on the server supply:

1. the initial server supply delay $\Delta = 2\left((\Pi_u - \mu) - \Theta_u\right)$ decreases and,
2. the server's replenishment period is shortened, thus the server is still supplying the capacity Θ_u just in shorter periods.

Hence, the server utilisation $U(\Pi)$ increases also with $\Pi : (\Pi_u - \mu) < \Pi < \Pi_u$, proving that Π_u denotes the largest period contributing to a local minimum on the optimal server utilisation graph $U(\Pi)$. □

3.4.2 Server context switch included

Considering the definition of the critical instant and that server capacity is not consumed by the context switch shows that the analysis for scenarios with and without server context switch is identical. The equations (Equation 3.7, 3.6, 3.10 and 3.12) calculating the initial server parameter values (Θ_s, Π_s), and the consequent server period upper bound and the corresponding server capacity (Θ_u, Π_u) depend only on the demand points that are not influenced by the server context switch time C_0.

Hence, to determine the server period upper bound Π_u and the corresponding capacity Θ_u in the presence of server context switch overhead, the approach for the optimal server period upper bound determination (presented in Section 3.4.1) can be used without modification.

In the following theorem we show that in the presence of server context switch overhead, the optimal server period upper bound and the corresponding server capacity defined in the previous section still represent on the server utilisation graph the local minima with the largest server period.

Theorem 3. (Server period upper bound considering server context switch overhead) $\Pi_u : \Pi_u \geqslant \Theta_u + C_0$, *with* Θ_u *denoting the smallest possible server capacity for* Π_u*, is the largest server period resulting in a local minima on the server utilisation graph.*

Proof. We assume $\Pi_u : \Pi_u \geqslant \Theta_u + C_0$ with Θ_u denoting the smallest possible server capacity such that the associated taskset is schedulable. Furthermore, it is assumed that the server parameters (Θ_u, Π_u) were determined according to the approach described in Lemma 2 and Theorem 2. We show:

1. for all server period values greater than Π_u that the server utilisation increases monotonically and,
2. Π_u is a local minima.

Referring to Lemma 1 and Theorem 2 we know that given (Θ_u, Π_u), both server parameters can only be uniformly increased if the smallest possible server utilisation shall be maintained. An arbitrary positive integer increment is denoted by x in the following equation.

$$\frac{\Theta_u + C_0 + x}{\Pi_u + x} = \frac{\Theta_u + C_0}{\Pi_u} + \frac{x\,(\Pi_u - (\Theta_u + C_0))}{\Pi_u(\Pi_u + x)} > \frac{\Theta_u + C_0}{\Pi_u} \quad \text{for} \quad x, \Pi_u \in \mathbb{N} \tag{3.13}$$

Equation 3.13 proves the first case that the sever utilisation monotonically increases for server period values larger than Π_u.

In order to prove the second case, i.e. Π_u is a local minima, we assume that the server capacity cannot be further decreased and show that the server utilisation also increases if the server period is decreased by an arbitrary small integer $\mu < \Pi_u$.

$$\frac{\Theta_u + C_0}{\Pi_u - \mu} = \frac{\Theta_u + C_0}{\Pi_u} + \frac{\mu(\Theta_u + C_0)}{\Pi_u(\Pi_u - \mu)} > \frac{\Theta_u + C_0}{\Pi_u} \tag{3.14}$$

The proofs (Equations 3.13 and 3.14) showed in the presence of server context switch overhead that Π_u still represents a valid upper bound on the optimal server period value. We proved that the server utilisation has a local minima at Π_u and for server period values larger than Π_u, the server utilisation increases monotonically with the server period. □

3.4.3 Server period upper bound determination algorithm

Given a set of demand points, a simple 6-step algorithm to determine the server period upper bound for a bandwidth optimal periodic server is defined in the following. The analysis and equations presented in the previous sections are used to build up Algorithm 6. Due to the assumed worst-case server capacity supply and context switch occurrence (see Sections 3.1, Sections 3.4.2, and Figure 3), the calculation of the server period upper bound remains the same whether or not server context switch time is considered.

```
In  : Set of demand points $\mathbb{G}$
Out : Server period upper bound $\Pi_u$ and capacity $\Theta_u$
1 Determine the demand point leading to the longest feasible server initial
  delay (see Equation 3.5);
2 Determine the initial server parameters $(\Theta_s, \Pi_s)$ (see Equations 3.6
  and 3.7);
3 For each demand point $G_i$, calculate the number of server capacity
  supplies $h_i$ (see Equation 3.9);
4 Calculate the required server capacity $\Theta_u$ (see Equation 3.10);
5 Determine the initial server parameter increase $\iota_s$ (see Equation 3.11);
6 Calculate server period upper bound $\Pi_u = \Pi_s + \iota_s$ (see Equation 3.12);
```

Algorithm 6: Server period upper bound

Algorithm 6 requires as input a set of demand points that describe the demand of a given taskset. Such a set can be determined by Algorithm 5 where the cardinality of the resulting set is less than or equal to the number of tasks in the taskset or used priority levels. Since the run time of the server period upper bound calculation depends on the number of demand points in the provided set, it can be concluded that Algorithm 6 has $O(n)$ time complexity, where n denotes the number of tasks in the taskset. However, recall that obtaining the demand points, used as input to Algorithm 6, requires pseudo-polynomial time.

An example, demonstrating the application of Algorithm 6, will be given after the introduction of the server period lower bound.

3.4.4 Server period lower bound

This section provides the definition for a server period lower bound. The main focus is on the scenario that takes the server context switch overhead into account.

The calculation of the optimal server period lower bound is based on the information about the application utilisation and the server utilisation resulting from the upper bound parameter values.

Starting at the critical instant, the required minimum server utilisation to schedule the tasks can be calculated based on the set of demand points $\mathbb{G}$. First we show that the utilisation value determined by the demand bound function provides a value that is greater than or equal to the sum of the task utilisations (see Equation 3.15). Such a greater utilisation value will be advantageous for the calculation of the server period lower bound.

$$\frac{dbf(t)}{t} = \frac{\sum_i \left(\left\lceil \frac{t}{T_i} \right\rceil C_i \right)}{t} \geqslant \frac{\sum_i \left(\frac{t}{T_i} C_i \right)}{t} = \sum_i \frac{C_i}{T_i} \tag{3.15}$$

Eventually, the minimum server utilisation necessary to schedule the application tasks can be determined by the demand point with the highest demand q_i per time interval t_i (see Equation 3.16).

$$U_A = \max_i \left(\frac{dbf(t_i)}{t_i} \right) = \max_i \left(\frac{q_i}{t_i} \right) \tag{3.16}$$

According to the optimality definition in this thesis, the server parameters leading to the lowest utilisation of the server plus context switch overhead are considered to be the optimal server parameters. As we showed in Theorem 1, the server utilisation monotonically increases for $\Pi \geqslant \Pi_u$, hence the period of the optimal server (with the lowest utilisation) must be less than or equal to Π_u. Moreover, under consideration of server context switch overhead, the optimal server utilisation $(U_o + (C_0/\Pi_o))$ has to be less than or equal to the upper bound utilisation $(U_u + (C_0/\Pi_u))$ resulting from the server period upper bound Π_u and the corresponding minimal server capacity Θ_u. Additionally, the optimal server utilisation cannot be less than the application's utilisation U_A. Hence, the optimal server utilisation is bounded and it can be concluded that $U_A < U_o + (C_0/\Pi_o) \leqslant U_u + (C_0/\Pi_u)$.

Given the server's upper bound and application's utilisation, $U_u + (C_0/\Pi_u)$ and U_A, the maximum remaining processor utilisation for the server context switch overhead can be determined as $(C_0/\Pi_l) = U_u + (C_0/\Pi_u) - U_A$. Considering the expected maximum server context switch execution time C_0, a lower bound Π_l on the optimal server period can be derived as shown in Equation 3.17.

$$\begin{gathered}\frac{C_0}{\Pi_l} = U_u + \left(\frac{C_0}{\Pi_u}\right) - U_A = \left(\frac{\Theta_u + C_0}{\Pi_u}\right) - U_A \\ \Pi_l = \max\left(1, \left\lfloor \frac{C_0}{U_u + \frac{C_0}{\Pi_u} - U_A} \right\rfloor\right)\end{gathered} \tag{3.17}$$

In the presence of server context switch overhead, Theorem 4 provides the proof for the existence of a server period lower bound Π_l.

Theorem 4. (Optimal server period interval) *Given the server period upper bound* Π_u *and* $U_u \geqslant U_A$*, the optimal server period interval is lower bounded by* Π_l *and upper bounded by* Π_u *such that* $\Pi_l \leqslant \Pi_u$.

Proof. We will prove the existence of a valid server period interval by contradiction, assuming that $\Pi_l > \Pi_u$.

$$\begin{aligned}
\Pi_l &> \Pi_u \\
\frac{C_0}{U_u + \dfrac{C_0}{\Pi_u} - U_A} &> \Pi_u \\
C_0 &> \Pi_u \left(U_u + \frac{C_0}{\Pi_u} - U_A \right) \\
C_0 &> \Pi_u \left(\frac{\Theta_u + C_0}{\Pi_u} - U_A \right) \\
C_0 &> \Theta_u + C_0 - \Pi_u U_A \\
\Theta_u &< \Pi_u U_A \\
U_u &< U_A
\end{aligned}$$

Given the server upper bound parameters (Θ_u, Π_u), the schedulability of the associated application's taskset is ensured (see Theorem 1, and Equations 3.10 and 3.12). Assuming that the server period lower bound is greater than the upper bound, i.e. $\Pi_l > \Pi_u$, led to the inequality $U_u < U_A$ that contradicts the assumption of the theorem. Hence, $\Pi_l \leqslant \Pi_u$ must be true. □

As will be shown in the context of the optimal server parameter calculation (see Chapter 4) the initial server period lower bound can be further improved during the execution of the iterative algorithm in order to decrease the runtime and improve the efficiency of the iterative optimal server parameter calculation algorithm.

3.5 Example

Using a numerical example, we demonstrate in this section the calculation of the server period lower and upper bound. Algorithm 6, Equations 3.17, 3.10 and 3.12 are applied on the taskset defined in Section 3.3.1 Table 1 and Table 2.

The demand points (i.e. $G_1 = (400, 1300)$, $G_2 = (2000, 3900)$ and $G_3 = (4600, 6500)$) determined in Section 3.3.1 serve as input into Algorithm 6. For the server context switch overhead C_0, we assume a maximal execution time of 100 ticks.

Among the three given demand points, $G_1 = (400, 1300)$ satisfies the condition $\min_i(t_i - q_i)$. Hence, $G_s = G_1 = (q_s, t_s) = (400, 1300)$ leads to the initial server period $\Pi_s = 850$ and capacity $\Theta_s = 400$. Based on the remaining two demand points (G_2 and G_3) we determine the number of server supplies and obtain the values $h_2 = 3$ and $h_3 = 3$. Then the server capacity is calculated as $\Theta_u = \max\left(\lceil q_2/h_2 \rceil, \lceil q_3/h_3 \rceil\right) = \max(667, 1534) = 1534$. Given the maximum required server capacity Θ_u, the initial server parameter increase is calculated as $\iota_s = \Theta_u - \Theta_s = 1134$. Finally, the server period upper bound is determined as $\Pi_u = \Pi_s + \iota_s = 1984$.

The next step is to determine the server period lower bound Π_l with an assumed server context switch execution time of 100 ticks.

The server's upper bound utilisation including server context switch overhead can be determined as $U_u + (C_0/\Pi_u) = 1534/1984 + 100/1984 = 0.7732 + 0.0504 = 0.8236$. The application's maximum utilisation is $U_A = 4600/6500 = 0.7077$. Eventually, the server period lower bound Π_l can be calculated by Equation 3.17 as $\Pi_l = \max\left(1, \lfloor 100/\left(0.7732 + 0.0504 - 0.7077\right) \rfloor\right) = 862$.

For demonstration purposes the server period lower and upper bounds $[\Pi_l, \Pi_u] = [862, 1984]$ are also depicted in Figure 7.

Assuming a smaller context switch time of 20 ticks, has an impact only on the server period lower bound, whereas the server period upper bound value remains unaffected. Considering the lower context switch time, the server's upper bound utilisation is: $U_u + (C_0/\Pi_u) = 1534/1984 + 20/1984 = 0.7732 + 0.0101 = 0.7833$. Recalculating the server period lower bound gives: $\Pi_l = \max\left(1, \lfloor 20/\left(0.7732 + 0.0101 - 0.7077\right) \rfloor\right) = 264$. Thus, the optimal server period interval for 20 ticks context switch overhead changes to $[\Pi_l, \Pi_u] = [264, 1984]$

3.6 Evaluation

As a pre-stage to the evaluation of the optimal server parameter determination algorithm's efficiency, we investigate in this section the interval length that contains the optimal server period.

Since the measured data for each test scenario are obtained on the basis of multiple random values for the input variables, the results have a certain distribution. However, the results presented in the evaluation sections report on the average case (unless otherwise stated) in order to focus only on the effect that certain variables have on the evaluation results.

3.6.1 Test data generation

In order to evaluate the different algorithms presented in this thesis, random numbers with certain constraints are generated in order to supply test data for each test-case. The procedure and constraints for the random number generation are described in the following.

A single test run with specific test data representing an artificial application is referred to as a *test-case*. The results of the empirical evaluation were obtained by generating 10,000 random tasksets for various test-case scenarios. For each taskset, the task parameters were created by applying the generation of uniformly distributed utilisation values, proposed by Bini and Buttazzo [15]. For each utilisation value, a random task period in the interval $[10^4, 10^6]$ is generated. Given the task utilisation and period, the calculation of its assumed worst-case execution time is straight forward. In the evaluation, the CPU tick is used as the unit for all temporal parameters.

Test-cases are generated for each permutation of the following three variables:

- taskset utilisation (also referred to as *Initial Target Utilisation (ITU)*) set to 10%, 25%, 40%, 55% and 70% of a full speed processor utilisation,
- number of tasks per taskset equal to 5, 15, 25 and 35,
- server context switch overhead equal to 1, 10, 20, ..., 100 ticks.

The combination of the aforementioned test variables gives 220 different test-case scenarios with 10,000 taskset each, leading to 2,200,000 specific test-cases for the entire evaluation.

With respect to the test-case generation, the server context switch overhead cannot be defined in terms of processor utilisation until the server parameters for a certain test-case are determined. But in order to investigate how the server context switch execution time influences other variables, it was expressed as an absolute value.

The test data generation procedure described in this section will also be used in Section 4.5 of the following chapter to evaluate the iterative optimal server parameter selection algorithm. In order to avoid the repetition of the test data generation procedure, we will only refer back to this section without providing the same specification a second time.

3.6.2 Experiment[3] 1: Server context switch overhead

Experiment 1 examines the absolute execution time values of the server context switch expressed as processor utilisation.

In order to illustrate the processor utilisation consumed by the server context switch, the absolute execution time of the server context switch is put in relation to the optimal server period that is determined for each taskset. The average processor utilisation values for the corresponding absolute server context switch execution times are rendered in Figures 10–11.

The rendered data indicates that the mapping of the server context switch from the absolute execution time to the corresponding processor utilisation is nearly a linear transformation. The results also show that in the worst-case the server context switch time corresponds to approximately 1% of the server utilisation. However, the actual mapping depends on the taskset's Initial Target Utilisation and also slightly on the number of tasks in the taskset. It can be observed that the processor utilisation corresponding to a certain absolute context switch execution time increases with decreasing taskset ITU. The number of tasks in a given taskset has only a marginal effect on the mapping but it is noticeable

[3]The experiments in this thesis are consecutively numbered with a unique number assigned to each of them.

that with decreasing taskset size the processor utilisation corresponding to an absolute context switch execution time decreases as well.

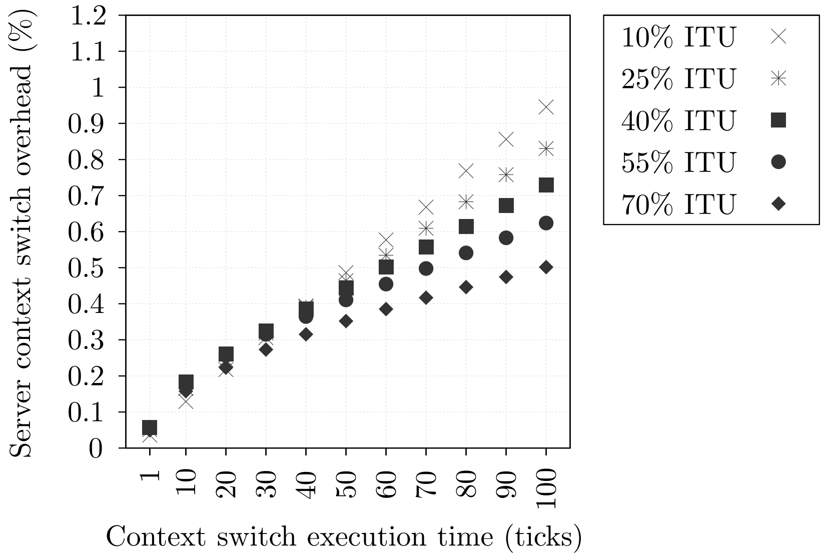

Figure 10: Processor utilisation consumed by server context switch (15 tasks)

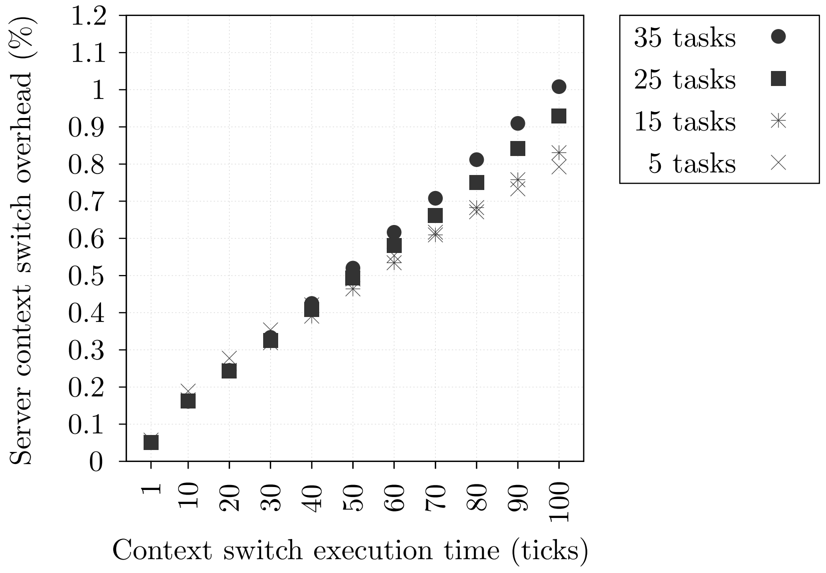

Figure 11: Processor utilisation consumed by server context switch (25% ITU)

3.6.3 Experiment 2: Optimal server period bounds

Experiment 2 is concerned with the optimal server period interval. The optimal server period interval, bounded by the lower and upper server period bound, $[\Pi_l, \Pi_u]$ is examined under different circumstances in order to capture the complexity of a non-analytic approach (e.g. brute-force search) for the optimal server period. The figures provide insight about the range of server period values that contains the optimal value.

Figures 12–16 show that the taskset's initial utilisation has an impact on the server period upper bound, whereas the lower bound is mainly influenced by the server context switch overhead. With increasing taskset utilisation, the value of ι_s, by which the initial server parameters (Θ_s, Π_s) are increased to obtain the upper bound value, increases as well. Consequently, the server period upper bound value gets larger and the interval of the optimal server period expands.

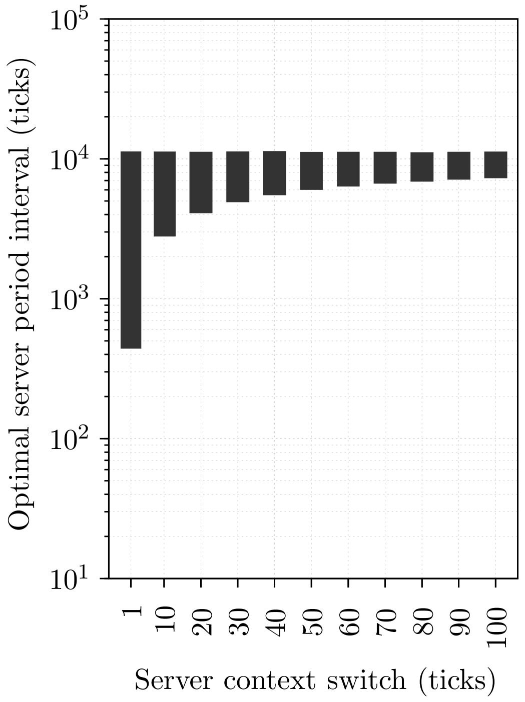

Figure 12: Optimal server period interval for 15 tasks and 10% ITU

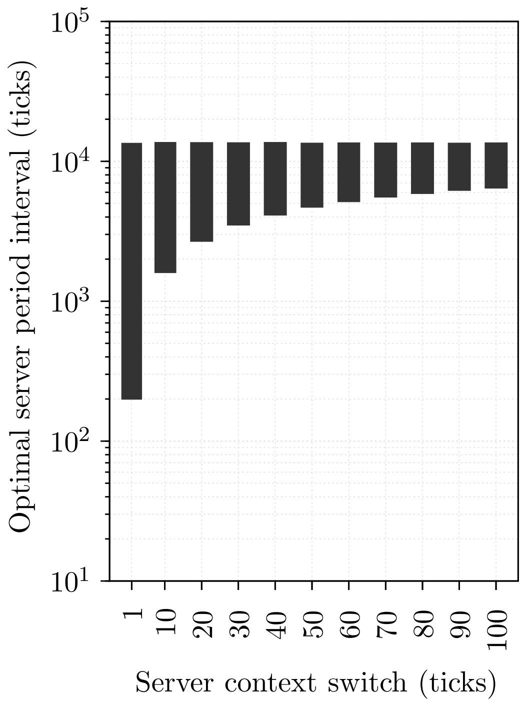

Figure 13: Optimal server period interval for 15 tasks and 25% ITU

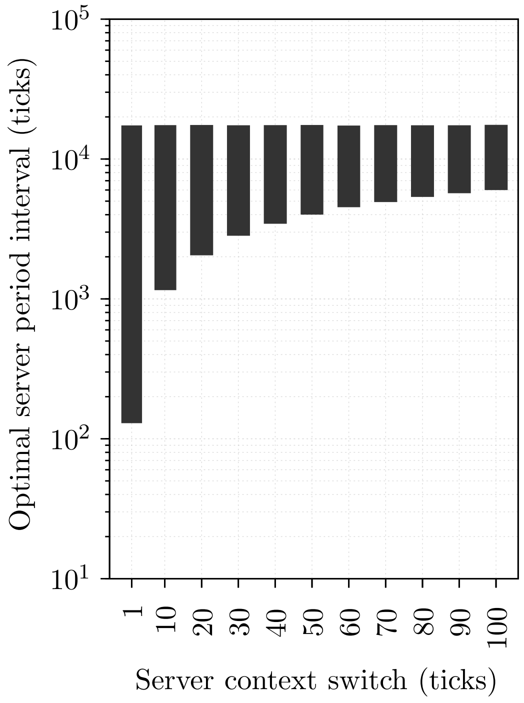

Figure 14: Optimal server period interval for 15 tasks and 40% ITU

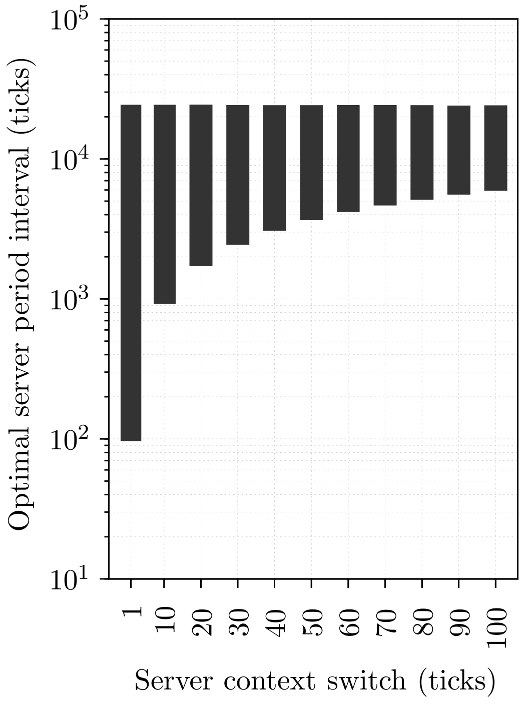

Figure 15: Optimal server period interval for 15 tasks and 55% ITU

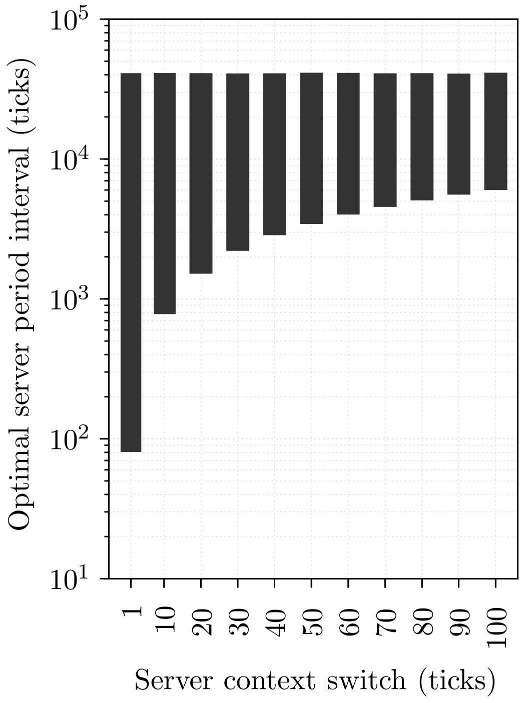

Figure 16: Optimal server period interval for 15 tasks and 70% ITU

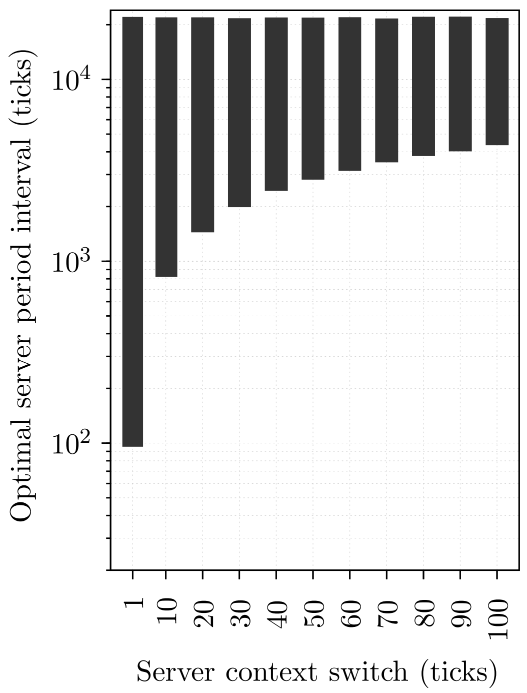

Figure 17: Optimal server period interval for 5 tasks and 25% ITU

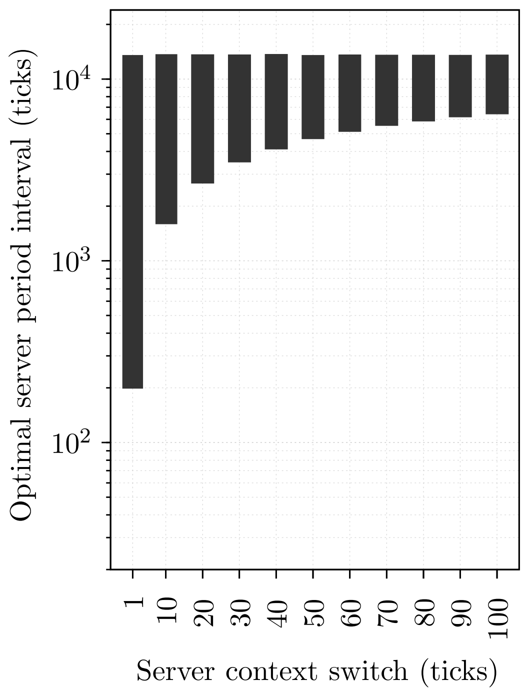

Figure 18: Optimal server period interval for 15 tasks and 25% ITU

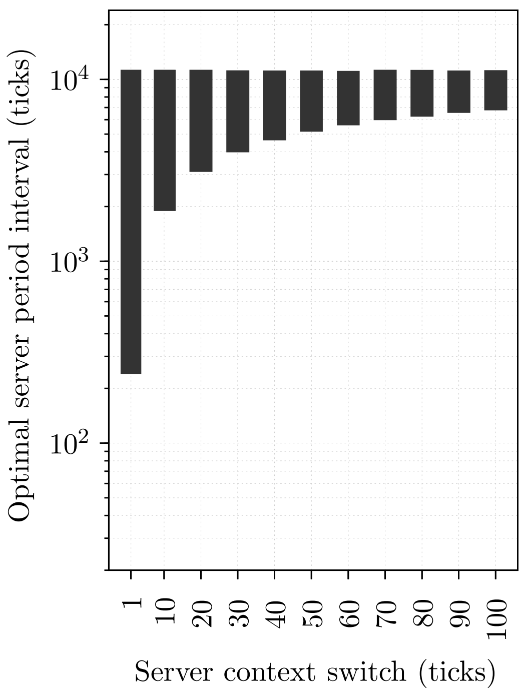

Figure 19: Optimal server period interval for 25 tasks and 25% ITU

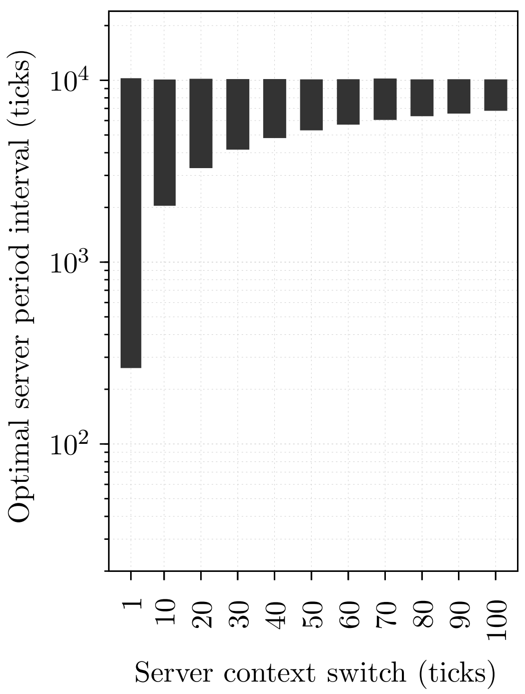

Figure 20: Optimal server period interval for 35 tasks and 25% ITU

The results in Figures 17–20 indicate that the increasing number of tasks in an application has a positive effect on the interval $[\Pi_l, \Pi_u]$ where the optimal server period value can be found. The increasing number of tasks reduces the server period upper bound and narrows the optimal server period interval.

The dependency of the server period lower bound on the context switch overhead can be explained by examining Equation 3.17. Considering that the server period upper bound remains nearly unchanged while the context switch time increases, we can conclude that the value of each term in Equation 3.17, except the server context time, remains for a certain test-case nearly the same. Hence, the largest effect on the server period lower bound is caused by the server context switch time in the numerator of the equation. That means, with increasing server context switch time the server period lower bound increases as well.

3.7 Summary

The main details of the assumed system model were briefly introduced in Chapter 2 by analysing previously published research. Building upon the previous research, in this chapter the system model for flexible open real-time systems was formalised, providing support for the variability of multiple task and server temporal parameter values.

Based on the specified system model, an approach to represent the taskset demand by *demand points* was presented. Since demand points also express the minimal required processing capacity for a taskset, they can be used to ensure the schedulability of the corresponding tasks. The taskset's schedulability can be examined by verifying that the processing capacity supply of a specific server satisfies the demand expressed by the demand points.

With respect to the optimal server parameter selection, the detailed examination and analysis of the minimal server utilisation for a schedulable taskset highlighted the complexity of the problem. The optimal server utilisation graph rendered as a function of the server period revealed a sawtooth shape. The conclusion was that a tailored approach is required in order to provide an efficient method for the optimal server parameter selection.

Our chosen approach first focuses on the localisation of an optimal server period interval that can be used to determine the actual optimal server parameters by further processing. This work was motivated by the need for an analytic method to determine the optimal server period interval.

Equations and algorithms were presented to autonomously determine the bounds of the optimal server period. In contrast to the autonomous approach, previous methods required that certain input values or constraints are specified by the engineers (a detailed analysis of this issue was provided in Section 2.7).

In summary it can be stated that we provided the necessary analysis for a given taskset in order to determine the optimal server period lower and upper bound for an associated server. Determining the optimal server period lower and upper bound provides a significant isolation of possible optimal period values. Nevertheless, a brute-force search over such an interval would still be very time consuming due to the number of possible optimal server period candidates as shown by the evaluation. Therefore the following chapter is concerned with the development of more efficient methods to determine on the one hand sufficiently good and on the other hand optimal server parameters.

Chapter

4

Optimal server period

The selection of server parameters resulting in a minimal processor resource reservation for a schedulable taskset, also expressed as minimal server utilisation, was the focus of many previous research efforts [6, 28, 39–41, 64, 65, 75]. However, the emphasis of previously presented server parameter selection algorithms was on the computational complexity (by introducing server supply function linearisation), the algorithm's accuracy (by providing exact schedulability conditions for given taskset and server parameters) or a trade-off between these two parameters.

One of the previous methods [41] for EDF scheduled tasksets provides an efficient and configurable approach, by trading the run-time of the algorithm with the accuracy of the approximated server parameters regarding the optimal values. Concerning the accuracy of the selected server parameter values and run-time complexity, the approximation approach is established between the methods based on the server supply linearisation and the optimal server parameter selection approach. In contrast to its original definition, the approximation algorithm [41] can, however, be further optimised in order to reduce its computational complexity.

Therefore, first the optimised approximation method, based on Fisher's approach [41] is defined in this chapter. Subsequently, the main contribution of this chapter, the introduction and definition of the iterative optimal server parameter selection algorithm is presented. Based on the optimal server period interval, an algorithm is devised to determine the optimal server period value. In order to specify the iterative algorithm, the information about the optimal server period upper bound and the sawtooth shaped minimal server utilisation graph are exploited.

Following the definition of the optimised approximation method and the iterative algorithm, the flexibility of adaptive tasks is considered in the server parameter selection.

Finally, the trade-off between processor utilisation loss and run-time effort between a fairly good but approximate method and our iterative optimal parameter selection approach is examined in the evaluation section of this chapter.

4.1 Simple server parameter approximation algorithm

As a reference for an approximate server parameter selection algorithm, in the evaluation section we will use the configurable approach that was introduced by Fisher [41]. Based on his initial algorithm, we present in this section an optimisation step for the server parameter approximation algorithm and define a slightly simpler approach to determine the server's execution capacity and period values.

The objective of both approximation algorithms, the one specified in this section and the other specified by Fisher [41], is to determine the server period and capacity parameters with the property that the resulting server utilisation is not larger than $(1 + \epsilon)$ times the optimal value. In order to determine the server parameters with the aforementioned property, a predefined optimal server period interval, limited by a lower and upper bound, is supplied and utilised by the algorithm.

However, the deficiency of the algorithm presented in [41] is that it examines more server parameter values than necessary in order to determine the server capacity and period pair with a utilisation less than or equal to $(1+\epsilon)$ times the optimal server utilisation.

Fisher's iterative algorithm performs a bisection on a given optimal server period interval and determines for each period value the corresponding server capacity. For the next iteration of the algorithm, the lower bound of the optimal server period interval is updated with the period value for which the server capacity value is less than or equal $(1+\epsilon)$ times the server capacity value of the previous iteration.

In summary, to determine the server parameters, with a resulting server utilisation less than or equal to $(1+\epsilon)$ time the bandwidth optimal server utilisation, binary search is used to select the largest feasible server period over an interval that is limited by a given server period lower and upper bound. The largest feasible server period selected from the server period interval is the one for which the aforementioned constraint of two successive server capacity values is still maintained. That means, in Fisher's algorithm the server capacity is used to control the recalculation of server parameters. The algorithm determines server parameters in a way such that the capacity ratio between two successive choices of server capacity value is not larger than $(1+\epsilon)$. However, while the algorithm advances, after each iteration only the lower bound of the optimal server period interval is increased but the upper bound remains fixed. Hence, during various iterations of the algorithm, larger server period values especially towards the upper bound of the server period interval are examined multiple times if the accuracy parameter ϵ is very small.

In the following, a simpler algorithm (Algorithm 7), inspired by Fisher's approach [41], is presented. The simpler algorithm also ensures that the server utilisation for a selected server period and capacity is at most a factor $(1+\epsilon)$ larger than the optimal server utilisation.

The central idea behind Algorithm 7 is that the server's minimal capacity monotonically increases with the server period [41] while the schedulability of the associated taskset is maintained. To the contrary, this means that decreasing the server period cannot result in an increased server capacity.

As previously mentioned, the objective of Fisher's approximation algorithms is to select server parameters that have a utilisation not larger than $(1+\epsilon)$ times the optimal server utilisation. Based on these facts a simpler approach than in [41] can be defined. Starting with the upper bound Π_u a geometric sequence of period values can be created. Subsequent server period values are obtained by dividing the previous server period by $(1+\epsilon)$ until the lower bound Π_l is reached. For each generated server period value, the corresponding minimum capacity is calculated and the parameters with the minimum server utilisation are stored.

The interval that is upper and lower bounded by Π_u and Π_l (Section 3.4) must contain the optimal server period. The geometric sequence of server periods generated within the aforementioned optimal server period interval guarantees that the server period, which is at most a factor of $(1+\epsilon)$ smaller than the optimal value, will be examined. Hence, a server with a period equal to a value in the geometric sequence is guaranteed to schedule the tasks if it has the same capacity as the optimal server. Furthermore, since the server capacity is monotonically non-decreasing with the server period, the minimum capacity for the period checked is at most the same as the optimal server. This means the utilisation of the best server parameters found is at most $(1+\epsilon)$ times the optimal utilisation.

In contrast to Fisher's algorithm where certain value ranges of the optimal server period interval (especially towards the server period upper bound) are examined multiple times, the simpler approximation algorithm presented in Algorithm 7 examines only $ln(\Pi_u/\Pi_l)/ln(1+\epsilon)$ server period values. Therefore, Algorithm 7 can faster approximate the near optimal server parameter values.

The algorithm starts with the initialisation of the prospective best server parameter choice (Θ', Π'), the calculation of the corresponding server utilisation $\hat{U}'$ including the server context switch time, and the assignment of a start value for the examined server period variable Π_{probe} (see lines 1–3). The main loop starts at line 4 and controls the evaluation of server period values until the optimal server period lower bound has been reached. For a schedulable taskset and a given server period value (Π_{probe}) the corresponding minimal capacity (Θ_{probe}) is calculated in line 5. Under consideration of the server context switch overhead, in line 6 the utilisation $(\hat{U}_{probe})$ of the new server parameters is determined. If the utilisation of the server with the new parameter values is smaller than for any previously examined server parameter value (line 9), then the new parameter values are stored as the currently best server parameter choice (lines 10–12). For the server period the subsequent value in a geometric sequence is calculated in line 14 by dividing the last server period value by $(1+\epsilon)$. However, since server parameters are assumed to be integer values, line 14 might not result in a changed probe value if the accuracy parameter (ϵ) is very small. Therefore,

```
In : Server period lower bound ($\Pi_l$), server upper bound parameters
     ($\Theta_u, \Pi_u$), server context switch time ($C_0$), accuracy parameter ($\epsilon$)
     and the associated taskset
1  $(\Theta', \Pi') = (\Theta_u, \Pi_u)$;
2  $\hat{U}' = (\Theta' + C_0)/\Pi'$;
3  $\Pi_{probe} = \Pi_u$;
4  while $\Pi_{probe} \geqslant \Pi_l$ do
5      Determine server capacity $\Theta_{probe}$ for $\Pi_{probe}$;
6      $\hat{U}_{probe} = (\Theta_{probe} + C_0)/(\Pi_{probe})$;
7      if $\hat{U}_{probe} > 1.0$ then
8          return 'Unschedulable taskset';
9      end
10     if $\hat{U}_{probe} < \hat{U}'$ then
11         $(\Theta', \Pi') = (\Theta_{probe}, \Pi_{probe})$;
12         $\hat{U}' = \hat{U}_{probe}$;
13         Recalculate the server period lower bound $\Pi_l$ based on $\Pi'$;
14     end
15     $\Pi_{temp} = \Pi_{probe}$;
16     $\Pi_{probe} = \lceil(\Pi_{temp})/(1+\epsilon)\rceil$;
17     if $\Pi_{probe}$ equals $\Pi_{temp}$ then
18         $\Pi_{probe} = \Pi_{probe} - 1$;
19     end
20 end
21 return $(\Theta', \Pi')$;
```

Algorithm 7: Simple approximate server parameter determination algorithm

in lines 15–16 it is ensured that in such a case the server period value of the last iteration is still decremented and all possible server period values down to the lower bound (Π_l) are examined.

The complexity of Fisher's and the previously presented approximation algorithm originates from the number of server capacity values that need to be calculated during the runtime of the algorithms. With the objective that the utilisation of the final server parameters is at most $(1 + \epsilon)$ times greater than or equal to the optimal server utilisation, our approximation method has to determine the server capacity for $ln(\Pi_u/\Pi_l)/ln(1+\epsilon)$ number of server period values. Assuming the same accuracy criteria, Fisher's algorithm has to determine $ln(\Theta_u/\Theta_l) \cdot (ln(\Pi_u)/\epsilon)$ [41] number of server capacity values, with Θ_u denoting the corresponding server capacity for Π_u and Θ_l for Π_l.

4.2 Iterative optimal server parameter selection

The general conditions to derive an efficient optimal server parameter selection algorithm were introduced in the previous chapter. We build upon the presented algorithms and equations from Chapter 3 that are concerned with the calculation of the lower and upper bound of the optimal server period value, and derive in this section a new iterative algorithm for server parameter calculation. This improved iterative approach determines the exact optimal server period by evaluating only certain values within the interval bounded by a lower bound Π_l and an upper bound Π_u. As a result, the majority of the possible values that would have been considered by a brute-force search over the aforementioned interval will be discarded, hence significantly reducing the search space for the optimal server period value.

Starting at the server period upper bound Π_u, we are interested in a server period value that may result in a smaller server utilisation than the one obtained for Π_u. Therefore, in the iterative algorithm the server period is successively decreased, starting with Π_u.

Rationale for the operations of the iterative optimal server parameter determination algorithm is based on Fisher's observation [41] that all server capacity determination algorithms are monotonically non-decreasing over the server period. That means, as the server period increases the server capacity either remains constant or increases with the period. We utilise this statement by considering the opposite case, i.e. as the server period decreases, the server capacity either remains constant or decreases with the period.

Informally, the following observation provides the basis of the introduced iterative algorithm. Given a server period upper bound value with the corresponding minimal server capacity, such that the associated taskset is still schedulable, the following two operations can be carried out provided that the minimal server utilisation is still maintained:

1. either only the server period can be decreased while the server capacity is kept constant, or
2. both, the server period and capacity can be decreased by the same value.

The two aforementioned operations form the sawtooth shape of the optimal server utilisation graph and provide also the explanation for it. If only the server period is decreased then the resulting server utilisation increases. On the other hand, if both the server period and the capacity is uniformly decreased, the server utilisation decreases as well. These properties of the server parameters are formally described in the following theorem. In fact, Theorem 5 specifies for a given server capacity and period the next possible value and the effect of the performed server parameter change on the minimal server utilisation.

Theorem 5. (Server parameter changes) *Maintaining the minimal server utilisation for a schedulable taskset, the next possible value of the corresponding server capacity* $\Theta \in \mathbb{N}$ *and server period* $\Pi \in \mathbb{N}$*, and the resulting effect on the server utilisation is defined as:*

$$\begin{array}{l}\Theta \rightarrow \Theta \\ \Pi \rightarrow \Pi - 1\end{array} \quad \Rightarrow \quad \frac{\Theta}{\Pi - 1} > \frac{\Theta}{\Pi} \tag{4.1}$$

$$\begin{array}{l}\Theta \rightarrow \Theta - 1 \\ \Pi \rightarrow \Pi - 1\end{array} \quad \Rightarrow \quad \frac{\Theta - 1}{\Pi - 1} < \frac{\Theta}{\Pi} \tag{4.2}$$

Proof. First we prove the case that the server utilisation increases if only the server period is decreased by 1 (Equation 4.1). We recall that the server parameters are defined in the integer domain and therefore the next smallest value is determined by subtracting 1 from the current value. We assume $\Pi > 1$, a fixed server capacity Θ and $0 < \Theta < \Pi$.

$$\begin{gathered}\Pi - 1 < \Pi \\ \frac{1}{\Pi - 1} > \frac{1}{\Pi} \\ \frac{\Theta}{\Pi - 1} > \frac{\Theta}{\Pi}\end{gathered}$$

Next, we prove the case that the server utilisation decreases if both server parameter values are decreased by 1 (Equation 4.2). We assume $\Pi > 1$ and $0 < \Theta < \Pi$, and prove that by decreasing the server parameters the resulting server utilisation increases.

$$\begin{aligned}
&\Pi - 1 < \Pi \\
&\frac{1}{\Pi - 1} > \frac{1}{\Pi} \quad \text{multiplying LHS with } \Theta\text{-1 and RHS with } \Theta \\
&\frac{\Theta - 1}{\Pi - 1} = \frac{\Theta}{\Pi} - \frac{\Pi - \Theta}{(\Pi - 1)\Pi} < \frac{\Theta}{\Pi}
\end{aligned}$$

$\square$

Starting from the optimal server period upper bound Π_u, successively performing the two operations introduced in Theorem 5, the peak and trough points on the minimal server utilisation graph can be determined. The following sections utilise Theorem 5 to specify a two staged iterative algorithm for the optimal server parameter calculation.

4.2.1 Stage 1

The first of the two possible operations of the iterative approach, the reduction of the server period while the capacity is kept constant, results in an increase of the server utilisation.

The objective of the server period reduction is to determine a peak point on the minimal server utilisation graph. Such a peak point is reached when the server period is reduced by the smallest value that enables further uniform decrease of both server parameters. This implies that only those demand points in the set $\mathbb{G}$ need to be considered that prevent further uniform decrease of both server parameters.

An efficient way to determine which demand points prevent the uniform decrease of both server parameters is to calculate the difference between the server supply $s(t_i)$ and the demand q_i for the demand point G_i. Assuming the domain of integer numbers for the server parameters, for the aforementioned demand points the inequality $s(t_i) - q_i < h_i$ holds.

Theorem 6. (Demand points prevent server parameter decrease) *A demand point G_i prevents the uniform decrease of both server parameters if $s(t_i)-q_i < h_i$. The set of demand point satisfying this condition will be denoted as $\mathbb{G}'$.*

Proof. In order to decrease the server parameters, the condition $s(t_i) > q_i$ must hold. However, in the integer domain, to enable the decrease of the server parameters, the difference between $s(t_i)$ and q_i has to be at least equal to or greater than the number of full server capacity supply periods h_i. If the difference $s(t_i) - q_i$ is less than h_i then the maximal possible decrement for the server parameters would be less than 1. In the integer domain, however, a value less than 1 is not meaningful as a decrement for server parameters. Hence, it can be concluded that demand points, satisfying the condition $s(t_i) - q_i < h_i$, prevent the uniform decrease of the server capacity and period. □

The inverse of the condition $s(t_i) - q_i < h_i$ can be used to determine the number of server replenishment periods that a demand point G_i must experience in order to enable the uniform decrease of the both server parameters. In this scenario, we will use κ_i instead of h_i to denote the required number of server replenishment periods. Therefore, the inequality $s(t_i) - q_i \geqslant \kappa_i$, which is the inverse of $s(t_i) - q_i < h_i$, expresses after how many server replenishment periods κ_i is it possible to equally decrease the server capacity and period. Based on this inequality, the equation for κ_i can be derived (see Equation 4.3).

$$\begin{aligned}
\kappa_i\Theta - \kappa_i &\geqslant q_i \\
\kappa_i\left(\Theta - 1\right) &\geqslant q_i \\
\kappa_i &\geqslant \frac{q_i}{\Theta - 1} \\
\kappa_i &= \left\lceil \frac{q_i}{\Theta - 1} \right\rceil \quad \text{for} \quad \Theta > 1
\end{aligned} \tag{4.3}$$

Equation 4.3 is defined for server capacity values greater than 1. Yet, this constraint is not obstructive since if the server capacity is equal to 1 in the integer domain then it is the smallest valid value and it cannot be further decreased. Therefore, the server utilisation could just increase if the server period is further decreased, but it would not yield to further optimal parameter candidates. Thus,

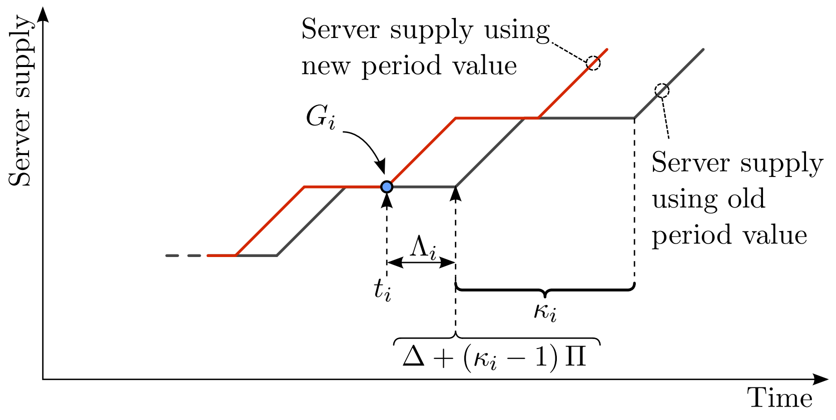
Server supply
Server supply using new period value
G_i
Server supply using old period value
Λ_i
t_i
κ_i
$\Delta + (\kappa_i - 1)\,\Pi$
Time

Figure 21: Server period decrement illustration

no further examination of server parameters is required and the search for optimal server parameters can be terminated.

The objective of this stage of the algorithm is to decrease the server period value so that the demand point $G_i \in \mathbb{G}'$ corresponds with the instant right before the start of the κ_i-th server supply (see Figure 21).

Hence, the equation to determine the accumulated surplus server period value (Λ_i, see Equation 4.4) can be easily derived. The instant right before the start of the κ_i-th server supply corresponds to the sum of the server's initial delay Δ and $(\kappa_i - 1)$ times the server period Π.

$$\Lambda_i = \Delta + (\kappa_i - 1)\,\Pi - t_i \tag{4.4}$$

However, since the server parameter values are defined in the integer domain, it is required to extend Equation 4.4. Using integer server parameters values, the demand points will only infrequently lie on the server supply function. Thus, the demand q_i specified by any demand point in the set $\mathbb{G}'$ may lie in the range $(\kappa_i - 1)\,\Theta \geqslant q_i \geqslant ((\kappa_i - 1)\,\Theta) - \kappa_i$ and still prevent the uniform decrease of the server parameters (see Figure 22).

Therefore, in the calculation of Λ_i the difference γ_i between the smaller demand q_i and the supplied full server capacity $(\kappa_i - 1)\,\Theta$ needs to be considered as well. This inaccuracy originating from the integer domain results in a smaller decrement that is required in order to enable the uniform decrement of both

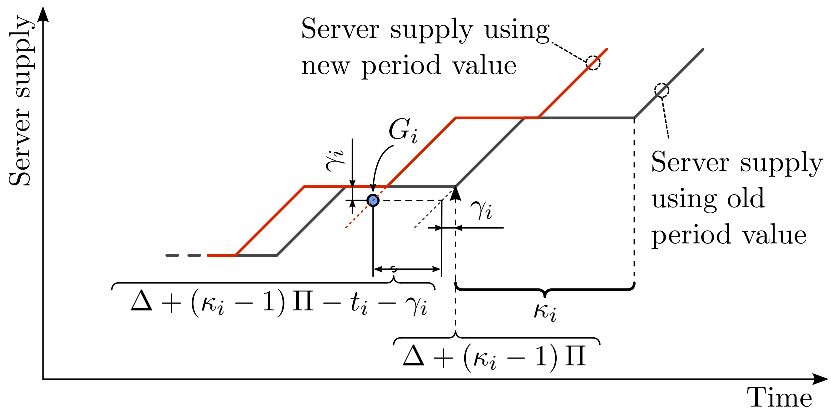
Server supply
Server supply using new period value
G_i
γ_i
γ_i
Server supply using old period value
$\Delta + (\kappa_i - 1)\,\Pi - t_i - \gamma_i$
κ_i
$\Delta + (\kappa_i - 1)\,\Pi$
Time

Figure 22: Server period decrement illustration with the integer domain

server parameters (see Figure 22). The extension of Equation 4.4 considering the inaccuracy originating from the integer domain is presented in Equation 4.5.

$$\begin{aligned}\Lambda_i &= \Delta + (\kappa_i - 1)\,\Pi - t_i - ((\kappa_i - 1)\,\Theta - q_i)\\ \Lambda_i &= \Delta + (\kappa_i - 1)\,\Pi - t_i - \gamma_i\end{aligned} \tag{4.5}$$

Considering only the demand point $G_i \in \mathbb{G}'$ and the determined Λ_i, the actual server period decrement ϱ_i can be calculated by Equation 4.6. The equation is based, on the rationale that by decreasing the server period and keeping the capacity constant, only the server's inactive time $(\Pi - \Theta)$ is reduced. Furthermore, it is also based on the observation that before the start of κ_i-th server supply, there are $(\kappa_i + 1)$ inactive server times (two of them during the initial delay Δ and $(\kappa_i - 1)$ before the start of κ_i-th server supply). Hence, Λ_i has to be divided by $\kappa_i + 1$ in order to ensure that the time instant of the demand point G_i and the end of the $(\kappa_i + 1)$-th inactive server time $(\Pi - \Theta)$ coincide.

$$\varrho_i = \frac{\Lambda_i}{(\kappa_i + 1)} \tag{4.6}$$

Finally, to determine the new server period value Π', the largest decrement ϱ_i that was calculated for the demand points in the set $\mathbb{G}'$ needs to be subtracted from the current server period value Π. Given the new server parameters (Θ', Π') the proceeding uniform reduction of server capacity and period can be carried out again without rendering the corresponding taskset unschedulable. The equations to determine the new server parameters, basically where the server capacity remains constants, is depicted in Equation 4.7.

$$\begin{aligned} \Pi' &= \Pi - \max_i (\varrho_i) \\ \Theta' &= \Theta \end{aligned} \tag{4.7}$$

The server period decrement is determined for each demand point, thus the runtime of *Stage-1* of the iterative algorithm depends on the number of demand points. As shown in Section 3.3.1, the number of demand points that is required to represent a fixed-priority scheduled taskset's demand, is less than or equal to the number of tasks in the taskset. Hence, the runtime of *Stage-1* of the iterative algorithm linearly depends on the number of tasks in a given taskset. Consequently, the time complexity of this stage of the algorithm is $O(n)$ with n denoting the number of tasks.

4.2.2 Stage 2

The second stage of the iterative algorithm is concerned with finding a trough point on the minimal server utilisation graph.

The effect of this operation on the server utilisation graph has already been proved in Equation 4.2 of Theorem 5. We showed that a uniform decrease of both, the server period and capacity, entails a monotonic decrease of the server utilisation.

Both server parameters can be decreased as long as the associated taskset is schedulable. The largest decrement of the server parameters results in a trough point on the minimal server utilisation graph. The trough point is reached since after the maximal uniform decrement of both server parameters, temporarily only the server period can be further reduced.

The approach that was taken to determine the server period upper bound, can be used in a slightly modified way to calculate the value by which both server parameters can be uniformly decreased. In the case of the server period upper bound calculation, the maximal required increment for both server parameters was sought. In the case of the iterative algorithm, we are interested in the maximal possible decrement for both server parameters.

For every demand point G_i in the set $\mathbb{G}$ we determine the decrement value φ_i for both server parameters such that the server supply for G_i is just sufficient. In order to obtain φ_i for G_i, the difference between the demand q_i and the server capacity supply after the prospective number of replenishment periods h_i is calculated. Analogous to the approach presented for the server period upper bound calculation, the number of prospective server replenishment periods h_i can be determined as defined by Equation 3.9.

Hence, the server capacity and period decrement φ_i, such that the server supply is still sufficient to satisfy the demand that is specified by the demand point G_i, can be calculated by Equation 4.8.

$$\varphi_i = \left\lfloor \frac{h_i\Theta - q_i}{h_i} \right\rfloor \tag{4.8}$$

The denominator represents the difference between the demand q_i and the server capacity supply after h_i replenishment periods. Since this value represents the cumulative difference over h_i server periods, it has to be divided by h_i in order to obtain the actual decrement φ_i for the server parameters.

From the calculated decrement values $\{\varphi_0, \ldots, \varphi_i\}$, the smallest value has to be chosen as the server capacity and period decrement in order to maintain the schedulability of the taskset that is executed on the server. Finally, the new server parameters based on the smallest uniform decrement $\min_i(\varphi_i)$ can be determined by Equation 4.9.

$$\begin{aligned} \Pi' &= \Pi - \min_i(\varphi_i) \\ \Theta' &= \Theta - \min_i(\varphi_i) \end{aligned} \tag{4.9}$$

Similarly to *Stage-1*, *Stage-2* of the iterative algorithm also depends on the number of demand points that need to be examined. Based on the rationale in Section 4.2.1 it can be concluded that the time complexity of *Stage-2* is also $O(n)$.

4.2.3 Iterative algorithm

In this section, the iterative algorithm to determine the optimal server parameters is presented (see Algorithm 8). It is based on the two aforementioned operations (described in *Stage-1* and *Stage-2*) that are carried out in the search for the optimal server parameters.

As an initial step (line 1 in Algorithm 8), the iterative algorithm determines the server period upper bound value Π_u and the corresponding server capacity Θ_u by Equation 3.12 and Equation 3.10, respectively. The algorithms starts with the upper bound value since it is the largest possible value for the optimal server period Π_u as was shown in Section 3.4.

In line 2, the prospective optimal server parameters are initialised with the server period upper bound value and the corresponding capacity.

Next, the main loop of the algorithm (line 3) is carried out, iterating over various server period values, as long as the predetermined server period lower bound is not reached. The main loop implements the previously described two stages of the algorithm. The successive execution of the two stages is bounded by the server period lower and upper bound, Π_l and Π_u, respectively.

Starting from the server period upper bound Π_u, only the server period can be further reduced. If it was possible to reduce both server parameters, then the server period upper bound algorithm would have determined a smaller Π_u. Hence, *Stage-1* of the algorithm is performed, as described in Section 4.2.1 (lines 4–6).

After decreasing only the server period in *Stage-1*, the newly determined server parameters may represent 100% server utilisation. Therefore, it is not meaningful to execute *Stage-2* of the algorithm after this situation occurred. Once the server utilisation including context switch time reaches 100%, implying that $\Theta + C_0 = \Pi$, the operations neither of *Stage-1* nor of *Stage-2* can lead anymore to a server capacity such that $\Theta + C_0 < \Pi$. That means, the optimal server period was examined and stored during previous iterations. Hence, the case with 100% server utilisation is detected in lines 7–8 of Algorithm 8 and the loop is aborted.

However, if the current server parameters do not result in 100% utilisation, then *Stage-2* of the algorithm, as described in Section 4.2.2, is executed (lines 11-13).

```
   In  : Set of demand points $\mathbb{G}$, server context switch overhead $C_0$
   Out : Bandwidth optimal server parameters $(\Theta_o, \Pi_o)$
1  Determine the optimal server period lower bound $\Pi_l$, upper bound $\Pi_u$
   and the capacity $\Theta_u$ (see Equations 3.12, 4.11 and 3.10;
2  Use the server period upper bound $\Pi_u$ and the corresponding capacity
   $\Theta_u$ as initial values for the current and optimal server parameters,
   $(\Theta', \Pi')$ and $(\Theta_o, \Pi_o)$, respectively;
3  while $\Pi' > \Pi_l$ and $\Theta' > 1$ do
       Algorithm Stage-1
4      foreach Demand point in $\mathbb{G}$ do
5          Determine the possible server period decrement $\varrho_i$;
6      end
7      Decrease current server period $\Pi'$ by $\max_i(\varrho_i)$;
8      if Server capacity + $C_0$ is equal to the server period then
9          Abort while-loop;
10     end
       Algorithm Stage-2
11     foreach Demand point in $\mathbb{G}$ do
12         Determine the possible server capacity and period decrement $\varphi_i$;
13     end
14     Decrease current server capacity $\Theta'$ and period $\Pi'$ by $\min_i(\varphi_i)$;
15     if Server utilisation using parameters $(\Theta', \Pi')$ is less than with
       $(\Theta_o, \Pi_o)$ then
16         Save current server parameters as potential optimal values,
           $(\Theta_o, \Pi_o) = (\Theta', \Pi')$;
17         Recalculate server period lower bound, $\Pi_l$, using the updated
           potential optimal values;
18     end
19 end
20 Return bandwidth optimal server parameters $(\Theta_o, \Pi_o)$;
```

Algorithm 8: Bandwidth optimal server parameter calculation

At the end of the loop, taking into account the server context switch time, the utilisation of the current server parameters (Θ', Π') is calculated and compared to the utilisation of the prospective optimal server parameters (Θ_o, Π_o). If the newly calculated server parameters (Θ', Π') have a lower utilisation than any previous server parameter pair, then they are stored as the new prospective optimal server parameters (Θ_o, Π_o). In this case, the server period lower bound Π_l is also recalculated since a tighter bound can be obtained for the newly determined prospective optimal server parameter values. The rationale that server parameter values with a lower server utilisation than previously determined values also increase the server period lower bound Π_l will be provided in the subsequent section (Section 4.2.4). As a consequence of the improved lower bound Π_l, the interval for the optimal server period is narrowed, requiring less computation to determine the optimal server parameter values.

Finally, when the main loop is completed due to reaching either 100% server utilisation or the server period lower bound, the server parameter values that were last considered as optimal, are returned as the bandwidth optimal server period and capacity.

In summary, Algorithm 8 consists of two stages. During the first stage of the iterative approach, the next local peak point on the server utilisation graph is found while the server period is decreased. Whereas during the second stage both server parameters are equally reduced leading to the next local trough point on the server utilisation graph.

With respect to the complexity of the iterative algorithm, the *while*-loop (lines 3–19) has the main influence on the runtime. As shown in Section 4.2.1 and Section 4.2.2 the time complexity of *Stage-1* and *Stage-2* is $O(n)$, with n representing the number of tasks. However, the *while*-loop depends on the optimal server period bounds, which in turn depend on the taskset parameters. Based on this condition, it is shown in Theorem 7 that the iterative algorithm has pseudo-polynomial time complexity.

Theorem 7. (Iterative algorithm time complexity) *The iterative algorithm (Algorithm 8) has pseudo-polynomial time complexity.*

Proof. As previously noted, the *while*-loop in Algorithm 8 has the main impact on the iterative algorithm's runtime complexity. In order to prove that the algorithm runs in pseudo-polynomial time, we show that the optimal server period interval is bounded by the longest task deadline.

In Section 3.4, we showed that the optimal server period upper bound Π_u specifies the maximal length of the optimal server period interval, whereas Π_u implicitly depends on the longest task deadline $\max_i(D_i)$.

According to the definition of Algorithm 5, the time instants, used to determine the demand points, are bounded by the longest task deadline $\max_i(D_i)$. That means, $\max_j(t_j) \leqslant \max_i(D_i)$. Furthermore, due to the monotonic increasing task demand function, the demand q_{max} associated with $t_{max} = \max_j(t_j)$ denotes the largest demand in $\mathbb{G}$. Setting the server capacity to q_{max} and with the use of Lemma 2, which limits the maximal initial server supply delay, we can compile the following inequality.

$$\begin{aligned} \Pi + (\Pi - \Theta) &\leqslant t_{max} \\ 2\Pi - q_{max} &\leqslant t_{max} \quad \text{with} \quad \Theta = q_{max} \\ \Pi &\leqslant \frac{t_{max} + q_{max}}{2} \end{aligned}$$

In the worst-case, for a taskset with 100% utilisation, $t_{max} = q_{max} = \max_i(D_i) = \Pi$. Thus, the time complexity of the entire algorithm is pseudo-polynomial. □

4.2.4 Continuously improving the replenishment period lower bound

As indicated, the optimal server period lower bound introduced in Section 3.4.4 can, however, be further improved in order to reduce the interval of the optimal server period. A tighter lower bound Π_l for the optimal server period can be calculated if the optimal server utilisation $(U_o + (C_0/\Pi_o))$ is strictly less than the upper bound utilisation $(U_u + (C_0/\Pi_u))$.

The central idea is to reduce the maximum possible processor utilisation that might be available for the server context switch overhead (C_0/Π_l). The reduction of the server context switch utilisation can be accomplished if a feasible server utilisation value less than $(U_u + (C_0/\Pi_u))$ is found. The optimal server utilisation including the server context switch overhead $(U_o + (C_0/\Pi_o))$ represents such a value. Hence, in this revised scenario, the maximum possible processor utilisation of the server context switch overhead is calculated as the difference between the optimal server utilisation including the context switch overhead $(U_o + (C_0/\Pi_o))$ and the application utilisation U_A.

The approach to derive the improved server period lower bound is based on Equation 3.17. Lemma 3 and the corresponding proof show that a lower server utilisation value results in a larger value for the server period lower bound. This fact is advantageous since a larger value for the server period lower bound implies a reduction of the optimal server period interval. As a consequence, the search space and the effort required to determine the optimal server period value, is reduced.

Lemma 3. *In the optimal server period interval, if for two server utilisation values applies that* $U_z + (C_0/\Pi_z) < U_w + (C_0/\Pi_w)$ *then* $\Pi_z > \Pi_w$.

Proof. Assume that $U_o + (C_0/\Pi_o) \leqslant U_z + (C_0/\Pi_z) < U_w + (C_0/\Pi_w) \leqslant U_u + (C_0/\Pi_u)$, then

$$\begin{aligned}
U_z + \frac{C_0}{\Pi_z} &< U_w + \frac{C_0}{\Pi_w} \\
U_z + \frac{C_0}{\Pi_z} - U_A &< U_w + \frac{C_0}{\Pi_w} - U_A \\
\frac{C_0}{\Pi_z} &< \frac{C_0}{\Pi_w} \\
\frac{1}{\Pi_z} &< \frac{1}{\Pi_w} \\
\Pi_z &> \Pi_w
\end{aligned} \tag{4.10}$$

□

Finally, the improved equation for the calculation of a tighter server period lower bound (see Equation 4.11) can be derived analogous to Equation 3.17.

$$\begin{aligned}
\frac{C_0}{\Pi_l} &= U_o + \left(\frac{C_0}{\Pi_o}\right) - U_A = \left(\frac{\Theta_o + C_0}{\Pi_o}\right) - U_A \\
\Pi_l &= \max\left(1, \left\lfloor \frac{C_0}{U_o + \frac{C_0}{\Pi_o} - U_A} \right\rfloor\right)
\end{aligned} \tag{4.11}$$

The examination of Equation 3.17 and Equation 4.11 shows that the two equations differ only in the terms that represent the upper bound and optimal server utilisation with context switch overhead included.

Thus, Equation 4.11 can be easily integrated into the iterative algorithm for optimal server parameter determination. The initial server period lower bound Π_l is calculated (line 1 in Algorithm 8) using the optimal server period upper bound value Π_u since the upper bound value and the corresponding execution capacity represent potential optimal server parameters. In subsequent iterations of the algorithm (line 17 in Algorithm 8), the server period lower bound is recalculated each time new prospective optimal server parameter values are found resulting in a tighter optimal server period interval.

4.3 Example

In Section 3.5 we demonstrated the calculation of the optimal server period lower and upper bounds based on an exemplary taskset defined in Table 1. The corresponding optimal server period interval was determined as $[\Pi_l, \Pi_u] = [862, 1984]$.

Next we present the application of the iterative algorithm on the same taskset and use the demand points determined and presented in Table 2 to demonstrate the algorithm's operation and to determine the optimal server parameters. The assumed server context switch time C_0 is still 100 ticks.

In *Stage-1* the demand point G_3 is detected as the one which prevents the uniform decrease of both server parameters. ϱ_3, as presented in Equation 4.6, is determined to be 70 ticks. The prospective optimal server period (initialised with $\Pi_o = 1984$) is then decreased by ϱ_3 and the newly obtained server parameters are $(\Theta', \Pi') = (1534, 1914)$.

Followed by *Stage-2*, the maximal possible decrement φ_i for each demand point is determined as specified by Equation 4.8. The smallest value, i.e. $\varphi_3 = 384$, among all these decrement values is chosen to uniformly decrease the server parameters. The new server parameters are $(\Theta', \Pi') = (1150, 1530)$.

These two stages of the algorithm are successively executed until the main loop terminates. During the execution of the main loop the algorithm detects that the server parameter values $(1150, 1530)$ result in smaller server utilisation than any previous parameter set and stores these values as new prospective optimal server parameters. Due to the change of the prospective optimal server parameter values, the server period lower bound is also recalculated. The improved server lower bound is $\Pi_l = \max\left(1, \lfloor 100/\left(0.7516 + 0.0654 - 0.7077\right)\rfloor\right) = 914$.

After the server period value of 895 has been reached, the algorithm determines that the server period lower bound has been under-run and terminates the main loop. The last set of prospective optimal server parameters, i.e. $(1150, 1530)$, will become the optimal parameters.

Table 3: Iterative algorithm progress

Algorithm stage	Parameter decrement	Server parameters (Θ, Π)	Utilisation point type	Server utilisation including $C_0 = 100$
Upper bound		(1534,1984)	Trough	0,8236
1	max(ϱ_3=70)			
		(1534,1914)	Peak	0,8537
2	min(φ_1=1134, φ_2=1034, φ_3=384)			
		(1150,1530)	Trough	0,8170
1	max(ϱ_3=64)			
		(1150,1466)	Peak	0,8527
2	min(φ_1=750, φ_2=750, φ_3=230)			
		(920,1236)	Trough	0,8252
1	max(ϱ_3=45)			
		(920,1191)	Peak	0,8564
2	min(φ_1=720, φ_2=586, φ_3=153)			
		(767,1038)	Trough	0,8353
1	max(ϱ_3=34)			
		(767,1004)	Peak	0,8635
2	min(φ_1=567, φ_2=481, φ_3=109)			
		(658,895)	Trough	0,8469

In this example we can also observe that the optimal server period is not necessarily smaller than the shortest task deadline or period, i.e. $\Pi_o = 1530$ and $\min(D_i) = 1300$ (Table 1).

For the sake of completeness, the values calculated by the algorithm between the server period upper and lower bound are also presented in Table 3.

4.4 Flexible server parameter selection

As a last step of the offline optimal server parameter selection, we take into account and exploit the flexibility of tasks.

The server parameter selection for applications consisting of flexible tasks is divided into two approaches according to the classification of its tasks. As introduced in the system model (see Section 2.4.3) application tasksets are constrained to tasks either with fixed and continuous, or with fixed and discrete temporal parameter specifications. Based on these configurations, the server associated with one of the aforementioned sets of tasks is also classified either as continuous or discrete.

4.4.1 Continuous servers

An application containing tasks with continuous parameters can provide a valid result after a specified minimum processing time. These tasks can, however, improve the quality of the produced result if the processing time per time unit available to them is increased. This can be achieved either by enabling a longer execution time or by the reduction of the task period and deadline. Nevertheless, there is a lower limit on the task's deadline and period, and an upper limit on the worst-case execution time where the maximum quality of the produced result is achieved. Hence, it does not provide any further benefit for the system's QoS going beyond these limits.

Although, the optimal server parameter selection algorithm (Algorithm 8 specified in Section 4.2) is only applicable on a taskset with fixed temporal requirements, a minor extension enables the applicability of the algorithm to tasksets with flexible temporal specifications. In order to apply Algorithm 8 on a flexible application with continuous tasks, the optimal temporal parameter calculation is carried out in two steps for continuous servers:

1. In the first step, for each task τ_i in the given taskset, the task's minimum temporal requirement is selected, i.e. the shortest worst-case execution time $C_{i(\min)}$, and the longest period $T_{i(\max)}$. Based on these task parameter values a temporary taskset is created. This taskset, containing the minimum temporal requirements of the tasks, is supplied into Algorithm 8. The server parameter values, obtained by the algorithm, are then set as the minimum execution capacity $\Theta_{x(\min)}$ and the maximum period $\Pi_{x(\max)}$ of the associated server S_x, reflecting resource reservation for a taskset's minimal resource requirements.
2. During the second step, a new temporary taskset is created. However, this time the maximum temporal requirements are chosen, i.e. the longest possible worst-case execution time $C_{i(\max)}$ and the shortest period $T_{i(\min)}$. The server parameter values obtained by Algorithm 8 are considered as the maximum execution capacity $\Theta_{x(\max)}$ and the minimum replenishment period $\Pi_{x(\min)}$ of the associated server S_x, reflecting resource reservation for a taskset's maximal resource requirements.

Given a flexible application taskset with continuous tasks, the temporal parameters for a continuous server can be determined using the aforementioned two-step approach.

In the case that tasks with fixed parameters coexist along with continuous tasks, they are incorporated into the taskset definition by setting their minimum and maximum temporal parameters to the same value (i.e. $C_{i(\min)} = C_{i(\max)}$ and $T_{i(\min)} = T_{i(\max)}$). The resulting taskset consisting of continuous tasks definitions can be then supplied into the two-step approach to determine the optimal continuous server parameter values.

4.4.2 Discrete servers

The second type of flexible tasks considered in the system model belong to the category of discrete tasks.

It is assumed that applications having different modes of operation are defined by discrete tasks. The various discrete task parameter definitions represent the temporal parameter values for the associated modes of operation within the corresponding application. Based on the application's active mode of operation, the appropriate version of each task's temporal parameter definition becomes active [84]. That means, a discrete task has for each of its application's mode a distinct temporal parameter value specification, implying an association of discrete task parameter values to modes of operations.

We note that the application's modes might not be associated to the modes of the entire system. They rather reflect application states or reactions to events received by the application.

Since the taskset parameters are fixed in each mode of operation, a similar approach as presented in Section 4.4.1 can be used to determine the optimal server parameters. In contrast, however, the creation of the temporary tasksets supplied into Algorithm 8 is based on different application modes.

For each mode m of an application, the corresponding parameter values $\big(C_{i(m)}, T_{i(m)}, D_{i(m)}\big)$ of each task τ_i are used to create a temporary taskset.

Similarly to continuous tasks, it is also possible to define tasks with fixed parameters using the discrete task model. This can be accomplished by setting the temporal parameters of a fixed task to the same value in each mode (i.e. $\forall_{x,y \in modes} : \big(C_{i(x)}, T_{i(x)}, D_{i(x)}\big) = \big(C_{i(y)}, T_{i(y)}, D_{i(y)}\big)$).

The server's temporal parameters (Θ_m, Π_m) determined by Algorithm 8 represent the values for a specific application's mode of operation. Executing Algorithm 8 for each temporary taskset creates a set of parameter values for a discrete server, providing temporal partitioning for the different modes of operation of the associated application.

Using this approach, the created discrete server will have the same number of (Θ_m, Π_m) tuples in the set of possible server parameters values as the number of the associated application's modes of operations.

4.5 Evaluation

In the following section the performance of the iterative method is examined with respect to the execution time and the processor utilisation loss that is obtained by using Almeida's and Pedreiras' [6] method, Fisher's [41] configurable approximation approach and a similar but simpler approximation algorithm defined in Section 4.1 of this chapter. Finally, the quality of the server upper bound parameters is compared to the optimal server period.

In order to enable a comparative evaluation, Fisher's [41] and the simpler approximation algorithm were extended by a cost function for server context switch overhead, similar to the cost function proposed in [64] and [6]. Specifically, all server utilisation calculations in both approximation algorithms have been extended by the utilisation of the context switch overhead, i.e. $U_x = (\Theta_x/\Pi_x) + (C_0/\Pi_x)$. With this extension, the bandwidth optimal server parameters are selected under consideration of the server context switch cost. Furthermore, a server period lower and upper bound is required by both approximation algorithms. In the experiments, we used our approach to calculate the optimal server period bounds and provided them as input for the approximation algorithms.

In addition, Fisher's [41] and the simpler approximation algorithm depend on an unspecified capacity determination algorithm. The only requirement is that the capacity determination algorithm calculates for a given server period value the minimum server capacity value that is required for a feasible schedule of the associated taskset. In the experiments we integrated a simple binary search algorithm to determine such a server capacity value.

In summary, the evaluation results show the trade off between the accuracy of server period values, the performance of calculating these values and the processor utilisation reserved for the servers.

The linear approach, introduced by Lipari and Bini [64, 65], is not considered in the evaluation since it does not provide limits either for the server period or capacity. Similarly, the algorithm proposed by Easwaran et al. [38] is also left out of consideration since it does not define limits for the server period either. Hence, a brute-force search would have to be applied by these methods over the optimal server period interval. Consequently, the performance comparison of these methods to the iterative method is not meaningful.

In order to enhance the readability of some figures in the evaluation section, the legends were abbreviated at certain places. The abbreviations used in the figures for the various server parameter determination algorithms are explained in Table 4.

The test data generation used as input for the different experiments was described in Section 3.6.1.

Table 4: List of abbreviations for server parameter determination algorithms

Abbreviation	Description
Iter_d	Measured data for the iterative algorithm with the taskset's temporal requirements specified by demand points. Both, the iterative algorithm and demand points were introduced in this chapter.
Fish_a	Measured data for Fisher's approximation algorithm using the accuracy parameter set to the largest value, i.e. $\epsilon = 1.0$.
Simp_a	Measured data for the simple approximation algorithm using the accuracy parameter set to the largest value, i.e. $\epsilon = 1.0$.
Alme_e	Measured data for Almeida's and Pedreiras' server parameter determination algorithm using external points to specify the taskset's temporal requirements.
Iter_e	Measured data for the iterative algorithm with the taskset's temporal requirements specified by external points.

4.5.1 Experiment 3: A qualitative comparison of proposed methods for server parameter calculation

In experiment 3 the following five methods, to determine optimal and non-optimal but sufficiently good server parameter values, are compared with each other:

- Our iterative algorithm utilising the demand points. With this setup, the optimal server parameter values are determined and the resulting optimal server utilisation is used as a reference value.

- Fisher's algorithm [41] with the accuracy parameter $\epsilon = 1.0$. Using this setup the algorithm determines the server parameters with least accuracy, but the runtime of the algorithm is the shortest among all values of ϵ. The measurements show how far the server utilisation for approximated server parameters deviate from the optimal value.
- The simple approximation algorithm with the accuracy parameter set to $\epsilon = 1.0$. Analogous to the previous setup, the server utilisation deviation for server parameters determined by the simple approximation algorithm is put in relation to the optimal value.
- Almeida's and Pedreiras' algorithm [6] using a linear server model and external points. With this setup we investigate the effect of the linear server model and external points on the server utilisation in contrast to the optimal value.
- Our iterative algorithm, but using the external points as presented in [6]. We examine the server utilisation overhead caused by external points in contrast to demand points. In other words, the server parameters obtained for the external points are compared to the optimal values since the demand points result in optimal server parameters.

The resulting server utilisation that is obtained by the application of various server parameter calculation methods is examined in Figures 23–27.

Tests with Almeida's and Pedreiras' algorithm and the dedicated use of external points in the iterative approach were carried out only up to 40% Initial Target Utilisation of the taskset. This decision was based on the fact that schedulable tasksets requiring 100% of the processor time started to occur and become more frequent at and above 55% ITU when external points were used to express the taskset's temporal requirements. The actual problem is the proposed approach for the calculation of external points [6]. External points are specified as the taskset demand at task deadline points. The demand at a task's deadline might be higher than at an earlier time instant, since new tasks might have been released just before the deadline. Hence, this additional demand has to be taken into account for the external points and it might lead to demand requirements that are above 100% server utilisation. But since in the experiments only schedulable tasksets have been considered, we can conclude that even if the external points dictate server parameters that are infeasible, the taskset could be executed on the processor if it receives 100% processor time. Due to this anomaly of the external points at and above 55% ITU only Fisher's and the simple approximation algorithm have been further examined.

The server utilisation graph obtained for external points shows the disadvantage of working with the demand information that is calculated at the task deadline instants. The utilisation of the resulting server is a few percent higher if the server parameters are determined based on the demand information specified by external points in contrast to demand points as proposed in this thesis.

The measured data obtained for the iterative approach with the external points as input data also facilitates the quantitative comparison between the exact server model used in this thesis and the linear model applied by Almeida and Pedreiras [6]. It can be observed that the application of the linear server model also results in higher server utilisation than what is possible by using an exact server supply model.

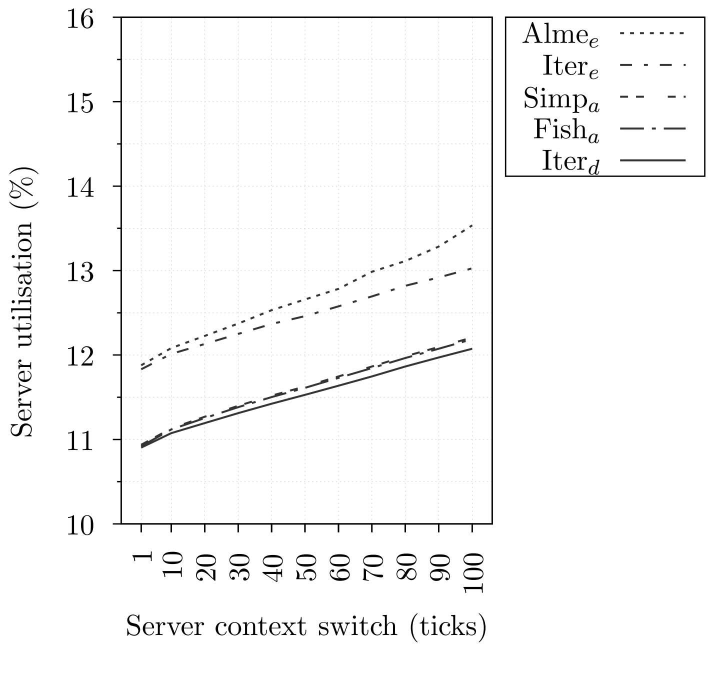

Figure 23: Quality of various server parameter selection approaches for 15 tasks and 10% ITU

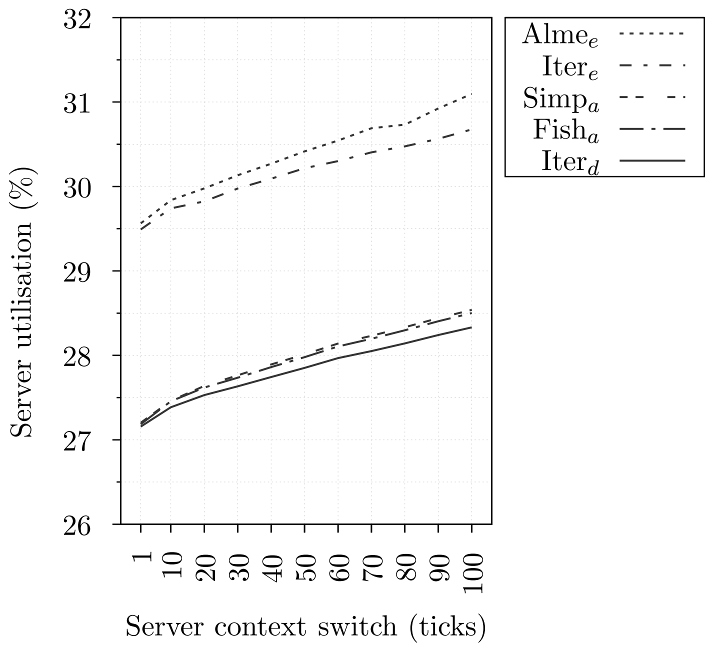

Figure 24: Quality of various server parameter selection approaches for 15 tasks and 25% ITU

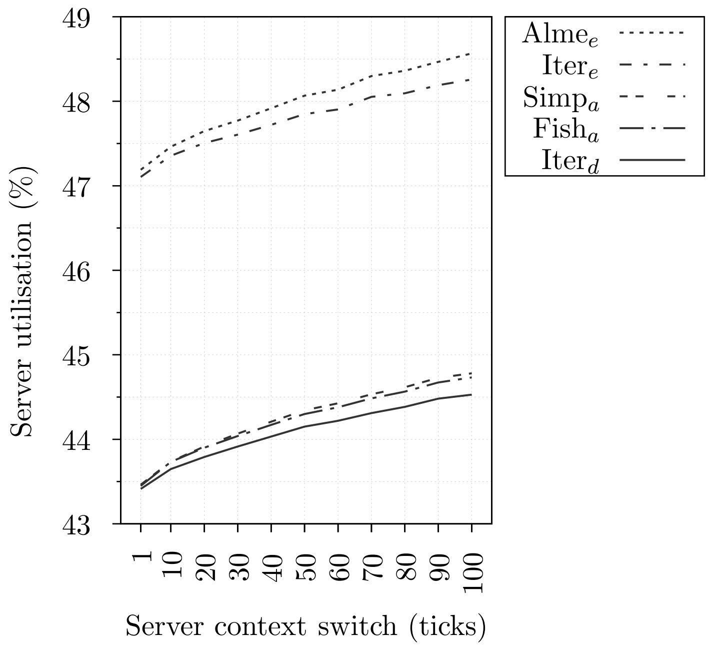

Figure 25: Quality of various server parameter selection approaches for 15 tasks and 40% ITU

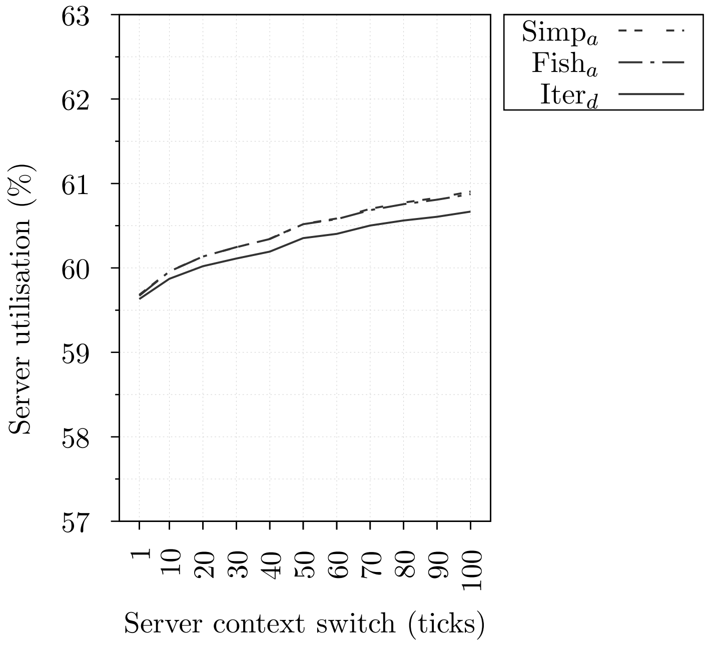

Figure 26: Quality of various server parameter selection approaches for 15 tasks and 55% ITU

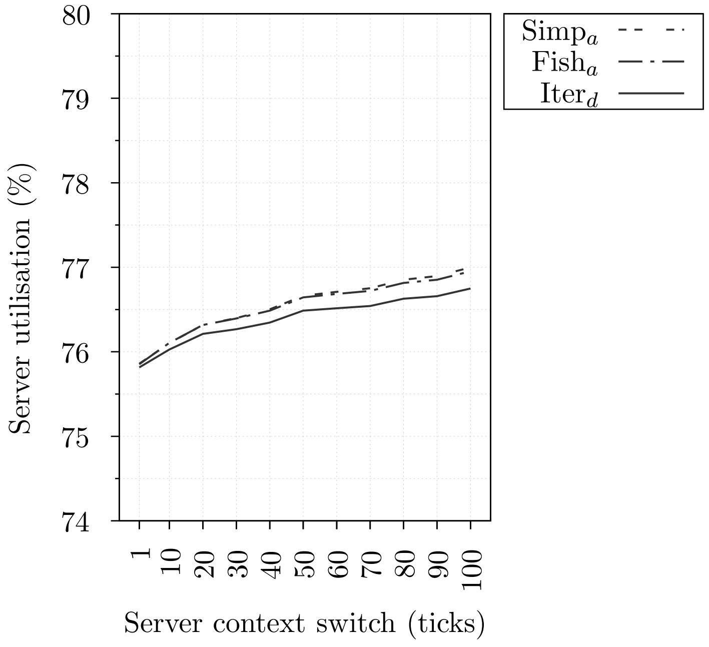

Figure 27: Quality of various server parameter selection approaches for 15 tasks and 70% ITU

In summary, the graphs in Figures 23–27 show that the server utilisation is the highest if the server parameters are determined by Almeida's and Pedreiras' algorithm [6] using a linear server model and external points. The second highest server utilisation is obtained for server parameters that are determined by the iterative method using an exact periodic server supply model but utilising the external points proposed by Almeida and Pedreiras. Among the examined approaches in this experiment, the best near-optimal server utilisation is achieved by Fisher's [41] and the simple approximation algorithm (Section 4.1). Since both methods are based on the same idea, the resulting graphs are nearly overlapping. The server parameters calculated based on the external points result in a few percent (2%–3%) higher server utilisation in contrast to the optimal values that are based on demand points.

Since the graphs in Figures 23–27 show the average server utilisation, the utilisation difference between the server with parameters determined by the approximation methods using $\epsilon = 1.0$ and the optimal server, is only a few percentage. Nevertheless, according to the definition of the approximation algorithms, in the worst-case the resulting server utilisation may be up to 100% higher than for the optimal server if $\epsilon = 1.0$ is used.

4.5.2 Experiment 4: A performance comparison of server parameter selection methods

The absolute execution time of the various algorithms to determine the optimal and non-optimal but sufficiently good server period values is examined in this experiment.

Among the algorithms examined in this experiment, Fisher's and our simple approximation algorithm determine the server parameters with a configurable, user specified, accuracy. That means, the approximated near optimal server parameters, determined by the two aforementioned algorithms, do not result in a server utilisation overhead larger than specified by the accuracy parameter ϵ. For comparative purposes, we set the accuracy parameter ϵ to 10^{-2}, which means the server utilisation of the calculated near optimal parameters is at most 1% higher than the optimal server utilisation. The impact of the accuracy parameter on the performance of these approximation algorithms will be further examined in a separate experiment.

The execution times of the different algorithms are expressed as the number of executed x86 assembly instructions. The results were obtained by utilising the CPU's hardware performance counters via the *Performance API (PAPI)*[4]. The number of executed assembly instructions that were monitored by the hardware performance counter for each algorithm are depicted in Figures 28–32 and Figures 33–36.

Although the quality of the server parameters determined by Almeida's and Pedreiras' algorithm had the largest deviation from the optimal values (see Section 4.5.1), the performance of this algorithm is predominant for small server context switch times and small tasksets. Furthermore, the performance of the algorithm depends only on the number of tasks and remains independent of the taskset ITU and server context switch time.

The performance of the simpler approximation methods configured with $\epsilon = 1.0$ is superior to the iterative algorithm. However, with $\epsilon = 1.0$ the approximation algorithm is usually not able to determine the optimal server parameters. Fisher's implementation of the approximation algorithm, also configured with $\epsilon = 1.0$, exposes a similar performance as the iterative algorithm. In particular, the iterative algorithm determines the optimal server parameter values in contrast to the approximation method that calculates near-optimal values.

The evaluation showed that if optimal server parameters are required for the system design, then the iterative algorithm presented in this chapter is superior in comparison to the brute-force search over the optimal server period interval and the approximation algorithms using a small ϵ. The execution time of the iterative algorithm to find the optimal server parameter value is significantly smaller than with previously presented approaches.

[4]http://icl.cs.utk.edu/papi/ (20 July 2011)

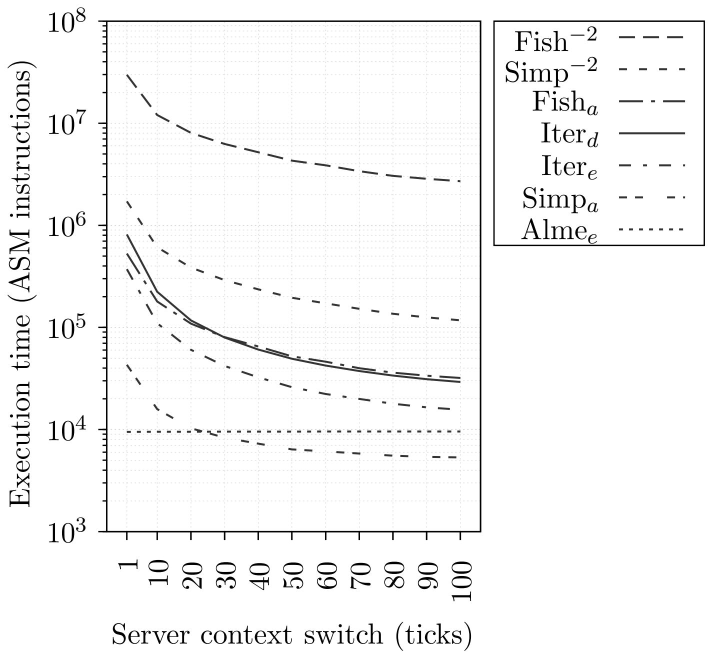

Figure 28: Performance of various server parameter selection approaches for 15 tasks and 10% ITU

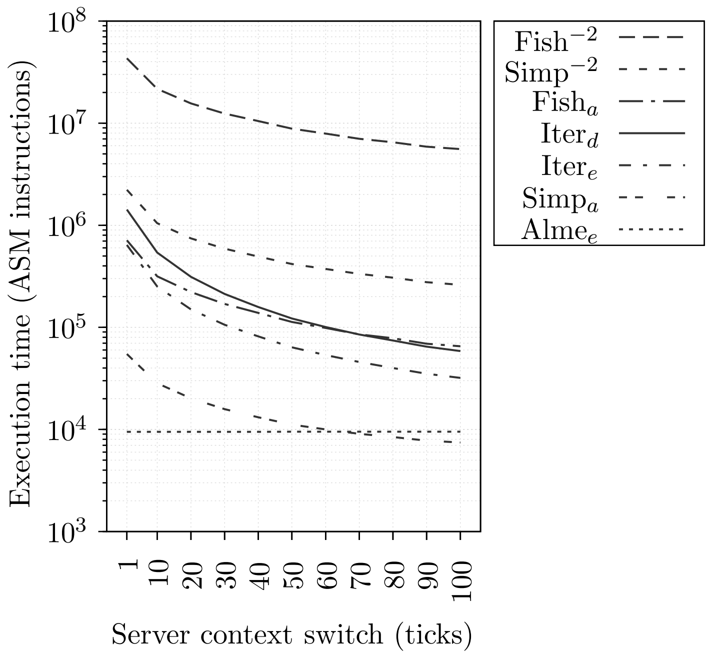

Figure 29: Performance of various server parameter selection approaches for 15 tasks and 25% ITU

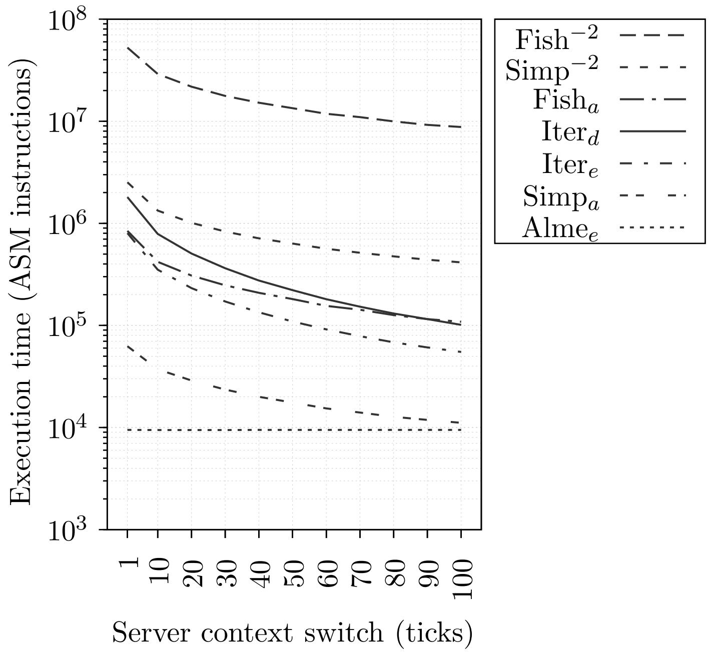

Figure 30: Performance of various server parameter selection approaches for 15 tasks and 40% ITU

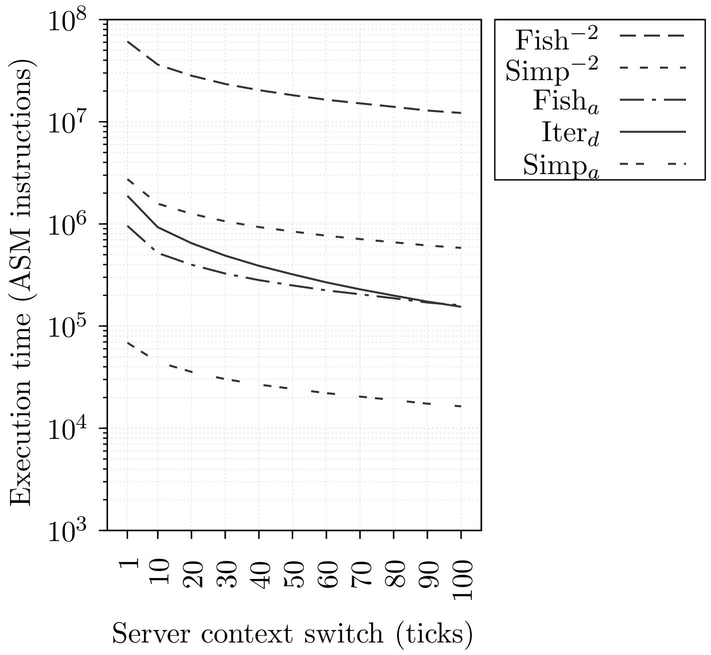

Figure 31: Performance of various server parameter selection approaches for 15 tasks and 55% ITU

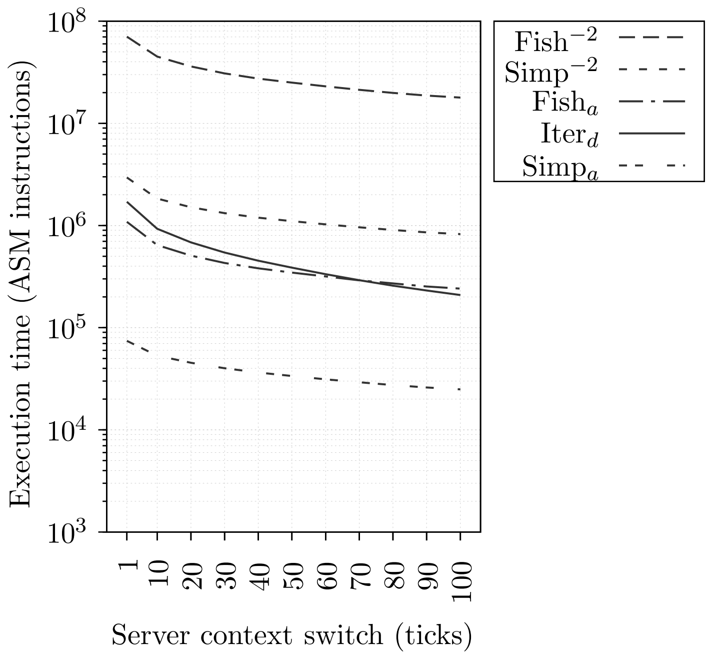

Figure 32: Performance of various server parameter selection approaches for 15 tasks and 70% ITU

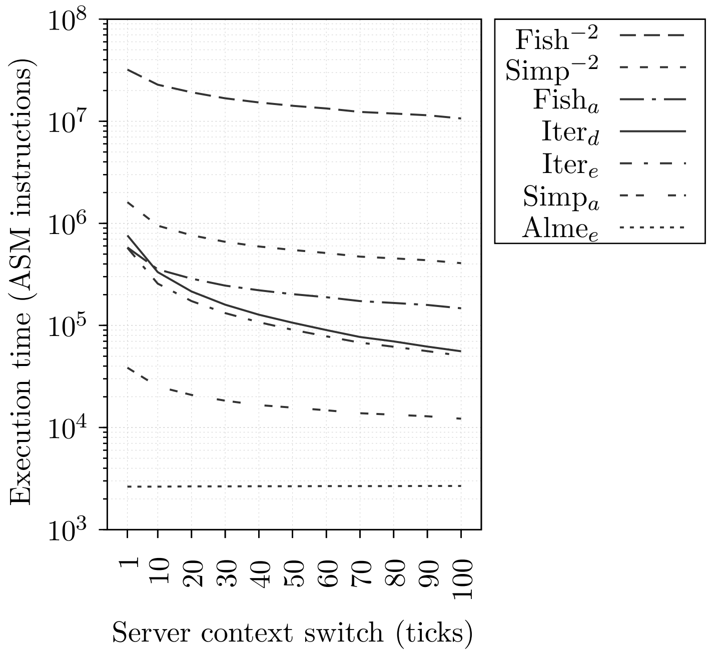

Figure 33: Performance of various server parameter selection approaches for 5 tasks and 25% ITU)

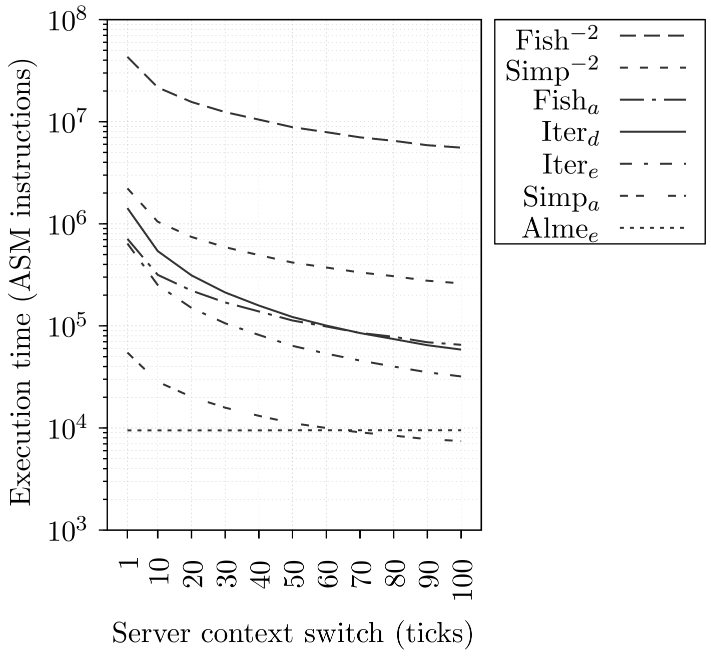

Figure 34: Performance of various server parameter selection approaches for 15 tasks and 25% ITU

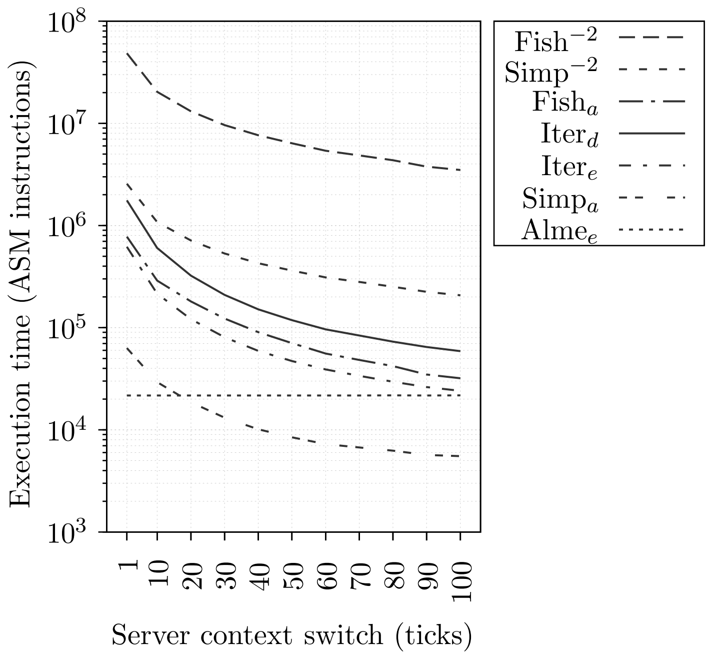

Figure 35: Performance of various server parameter selection approaches for 25 tasks and 25% ITU

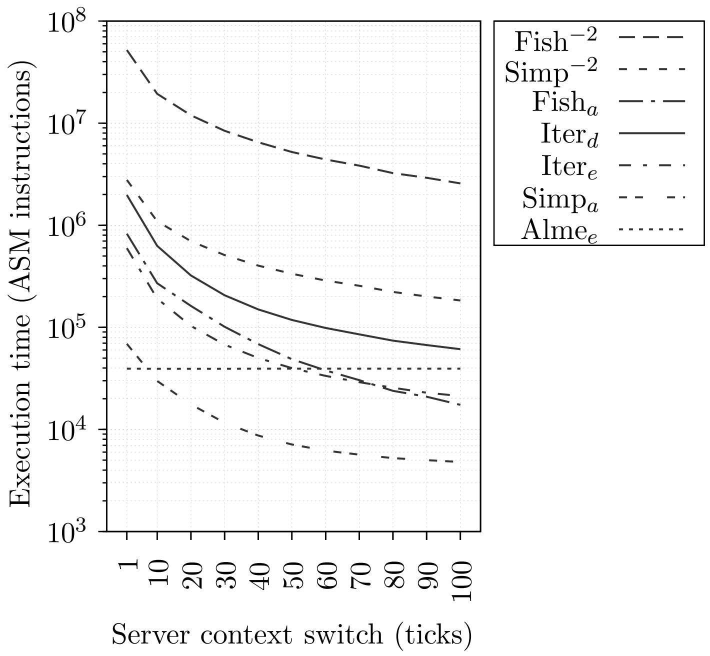

Figure 36: Performance of various server parameter selection approaches for 35 tasks and 25% ITU

4.5.3 Experiment 5: Approximation algorithm performance

This experiment focuses on the impact of the accuracy parameter ϵ on Fisher's and our simple approximation algorithm.

The accuracy parameter ϵ is varied from 1 (used as the maximum value in [41]) down to a value that enables the calculation of the optimal server parameters in the integer domain. To determine the optimal server parameters with the approximation algorithms, the accuracy parameter ϵ was set to 10^{-7} based on the following constraints for the test-cases:

- the uniformly distributed random task periods are selected from the interval $[10^4, 10^6]$,
- the optimal server period has to be less than the largest task period,
- the use of the integer domain for the server parameters.

Based on the setup of the experiments, the largest possible value for the server period and capacity is 10^6. Fisher's algorithm uses the server capacity with an accuracy of $(1+\epsilon)$ to control the search for the optimal parameters. The simple approximation algorithm uses the server period in contrast to the capacity to control the search for the optimal parameters. Though, the ratio between two successive server period values is also at most $(1+\epsilon)$. Setting $\epsilon = 10^{-7}$ forces the approximation algorithms to examine every value in the search interval. This configuration ensures that in the integer domain the optimal server parameters are found by both, Fisher's and the simple approximation algorithm. But due to the computational complexity of Fisher's initial algorithm, measurements for this algorithm were omitted with $\epsilon = 10^{-7}$.

In order to facilitate performance comparison of the algorithms examined in this experiment with the results presented in Experiment 4, the data for the iterative algorithm (Iter_d) is also rendered in Figures 37–41 as a reference. Furthermore, the graphs depicting the performance of Fisher's and the simple approximation algorithm differ in line colour (light and dark grey, respectively) but for a specific ϵ value the same line type pattern is used.

In Figures 37–41, the measurements for the approximation algorithms indicate that given a specific ϵ value, the performance of Fisher's algorithm is approximately a factor of 10 slower in contrast to our simple approximation algorithm. This is due to a much lower number of server parameter values evaluated by the simpler approximation method (also mentioned in Section 4.1). The performance difference of approximately factor 10 between the two approximation method remains the same for each of the examined ϵ values. However, the simple approximation algorithm configured by setting $\epsilon = 10^{-7}$, to determine the optimal server parameters in the integer domain, is still at least a factor of 10^2 slower than the iterative algorithm introduced in Section 4.2.

As the graphs in Figures 37–41 show, the execution-time of the iterative algorithm determining the optimal server parameter values is between the simple approximation algorithm's execution-time for $\epsilon = 10^{-1}$ and $\epsilon = 10^{-2}$, respectively. That means, the server parameters determined by the approximation algorithm with accuracy parameter $\epsilon = 10^{-1}$ may result in a server utilisation that is up to 10% higher than the optimal server's utilisation. Concluding from the measurements, the server parameters determined by the approximation algorithm within approximately the same time as the iterative algorithm, may lead to a processing resource reservation that is up to 10% higher than for the optimal server. Hence, from the performance point of view, the application of the simple approximation algorithm is only meaningful if server parameters, with a resulting utilisation that is 1% or larger than the optimal server utilisation, are acceptable.

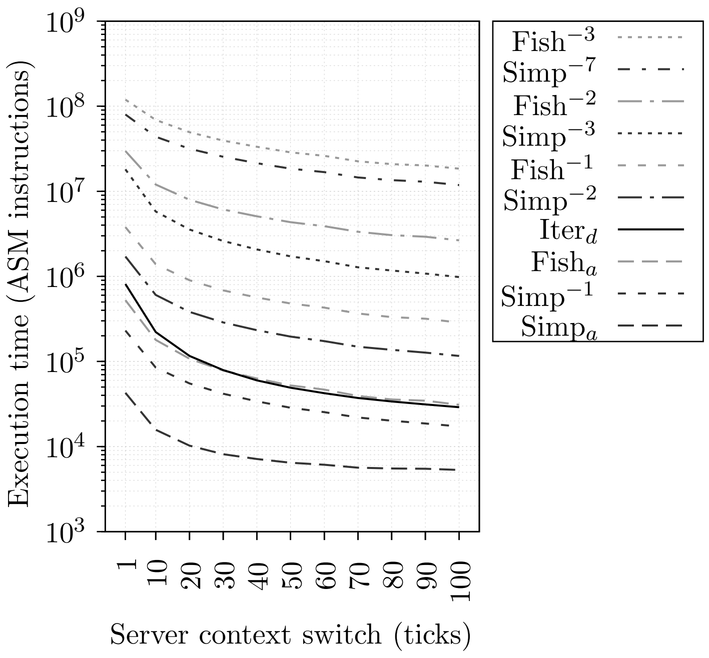

Figure 37: Performance of server parameter approximation algorithm for 15 tasks and 10% ITU)

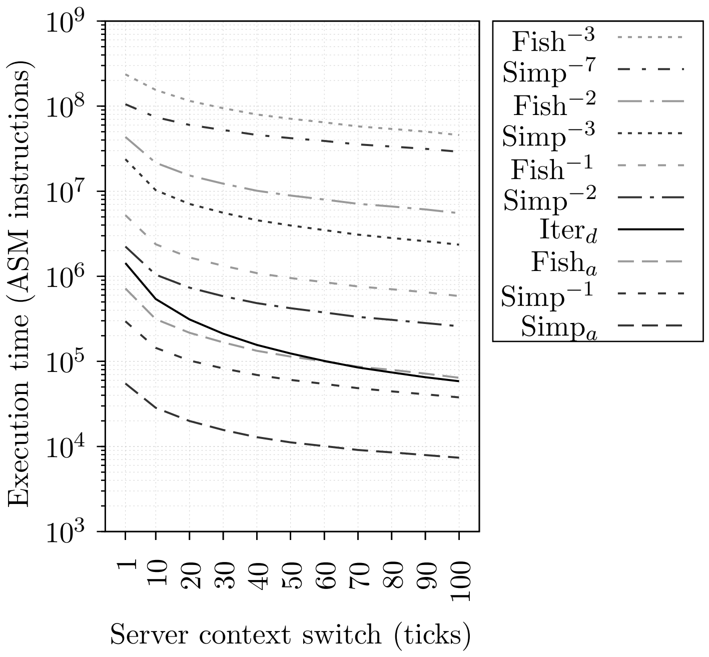

Figure 38: Performance of server parameter approximation algorithm for 15 tasks and 25% ITU

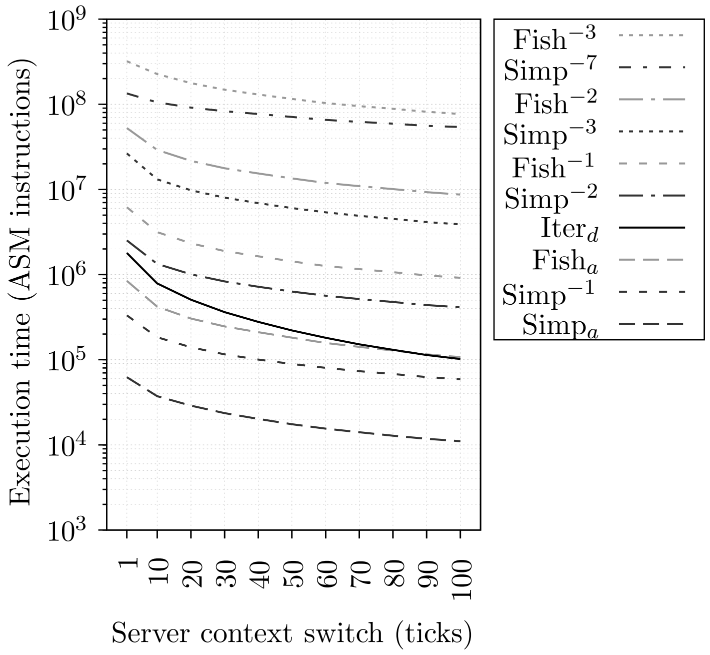

Figure 39: Performance of server parameter approximation algorithm for 15 tasks and 40% ITU

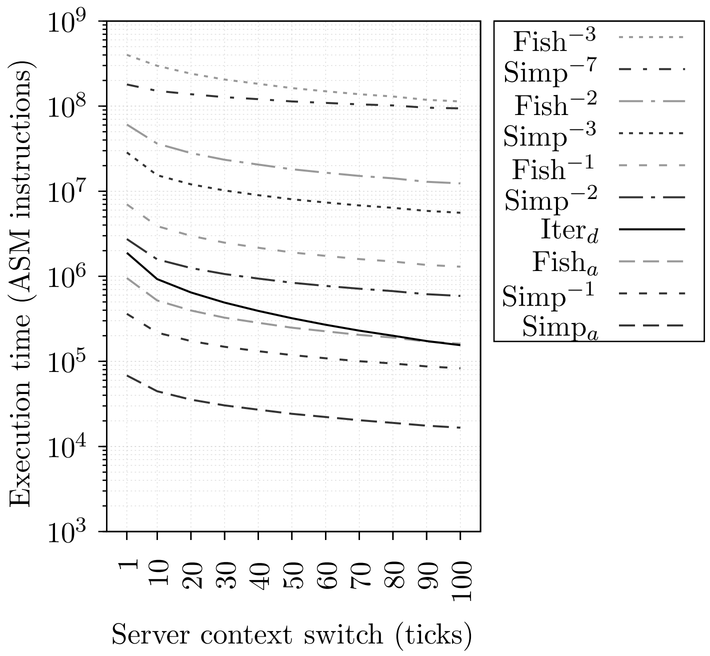

Figure 40: Performance of server parameter approximation algorithm for 15 tasks and 55% ITU

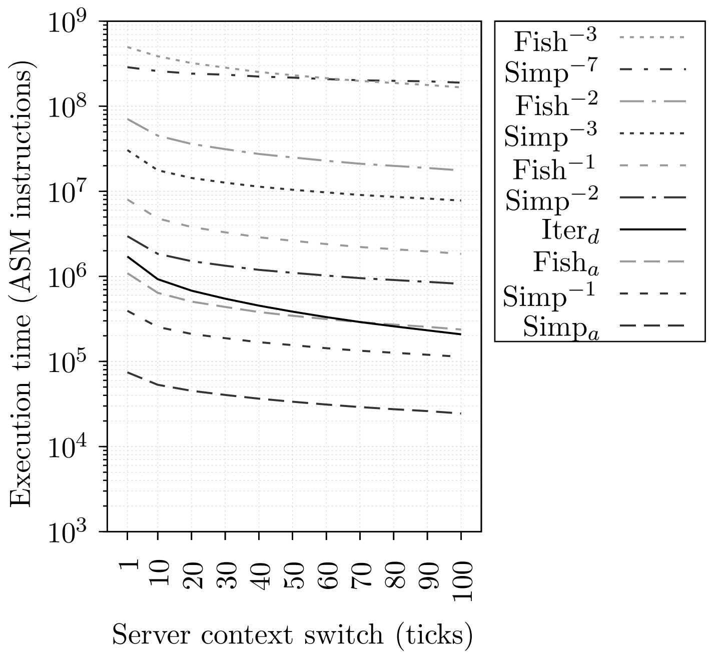

Figure 41: Performance of server parameter approximation algorithm for 15 tasks and 70% ITU

4.5.4 Experiment 6: Use of upper bound values as server parameters

Experiment 6 investigates the overestimation of server utilisation in the case where the server period upper bound Π_u and the corresponding capacity Θ_u are used instead of the optimal server parameters.

Figures 42–46 indicate the behaviour, especially for larger ITU values, that could be observed in the introductory example (Figure 7). The server utilisation difference that is obtained by the use of the optimal and upper bound parameter values, decreases with increasing server context switch overhead.

As the server context switch execution time increases, it has a more significant impact on the server utilisation with the additional consequence that the optimal server period value is shifted toward the upper bound value. The effect of large context switch times, in our test-cases larger than 100 ticks, is that the optimal and the upper bound parameter values coincide more and more frequently. As a consequence, the average server utilisation difference between the optimal and upper bound values is tending towards zero.

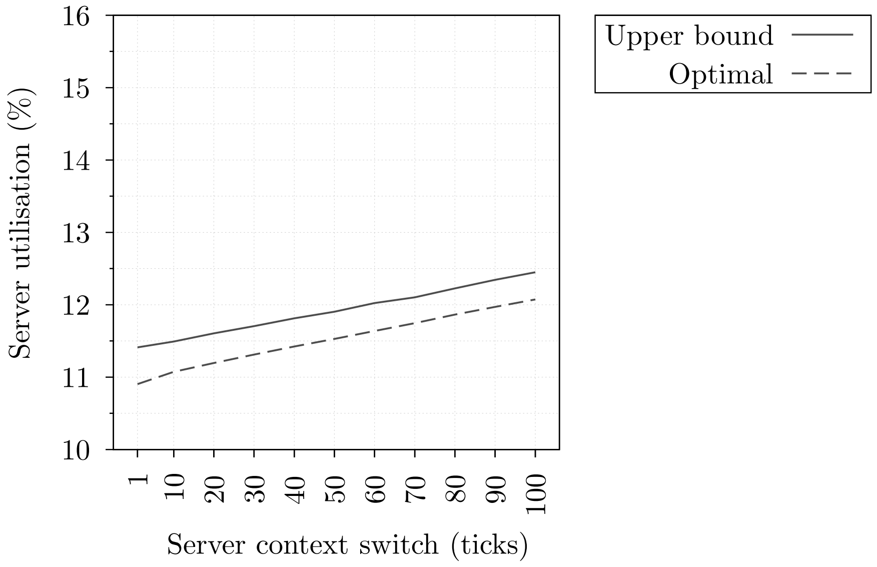

Figure 42: Server utilisation difference for optimal and upper bound parameter values for 15 tasks and 10% ITU

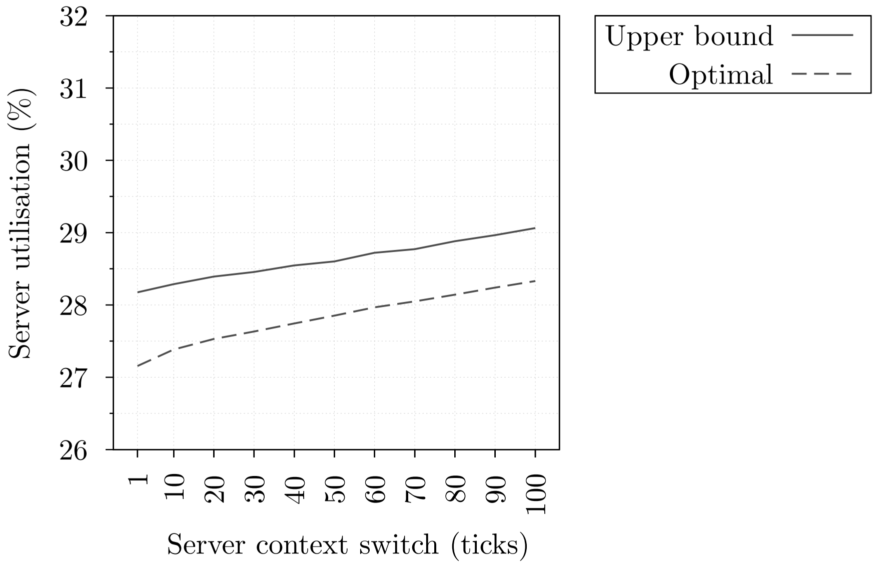

Figure 43: Server utilisation difference for optimal and upper bound parameter values for 15 tasks and 25% ITU

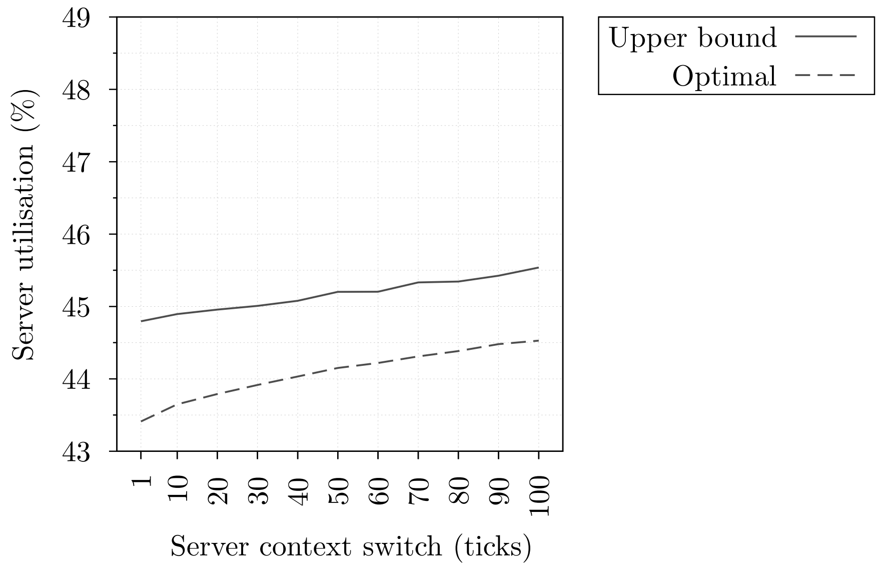

Figure 44: Server utilisation difference for optimal and upper bound parameter values for 15 tasks and 40% ITU

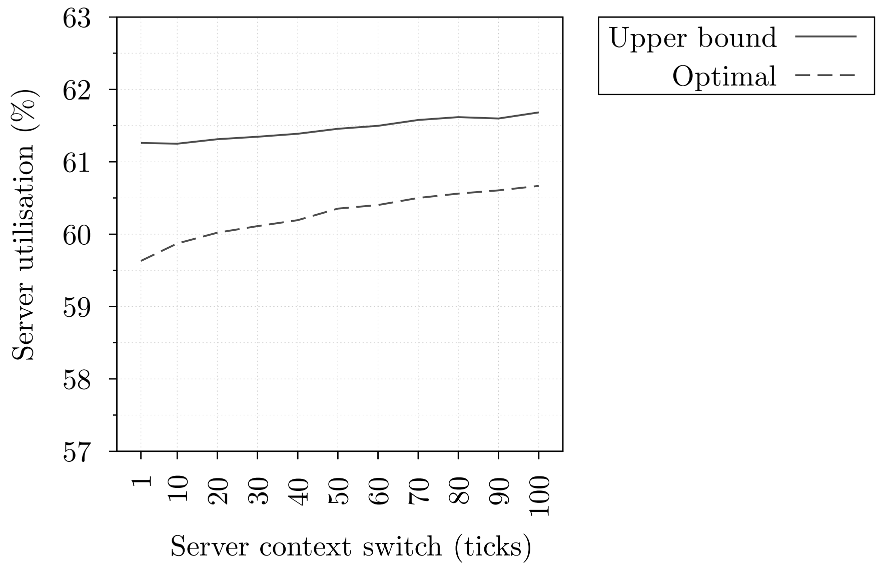

Figure 45: Server utilisation difference for optimal and upper bound parameter values for 15 tasks and 55% ITU

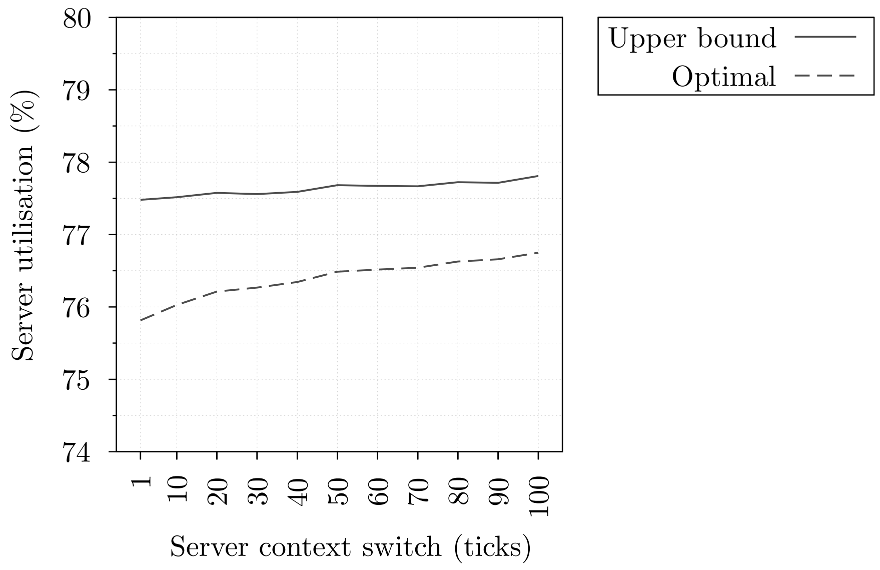

Figure 46: Server utilisation difference for optimal and upper bound parameter values for 15 tasks and 70% ITU

The second part of this experiment focuses on the number of test-cases that overestimate the server utilisation by using the server upper bound parameter values in contrast to the optimal values. The results of this experiment can be examined in Figures 47–48. The graphs show the cumulative percentage of test-cases with overestimated utilisation equal to or less than a certain value. The overestimated utilisation is the difference between the optimal server and the server determined for the period upper bound.

Figures 47–48 show that for the majority of test-cases the utilisation difference was less than 10%. For applications containing 15 or more tasks, it is even likely that the utilisation difference will be smaller than 5%.

The number of tasks in an application also affect the quality of the upper bound values in contrast to the optimal values (see Figures 49–52 and Figures 47–48). As the number of tasks in an application increases, the server utilisation difference between the optimal and upper bound parameter values decreases. The utilisation difference is less than 1% for applications with 15 tasks or more. With the increasing number of tasks in a taskset the number of generated demand points increases as well, resulting in a flattened sawtooth shaped minimal server utilisation graph. A flattened sawtooth server utilisation graph can be justified by the properties of *Stage-1* and *Stage-2* of the iterative algorithm. Within these two stages of the algorithm the server parameter decrements get smaller as the number of demand points increases, resulting in smaller jumps of the server utilisation graph trough and peak points.

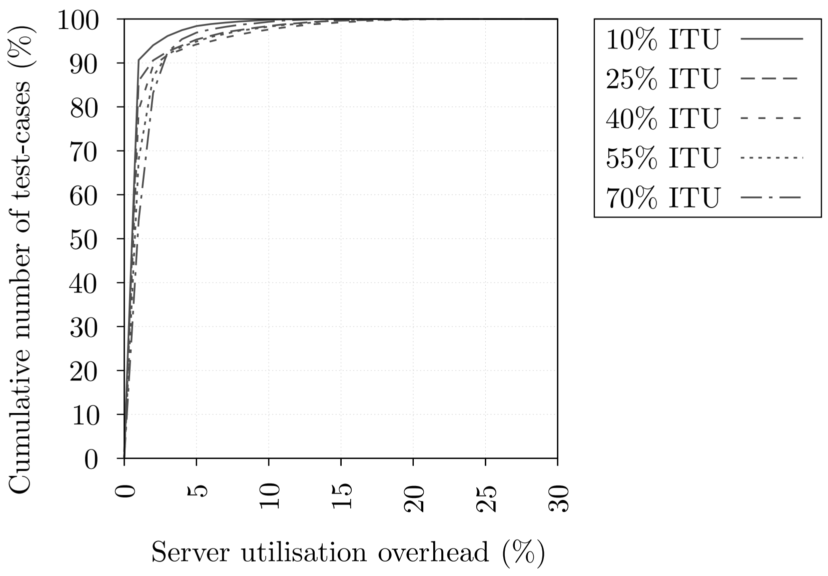

Figure 47: Quality of server upper bound utilisation compared to optimum with $C_0 = 20$ ticks and different ITU values

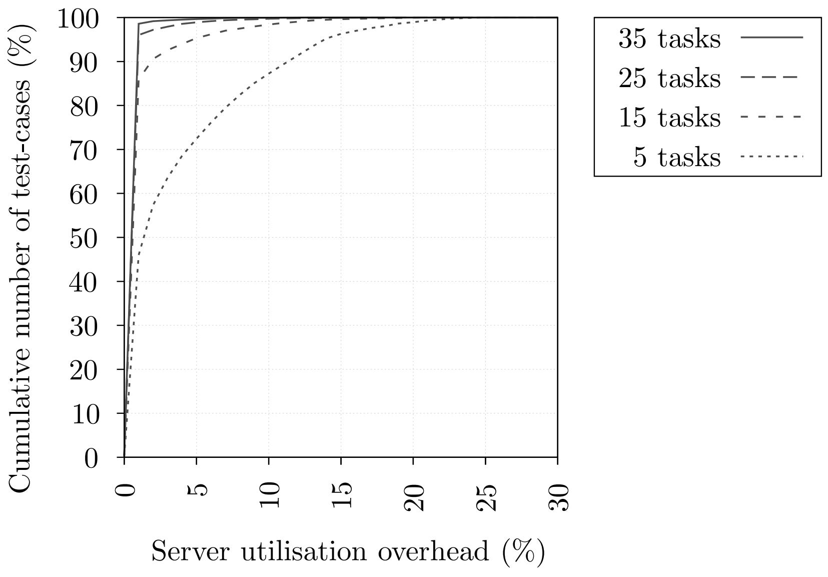

Figure 48: Quality of server upper bound utilisation compared to optimum with $C_0 = 20$ ticks and different taskset sizes

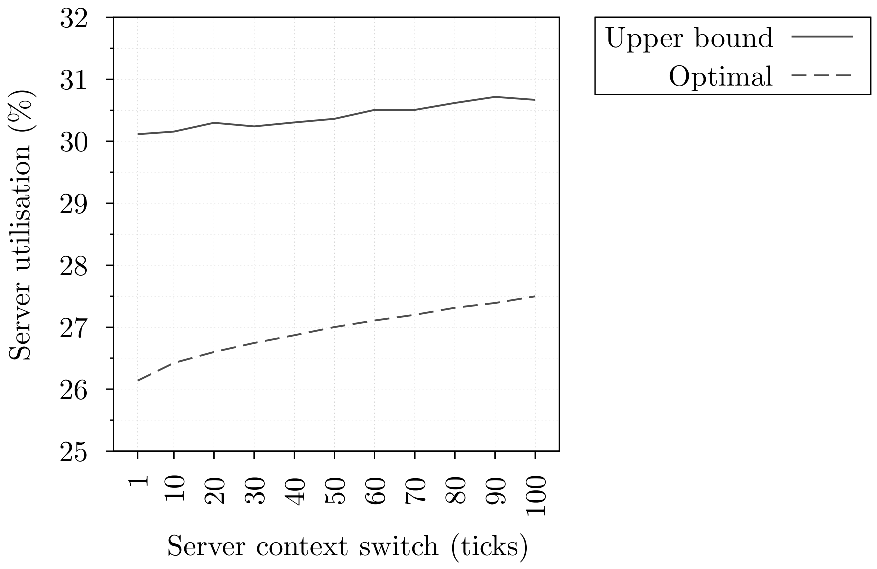

Figure 49: Server utilisation difference using optimal and upper bound parameter values for 5 tasks and 25% ITU

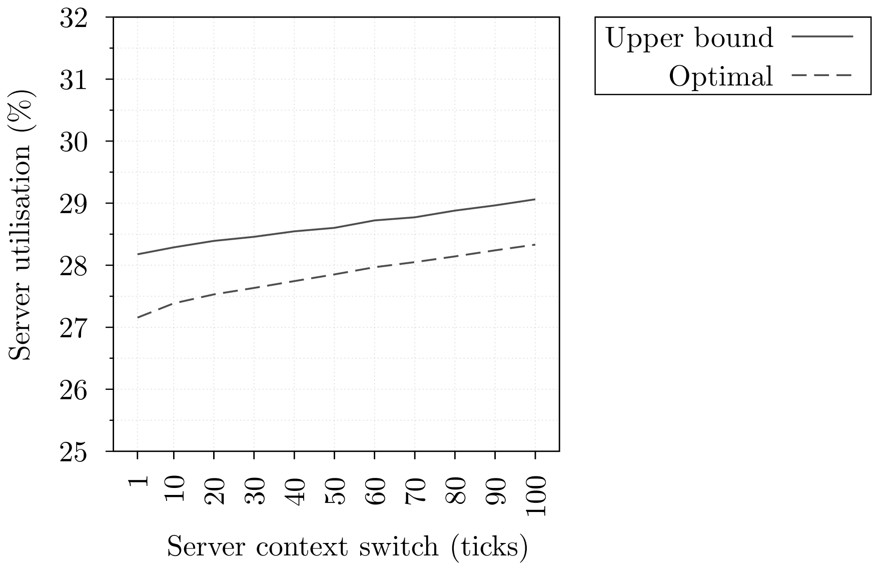

Figure 50: Server utilisation difference using optimal and upper bound parameter values for 15 tasks and 25% ITU

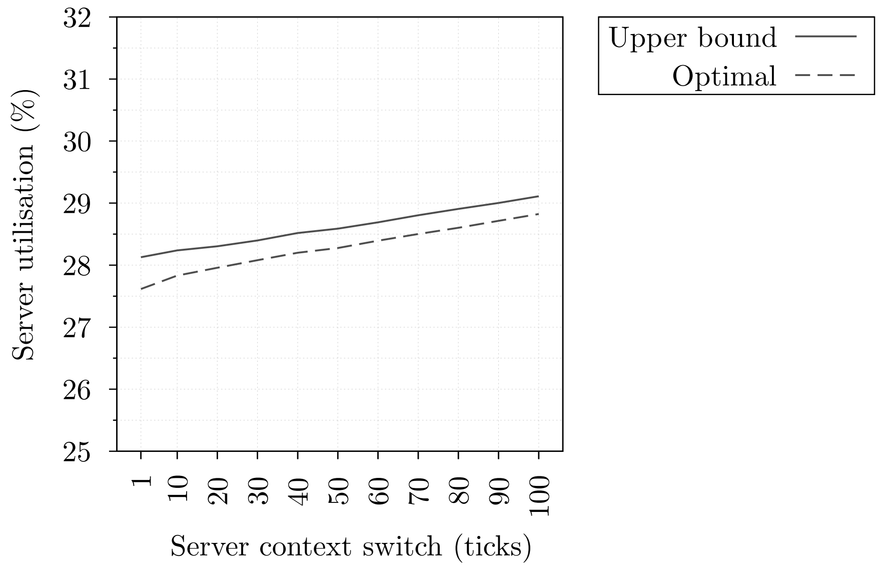

Figure 51: Server utilisation difference using optimal and upper bound parameter values for 25 tasks and 25% ITU

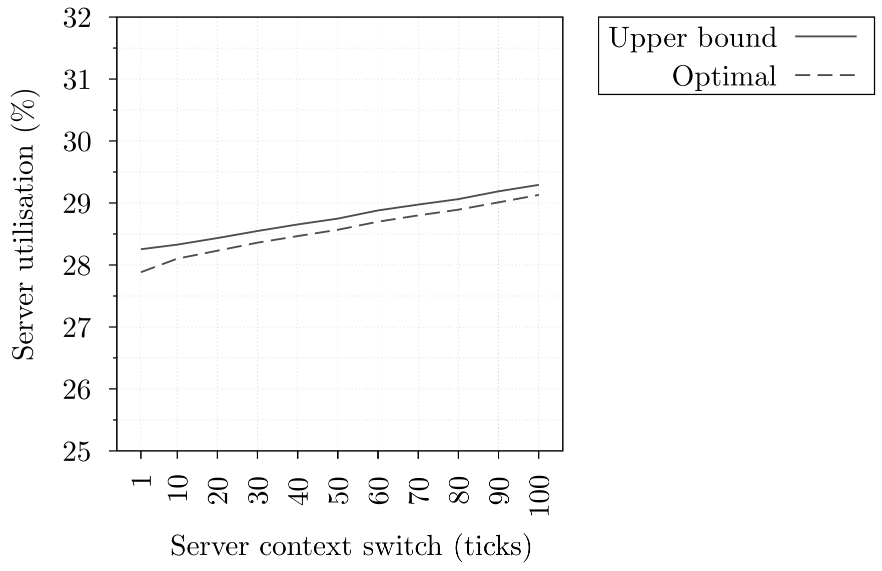

Figure 52: Server utilisation difference using optimal and upper bound parameter values, for 35 tasks and 25% ITU

4.5.5 Evaluation summary

For the selection of non-optimal server parameter values our simple approximation algorithm, based on Fisher's approach [41], or Almeida's and Pedreiras' algorithm [6] can be utilised. With respect to the execution time of the algorithms it was observed that the approximation algorithms are efficient only if the accuracy parameter value $\epsilon > 10^{-1}$ is chosen. In contrast to the optimal values, the server parameters selected by an approximation algorithm result in a slightly higher server utilisation. The utilisation difference is in the range of a few percent for $\epsilon = 1.0$.

In summary, the evaluation showed that the iterative approach for optimal server parameter calculation requires slightly less effort as the simpler approximation algorithms configured with $\epsilon = 10^{-2}$ to calculate non-optimal server parameter values. Hence, our iterative algorithm provides an efficient method to determine the bandwidth optimal server parameter values.

4.6 Summary

In this chapter we examined the selection of optimal server parameter values for fixed-priority scheduled tasksets. The work was motivated by the requirement for an efficient method to calculate the bandwidth optimal server parameter values.

A performance improvement to an approximation algorithm [41] was presented and a new iterative algorithm to determine the bandwidth optimal server parameters for a given taskset was defined.

In summary it can be stated that:

- we analysed the task demand function and further investigated its applicability to the parameter selection of execution-time servers,
- in contrast to most published methods, we incorporated the server context switch overhead into the analysis,
- presented a simple approximation algorithm for server parameter selection based on Fisher's approach [41].
- presented an efficient iterative approach to determine the bandwidth optimal server parameters. The algorithm discards the majority of values

within the optimal server period interval and evaluates only the server period candidates that might result in bandwidth optimal parameters.

In the worst-case, observed over 2,200,000 evaluated tasksets, the maximum number of executed assembly instructions was just below $5{\cdot}10^6$ to find the optimal server parameters using the iterative approach introduced in this chapter. In the evaluation with the task period values limited to the range $[10^4, 10^6]$ time ticks, this worst-case situation occurred for the test-case configuration with 35 tasks in a taskset, 40% ITU and server context switch time equal to 1 tick.

Assuming a performance of 14,000 MIPS[5] for a modern desktop PC with a 2.0 GHz CPU, the $5{\cdot}10^6$ assembly instructions executed by our iterative approach correspond approximately to an absolute execution time of 360 µs. The short absolute execution time shows that the iterative algorithm is easily applicable to tasksets with more than 35 tasks and to a larger interval of task period values before the calculation of the optimal server parameters by the iterative algorithm becomes intractable.

We conclude that for a wide range of real systems, this method provides a viable and efficient way of determining optimal server parameters.

Given the efficient offline methods to determine near optimal and optimal server parameter values for a taskset, the remaining of the thesis is focusing on providing similarly efficient approaches to verify the schedulability of application tasksets and to distribute during runtime the possible spare processing capacity among flexible applications in the system.

[5]Million Instructions Per Second

Chapter

5

Efficient online acceptance test

A method, to calculate the optimal server parameters for a given taskset, was defined in the previous chapter. For the determined server parameters, the processing resource reservation enforced by the server mechanism is the smallest possible reservation, such that the tasks in the associated taskset are still schedulable assuming the worst-case timing for task releases, and the server's execution capacity supply and replenishment.

Since the schedulability of the taskset was ensured at the time of the server parameter selection, in an open real-time system where applications are submitted into the system and may terminate after a while, only the schedulability of the corresponding servers needs to be verified. The schedulability analysis has to determine for the prospective set of servers in the system whether the execution capacity of each server can be supplied to the associated tasks or not. In order to maintain the schedulability of all real-time tasks in the system, the resource reservations expressed by the servers have to be maintained at all time. Hence, a new application can be submitted into the system, and the corresponding server can be created if the schedule of the new set of servers in the system remains feasible. That means, a new application is accepted for execution only if it does not jeopardise the schedulability of already running applications.

Based on the available processing resources, the decision about whether a new application can be launched in an open real-time system or not, and the resulting schedulability test of active servers has to be carried out during runtime of the system due to its dynamic behaviour. Therefore, an efficient online schedulability test is required to determine the feasibility of requested changes of resource reservation in a running system. The following sections examine the implementation of an effective schedulability test providing the necessary means for an online acceptance test.

5.1 Need for an efficient schedulability test

Schedulability tests are a vital component of the real-time theory. Especially the runtime complexity of schedulability tests represents an important aspect that becomes even more crucial if applied by runtime tests to reason about the schedulability of a running system.

In open real-time systems, it is important to determine the schedulability of servers that would be created in the case of acceptance of newly submitted applications. That means, the online acceptance test in an open real-time system has to determine whether the entire execution capacity of each server in the system can be utilised within the corresponding server's replenishment period or not, if the prospective set of active applications changes.

The importance of an efficient online schedulability test in open real-time systems is motivated by the need to determine the schedulability of a server set in a timely manner using as little as possible of the system resources. Therefore, optimisation steps are required to enable the reduction of resource reservation for the online acceptance test that is considered to be an inherent part of the system specific functionality.

However, as indicated by Davis et al. [32] not all theoretical optimisation steps of a schedulability test's runtime complexity lead to an improvement of the test's execution time. For example, based on the applied evaluation method, the *Hyperplane Exact Test* [14] might outperform the *Response Time Analysis* based test, but measurements carried out on a real embedded system suggest the opposite [32]. Hence, it is important not just to consider theoretical but also implementation related optimisation steps for a schedulability test in order to provide hints and rationale for certain optimisation steps. This implies the need for measurements on real embedded systems and shows the added value of results obtained by real execution time measurements.

The theoretical foundation provided for the schedulability analysis of periodic and sporadic task will be used in the following sections as a starting point to define and suggest optimisation steps for an efficient online acceptance test of real-time applications. The rationale for this decision is based on the fact that classical schedulability analysis concepts for periodic and sporadic real-time tasks can be reused (see also Section 2.3) to determine if multiple periodic servers are feasibly schedulable on a uniprocessor system or not.

5.2 Schedulability test optimisation

In fixed-priority scheduled systems we distinguish between *sufficient* and *exact* schedulability tests. For example, the test introduced by Liu and Layland [67] is based on the utilisation of a taskset and it belongs into the category of sufficient tests. With respect to the execution time, sufficient tests usually outperform the exact tests, but they also provide *false negative* results. That means, some task or server sets are actually schedulable but the sufficient test classifies them as unschedulable.

In order to avoid the reduction of usable processor capacity resulting from an inaccurate schedulability test, we are interested in an efficient test for fixed-priority scheduled systems that can exactly determine the schedulability of a task or server set. That means, if a set of entities is schedulable on a uniprocessor system, then the schedulability test shall identify it as such. The test shall also avoid false negative results by classifying task or server sets unschedulable only if they cannot be feasibly scheduled by the applied scheduling policy.

In fixed-priority scheduled systems the demand based [58] and the response time [8] analysis are the most common methods to exactly determine the schedulability of a system. Due to a wider applicability of the response time analysis, not just to determine the schedulability but also to calculate the response time of tasks, most optimisation steps focus on this approach. Since the response time is calculated by a recurrence relation, the optimisations are mainly concerned with the definition of improved initial values. The objective of improved initial values is to reduce the number of iterations required by the recurrence relation in order to determine the worst-case response time. However, the best performance improvements are achieved by the combination of different optimisation approaches as the following sections will show.

5.2.1 Response time calculation

In an open real-time system it is crucial to guarantee that each active server can supply its full capacity to the associated tasks. This assurance is important since the server parameters, expressing the capacity supply and resource reservation, were used by the offline method (presented in Chapter 4) to provide offline guarantees that the temporal requirements of the tasks can be satisfied.

Initially the response time calculation was applied to periodic tasks to calculate their worst-case response times in a fixed-priority scheduled system. The corresponding recurrence relation was introduced in Section 2.5 (see Equation 2.7). Based on the worst-case response time, the schedulability of a task could be determined by comparing this value to its deadline.

In the following, we will consider the schedulability analysis in the context of execution-time servers and adapt the equations using appropriate notation. With a focus on open real-time systems and the resource reservation enforced by servers, the worst-case response time value determined for a server actually denotes the time by which the server's execution capacity is fully consumed if the associated taskset never idles.

The recurrence relation used to calculate the response time of tasks is restated in Equation 5.1 considering the notation for the server parameters. ω_i^n denotes the response time of server S_i in the n-th iteration of the recurrence relation. Θ_j refers to the execution capacity and Π_j to the replenishment period of a server S_j with priority higher than S_i.

$$\omega_i^{n+1} = \Theta_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{\omega_i^n}{\Pi_j} \right\rceil \Theta_j \tag{5.1}$$

In order to ensure during runtime the resource reservation that is represented by the server, the response time analysis has to determine a time instant for the server supply completion that is less than or equal to the server's replenishment period.

Although the schedulability and acceptance test based on the response time analysis has a pseudo-polynomial runtime complexity, the first optimisation step is focusing on the recurrence relation. The optimisation trial is concerned with the number of iterations that are required by the recurrence relation to determine the completion time of the server supply.

A detailed pseudo-code for the standard implementation of the recurrence relation calculating the response time of a server, is depicted in Algorithm 9. It is assumed that the servers are ordered according to their priorities, with 1 denoting the highest priority server.

```
1 Initialise the last calculated response time $\omega'_i$ to 0;
2 Determine and assign a feasible initial value to current response time $\omega$;
3 while ($\omega_i > \omega'_i$) and ($\omega_i \leqslant \Pi_i$) do
4     Save current response time $\omega_i$ as last calculated response time value
      in $\omega'_i$;
5     Include the server's own execution capacity into the calculation of
      the response time by assigning $\omega_i = \Theta_i$;
6     for each server with priority higher than i, i.e. $j = 1$ to $(i - 1)$ do
7         Calculate intermediate response time for server $S_i$ by the
          equation $\omega_i \mathrel{+}= \left\lceil \frac{\omega'_i}{\Pi_j} \right\rceil \Theta_j$;
8     end
9 end
```

Algorithm 9: Standard implementation of the recurrence relation [32]

At this point the focus of optimisation is specifically on the computational window that is used in the recurrence relation to consider higher priority server interference. The purpose of the optimisation step is to reduce the number of iterations required to determine the worst-case completion time of the server's execution capacity supply. It should be noted that the completion time of the server supply is also considered as the length of the computational window during which the interference caused by higher priority server is considered.

Although the temporarily computed response time ω_i, in line 7 of Algorithm 9, is successively updated while the *for*-loop is executed, a possible increase of this value is not immediately considered for the calculation of higher priority interference. The computational window ω'_i for the calculation of higher priority interference is only updated after the interference on S_i caused by all higher priority servers has been determined for one complete iteration of the recurrence relation.

The general idea in a gradually updated value for the computational window is to accelerate the convergence of the recurrence relation and consequently reduce the number of iterations that is required to determine the final solution. In order to implement the improvement resulting from the gradual update of the computational window length, the standard implementation is split into two parts. The pseudo-code implementation of the envisaged optimisation step is depicted in Algorithm 10.

```
1  Determine and assign a feasible initial value to last calculated response
   time $\omega_i'$;
2  Include the server's own execution capacity into the calculation of the
   response time by assigning $\omega_i = \Theta_i$;
3  **for** *each server with priority higher than i, i.e. $j = 1$ to $(i-1)$* **do**
4      Calculate the interference $I_j$ caused by the higher priority server $S_j$
       on server $S_i$ by evaluating $I_j = \left\lceil \frac{\omega_i'}{\Pi_j} \right\rceil \Theta_j$;
5      Use the higher priority interference $I_j$ to increment the intermediate
       response time for server $S_i$, i.e. $\omega_i += I_j$;
6  **end**
7  **while** *$(\omega_i > \omega_i')$ and $(\omega_i \leqslant \Pi_i)$* **do**
8      Save current response time $\omega_i$ as last calculated response time value
       in $\omega_i'$;
9      **for** *each server with priority higher than i, i.e. $j = 1$ to $(i-1)$* **do**
10         Calculate higher priority interference $\left\lceil \frac{\omega_i}{\Pi_j} \right\rceil \Theta_j$ for server $S_i$ and
           store the value in a temporary variable *tmp*;
11         Use the higher priority interference, $tmp - I_j$, caused by server $S_j$
           to increment the intermediate response time $\omega_i$ for server $S_i$;
12         Store *tmp* in $I_j$ as the new value representing the interference of
           server $S_j$ on $S_i$;
13     **end**
14 **end**
```

Algorithm 10: Optimised implementation trial of the recurrence relation [32]

With the new alternative implementation of the recurrence relation, the interference of higher priority servers $S_{j:j \in hp(i)}$ on S_i is successively determined and the potentially larger intermediate value of the computational window is used to determine the interference of the next server on S_i. The main difference of the optimised implementation in contrast to the standard version include the lines 3–5 and 10–12 of Algorithm 10. The actual implementation of the recurrence relation is still located in the *while*-loop. However, due to the alternative implementation, certain values need to be calculated beforehand. The first *for*-loop is in charge of calculating a starting value for the higher priority interference I_j. In the second *for*-loop the interference caused by each server S_j on S_i is recalculated based on the latest intermediate value of the computational window and it is stored again in I_j.

Since the most recent value of the computational window is used to calculate the interference of each higher priority server on S_i, it is expected that the calculation of the server's response time is accelerated. The performance impact of the alternative implementation will be further examined and final conclusion will be drawn in the evaluation section. The next section takes a different approach but also addresses the performance of iterative algorithm's convergence towards a solution.

5.2.2 Initial values for response time analysis

For tasksets, Audsley et al. [8] suggested in their initial work that the choice of an appropriate initial value could accelerate the convergence of the recurrence relation. They proposed to evaluate the response time of tasks in decreasing priority order and to use the response time of the next higher priority task instead of zero, as an initial value to start the execution of the iterative algorithm. For the analysis of servers, this initial value would be equivalent to the completion time of the next higher priority server.

Subsequent publications [7, 17, 94] provided further improved initial values by exploring in detail the properties of fixed-priority scheduled systems. The main focus was on the increase of the response time lower bound.

The most common suggestions for an improved initial value are the response time of the next higher priority task plus the task's execution time [7]

$\omega_{i-1} + C_i$. Sjödin and Hansson extended this initial approach by accounting for blocking [94]. For servers, the corresponding initial value without the blocking factors would be defined as the sum of the response time ω_{i-1} of server S_{i-1} and the execution capacity Θ_i of server S_i (see Equation 5.2). Bril et al. [17] derived a closed form equation for the initial value calculation by approximating the ceiling function in the recurrence relation (see Equation 5.3).

$$\omega_i^0 = \omega_{i-1} + \Theta_i \tag{5.2}$$

$$\omega_i^0 = \frac{\Theta_i}{\left(1 - \sum\limits_{\forall j : j \in hp(i)} U_j\right)} \tag{5.3}$$

All these initial values represent a feasible lower bound on the server's response time, hence they can be used not just to determine the schedulability but also to calculate the exact worst-case response time of a server.

In contrast to previous approaches, with a single initial value for each priority level, the following citation from [32] with adaptation of the notation for servers, introduces a series of initial values of which the maximum may further reduce the number of recurrence relation iterations required to determine the response time.

> For each priority level i, there are i initial values in the series. To form each new initial value, identified by the index k where $(k = 1..i)$, we partition the set of servers of higher than or equal priority to i, i.e. $hep(i)$, into two sets: $hp(k)$ and $lep(k) \cap hp(i)$.
>
> Following the approach of Lu et al. in [69], we consider the servers in $hp(k)$ as taking a proportion of the available processing time α_k where:
>
> $$\alpha_k = \sum_{\forall j \in hp(k)} U_j$$
>
> Thus only a fraction of the processing time $1-\alpha_k$ remains available to accommodate the remaining servers. Given that $\omega_i \geqslant \omega_{i-1}$, the contribution of each server in $hep(i)$ to the total server load in i is

at least $I_j(\omega_{i-1})$, where $I_j(\omega_{i-1})$ is the worst-case interference due to server $S_j \in hp(j)$ occurring during the response time of server S_{i-1}.

$$I_j(\omega_{i-1}) = \left\lceil \frac{\omega_{i-1}}{\Pi_j} \right\rceil \Theta_j$$

We note that as the response time of server S_i is only computed if server S_{i-1} is schedulable, $I_{i-1}(\omega_{i-1}) = \Theta_{i-1}$.

Using this information, we compose a series of i lower bounds on corresponding to each priority k from 1 to i.

$$\omega_i^{LB}(k) = \frac{\Theta_i + \sum\limits_{\forall j \in lep(k) \cap hp(i)} I_j(\omega_{i-1})}{1 - \sum\limits_{\forall j \in hp(k)} U_j} \tag{5.4}$$

The largest such bound is given by:

$$\omega_i^{LB} = \max_{\forall k=1 \ldots i} \omega_i^{LB}(k) \tag{5.5}$$

Depending on the purpose of the response time analysis, it might be feasible to apply initial values larger than the response time lower bound. That means, if the exact completion time of the server supply is required, the largest safe initial value is the response time lower bound. But if a boolean result for the schedulability of server set is sufficient, then initial values larger than a lower bound may be used.

Initial values might be larger than the smallest converging solution of the recurrence relation, but still sufficiently small to ensure that for a schedulable server the determined response time is smaller than the given replenishment period. Utilising the server's replenishment period, two of these sufficient initial values, i.e. $\Pi_i/2$ and $\Pi_i - \Pi_{i-1}$, were proposed by Lu et al. [69]. Based on the work of Lu et al. new sufficient initial values were derived by Davis et al. [32] that guarantee the convergence of the recurrence relation to ω_i^{UB} where $\omega_i \leqslant \omega_i^{UB} \leqslant \Pi_i$ if server S_i is schedulable. The sufficient initial values [32] are summarised in the following four Equations.

$$\omega_i^0 = \Pi_i - \Pi_{i-1} \tag{5.6}$$

$$\omega_i^0 = \Pi_i - \omega_{i-1}^{UB} \quad \text{assuming } S_{i-1} \text{ is schedulable} \tag{5.7}$$

$$\omega_i^0 = \Pi_i/2 \tag{5.8}$$

$$\omega_i^0 = (\Pi_i + \Theta_i)/2 \tag{5.9}$$

A further optimisation step can be considered in the case of the four aforementioned initial values [32]:

> Using the initial values given by Equations 5.6 to 5.9, the next value generated by the recurrence relation may, in some cases, be smaller than the initial value. If so, iteration can be terminated immediately, as the server is then known to be schedulable, with the value computed on the first iteration providing an upper bound on the worst-case response time.

Given the various initial values as an improvement for the response time analysis, in the evolution section we will examine their impact on the performance of the exact boolean schedulability test implemented through response time calculation and provide suggestions for the best choice in an online test.

5.2.3 Sufficient schedulability test

Previous publications [12, 16, 67, 79] defined different upper bounds for the processor utilisation and response time that can be utilised to carry out sufficient schedulability tests. It has been shown [12, 32] that these bounds have a lower computational complexity than the schedulability test based on response time analysis.

The response time upper bound (see Equation 5.10) proposed by Bini and Baruah [12] allows a sufficient test to be applied on a per server basis, instead of the entire server set. Furthermore, using Equation 5.10, the time complexity to calculate the response time upper bound is $O(n)$ in contrast to the approach proposed by Rahni et al. [79] that is based on the response time recurrence relation with pseudo-polynomial time complexity.

$$\omega_i^{UB} = \frac{\Theta_i + \sum\limits_{\forall j \in hp(i)} \Theta_j \left(1 - U_j\right)}{1 - \sum\limits_{\forall j \in hp(i)} U_j} \tag{5.10}$$

For each server, using Equation 5.10 the response time upper bound can be efficiently calculated. Therefore, previous to a thorough response time analysis of a server, its response time upper bound can be calculated and compared to its replenishment period. If the response time upper bound already satisfies the condition, $\omega_i^{UB} \leqslant \Pi_i$, then there is no need to carry out the expensive iterative response time calculation utilising the recurrence relation. Hence, the response time upper bound based test can be used as a pre-test to decide if the more elaborate schedulability test has to be carried out or not.

The advantage of the response time upper bound based test as a pre-test will be evaluated in the following section. Furthermore, thorough evaluation of the two implementation approaches (introduced in Section 5.2.1) and the impact of the different initial values (see Section 5.2.2) on the performance of the exact boolean schedulability test are presented.

5.3 Evaluation

In order to obtain representative performance measurements and to draw the right conclusions from the captured values, the various optimisation approaches are evaluated on a real embedded hardware platform additionally to the more abstract metrics that are captured on a desktop computer.

In contrast to the evaluation in Section 4.5, where the number of executed assembly instructions on a desktop computer was used, in the following sections we will use the execution time measurements obtained on a real embedded development board. The rationale for this approach is that the algorithms presented in Chapter 3 and Chapter 4 are offline algorithms executed on desktop computers during the design of the applications and the system, whereas the algorithms presented in Chapter 5 and Chapter 6 are intended to be executed online as an inherent part of a running system. Therefore, the measure of executed assembly instructions is sufficient to judge the efficiency of an offline algorithm, but for an online algorithm the real execution time is an advantage, providing an insight into the runtime behaviour of the proposed implementations.

The embedded hardware platform used for real execution time measurements contained an MPC555 microcontroller running at 40 MHz system clock. All the relevant data used in the tests, like the temporal parameter values of servers, was allocated in the 4 MByte external SDRAM. The compiled code and the function call stack were allocated in the microcontroller's 26 KByte internal SRAM.

To create the target executable file, a development environment comprising the GNU C compiler and RapiTime version 1.2, a tool for worst-case execution time analysis, was used. The target executable files were compiled with compiler optimization enabled, using the compiler option *-O2*. To avoid falsification of the measurements by intermediate performance monitoring of the schedulability test, only end-to-end execution times were recorded. The obtained execution time and the corresponding number of iterations were transferred to the host (desktop) computer where all the data was stored and analysed.

However, additionally to the real execution times, a more abstract metric that is not affected by hardware specific features is beneficial. Such a metric is the *number of ceiling operations (CeilOps)* required by the recurrence relation to converge. The number of CeilOps allows us to examine the computational complexity of the schedulability test as a function of its input variables. Apart from the number of ceiling operations the number of iterations required by the recurrence relation was also used in previous publications [69, 94]. However, it was argued [32] that during each iteration of the recurrence relation a varying number of CeilOps will be performed, leading to a less accurate measure to express the computational effort required by the schedulability test.

For tasks, Davis et al. [32] showed that the performance of the schedulability test is also influenced by the range of task periods. In order to examine the effect of different ranges of server periods on the schedulability test, we executed the test-cases with a two, four and six orders of magnitude ranges of server period. These ranges of period are considered [32] to reflect the period distribution found in commercial applications. However, unlike tasks, servers are not expected to have a wide range of periods. Therefore the large part of the evaluation is limited to four period magnitudes. The periods were defined in the intervals $[10^3, 10^4]$, $[10^4, 10^5]$, $[10^5, 10^6]$ and $[10^6, 10^7]$. The servers in the system were evenly distributed among the period magnitude ranges. For example, in an experiment performed with 24 servers and periods spanning four orders of magnitude, there are groups of six servers that have the same order of magnitude periods.

The test-cases generated for the evaluation are comprised of a set of servers. According to the number of servers in the system and the maximal predefined utilisation (subsequently referred to as the *Initial Target Utilisation (ITU)*) of this set, the algorithm presented by Bini and Buttazzo [15] generates appropriately distributed utilisation value for each server. The server parameters are then derived from the randomly generated server utilisation value. First, the server's period is randomly chosen from its preassigned period magnitude range. Given the server's utilisation and period, the calculation of its execution capacity is straight forward.

In the experiments, before a test-case was carried out, it was ensured that for a chosen ITU the derived server set is schedulable. Test-cases with initially unschedulable server sets were not considered in the measurement and they were replaced with a new and schedulable server set. Since the performance measurements in this chapter are concerned with online schedulability test, the generated servers are assumed to have fixed parameters. The possibility to vary the server's temporal parameters will be examined in Chapter 6.

For the performance evaluation on a desktop computer, the impact of the different optimisation possibilities presented in this chapter was considered based on 10,000 different server set configurations. The performance measurements on the embedded system, were slightly different to that on the desktop computer. Due to the limited performance and resources of the embedded system the number of generated test-cases was reduced to 3,000 for each setting of the test variables.

To enhance the readability of the figures in the following sections, the legends were abbreviated. The list of abbreviations is defined in Table 5. Most of them denote initial values that are used by the standard implementation (as defined by Algorithm 9) of the recurrence relation. Only the last two entries in Table 5 refer to initial values used in conjunction within the alternative implementation presented in Algorithm 10.

Table 5: List of abbreviations used for schedulability test experiments

Initial value	Description
Θ_i	Using the server's execution capacity as the initial value. This is the classical approach.
$E_{5.3}$	Using the information about the utilisation of higher priority server to determine the initial value (Equation 5.3).
$E_{5.2}$	This initial value relies on the response time calculation of the next higher priority server (Equation 5.2).
$E_{5.3}^{5.2}$	Utilising the maximum of the two previous initial values, i.e. the maximum of Equation 5.3 and Equation 5.2.
$E_{5.5}$	Selecting the maximum of a series of initial value candidates (Equation 5.5).
$E_{5.6}$	Setting the initial value to the difference between the current and the next highest priority server period (Equation 5.6).
$E_{5.7}$	Setting the initial value to the difference between the current server's period and the next highest priority server's response time (Equation 5.7).
$E_{5.9}$	Setting the initial value to half the sum of the server's period and execution capacity (Equation 5.9).
$Comb$	First, the sufficient test based on Equation 5.10 is performed and if response time analysis is required, then the initial value is determined as the maximum of the Equation 5.3, Equation 5.7 and Equation 5.9.
Θ_i^{inc}	Using the server's execution capacity as the initial value in conjunction with Algorithm 10.
$E_{5.5}^{inc}$	Selecting the maximum of a series of initial value candidates (Equation 5.5) in conjunction with Algorithm 10.

5.3.1 Experiment 7

This experiment examines the efficiency of the sufficient schedulability test that is based on the response time upper bound value (see Equation 5.10). The results show the average percentage of servers that are considered schedulable by the sufficient test belonging to a server set. Furthermore, the percentage of entire server sets that are deemed schedulable by the sufficient test is also considered.

The tests were carried out for a set of 12, 24 and 36 servers with an ITU varied from 75% to 97.5% and the server periods spanned over four orders of magnitude. The investigated ITU interval had a granularity of 2.5%. Only schedulable server sets were considered in this experiment, which is symbolically represented by the level at 100%. The obtained results are rendered in Figures 53–55 showing the percentage of servers and server sets considered schedulable, using the response time upper bound in contrast to the exact schedulability test.

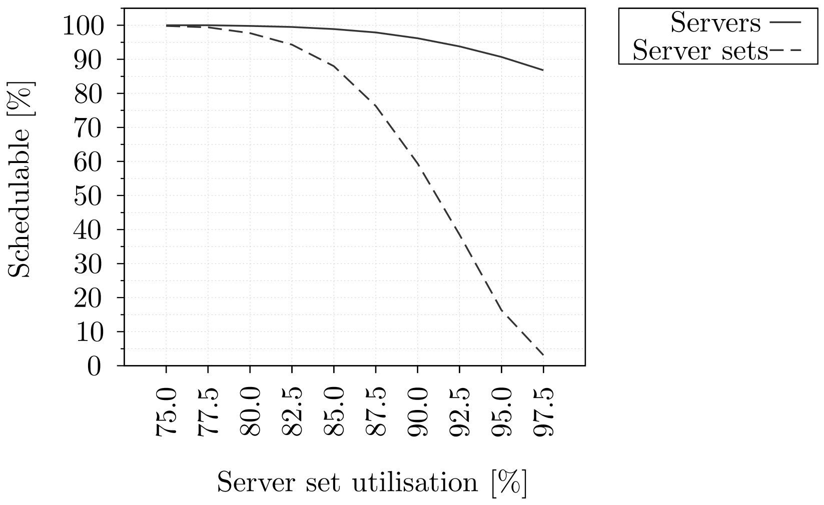

Figure 53: Percentage of schedulable servers and server sets (12 servers)

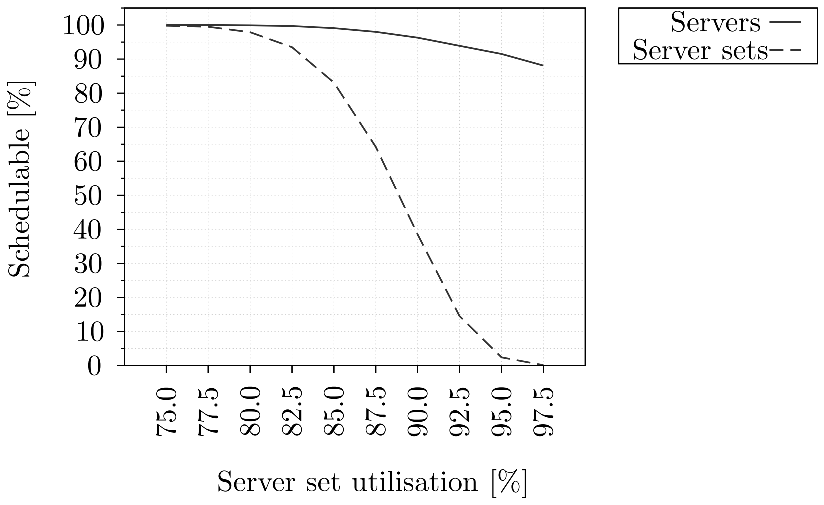

Figure 54: Percentage of schedulable servers and server sets (24 servers)

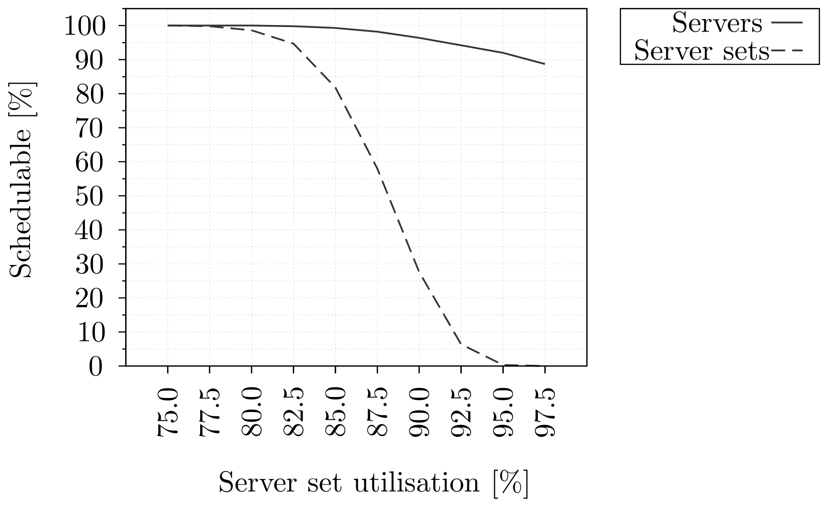

Figure 55: Percentage of schedulable servers and server sets (36 servers)

Using the response time upper bound for the schedulability test of an entire server set does not perform very well for high system utilisation values. Figures 53–55 show that for the examined test-cases the percentage of server sets considered as schedulable rapidly decreases over 80% system utilisation. At the same time, in a server set, the percentage of servers deemed schedulable by a sufficient test, is still above 85% for a system utilisation as high as 97.5%.

Thus, applying the response time upper bound based schedulability test on a per server basis can significantly reduce the number of servers that require a more exhaustive schedulability analysis by utilising the recurrence relation to calculation a server's response time.

5.3.2 Experiment 8

The main focus of this experiment is to evaluate the efficiency of the response time analysis based schedulability test using the initial values that were introduced in this chapter. The presented figures show the average number of CeilOps required by the recurrence relation to converge, the distribution of the number of CeilOps and the impact of the number of orders of server period magnitudes on the recurrence relation's convergence. The results obtained for the standard implementation of the response time calculation, using Θ_i as the initial value for the recurrence relation, is rendered in all figures of this experiment in order to simplify the comparison of the measurements.

Figure 56 shows that the best performance of the exact response time based schedulability test is achieved with the initial values calculated as $\max(E_{5.3}, E_{5.2})$.

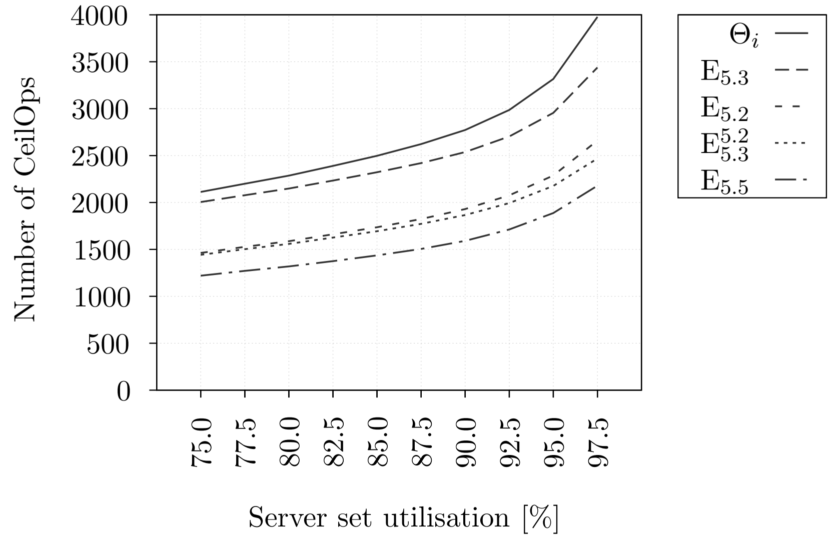

Figure 56: Average number of ceiling operations for the exact response time based schedulability test

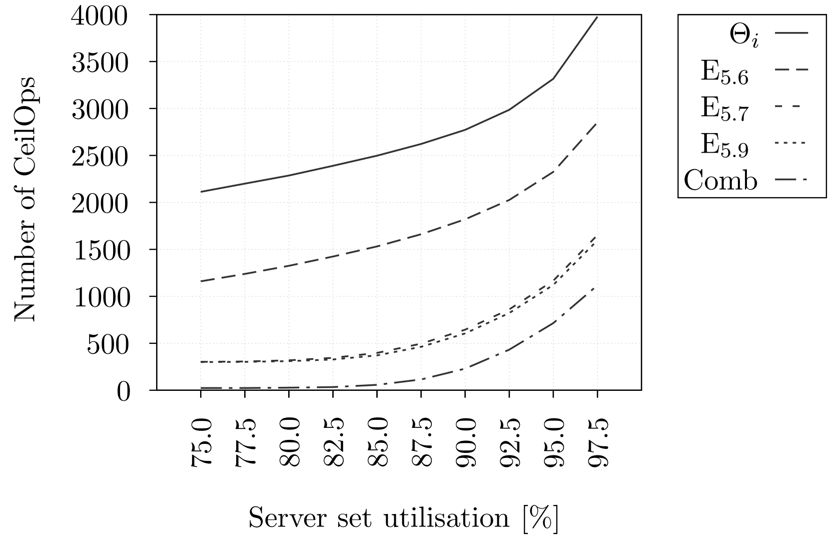

Figure 57: Average number of ceiling operations for the exact boolean schedulability test

At first glance, the initial values calculated by Equation 5.5 result in fewer number of CeilOps of the recurrence relation than for $\max(E_{5.3}, E_{5.2})$. However, taking into account the additional $n(n-1)/2$ ceiling operations required to calculate the series of initial values, raises the complexity of the response time calculation to the same level as for $\max(E_{5.3}, E_{5.2})$, eliminating the advantage of a better initial value determined by Equation 5.5.

The exact boolean schedulability test that is of main interest to us due to its intended application as an online acceptance test, reaches the best performance for the combination of different approaches. The observed efficiency of the exact boolean schedulability test is achieved with the composition of the response time upper bound test as a sufficient schedulability condition, and using the initial values calculated by Equation 5.7 and Equation 5.9 in combination with the recurrence relation to determine the response time of servers (see Figure 57). The performance improvement is a consequence of the response time upper bound calculation, reducing the number of servers in a server set to less than 15% (as observed in Experiment 7 - Figures 53–55) requiring exact schedulability analysis.

Figures 58–59 show in a histogram the distribution of the number of CeilOps for the 10,000 test-cases with 95% system utilisation. In contrast to the default approach with $R_i^0 = \Theta_i$, the distribution of the number of CeilOps required by the exact response time based schedulability test is the narrowest, and has a smaller average and maximum value for the initial values determined by Equation 5.5 and $\max(E_{5.3}, E_{5.2})$.

For the exact boolean schedulability test, the average and maximum number of CeilOps to determine the schedulability of a server set is further decreased. The most efficient behaviour of the exact schedulability test is when the *Comb* method, as a combination of the schedulability test simplification and new initial values, is used. We note that the initial peak on the *Comb* method graph accounts for the server sets where the sufficient schedulability test is adequate to determine schedulability.

Although it is unlikely that, as opposed to tasksets, servers have a wide range of periods, the effect of different orders of server period magnitudes on the performance of the schedulability test is examined and the results are depicted in Figures 60–61. The graphs show that for the exact response time test, the effort of determining the response times and the schedulability of a server set

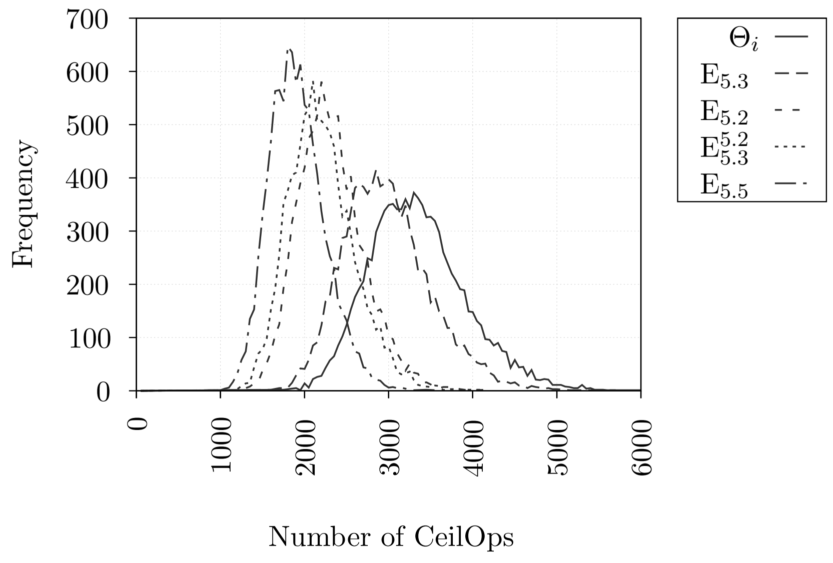

Figure 58: Frequency distribution of number of ceiling operations for the exact response time based schedulability test

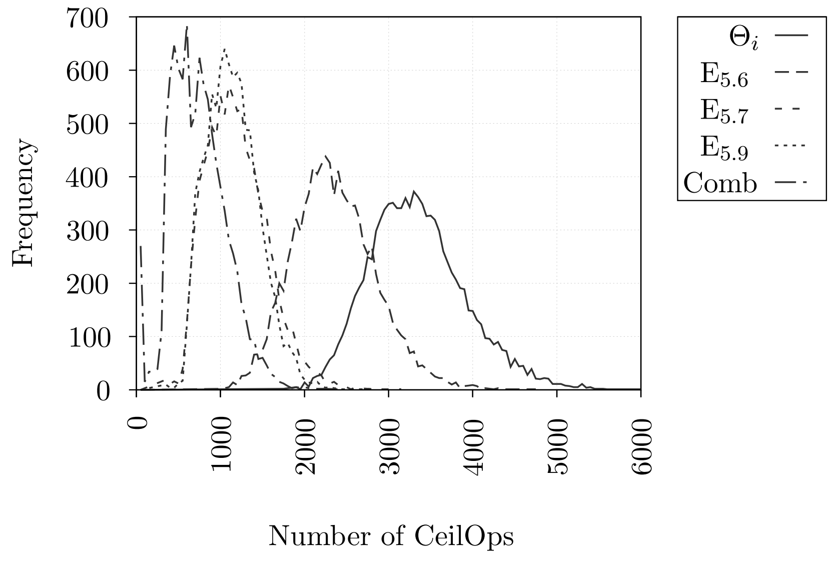

Figure 59: Frequency distribution of number of ceiling operations for the exact boolean schedulability test

increases with an increasing range of server periods. While for the exact boolean schedulability test, using the corresponding initial values ($E_{5.6}$, $E_{5.7}$, $E_{5.9}$), the number of CeilOps remains approximately constant irrespective of the range of server period values. Using the *Comb* method, the number of CeilOps decreases with increasing range of server period values. This behaviour emerges in fixed-

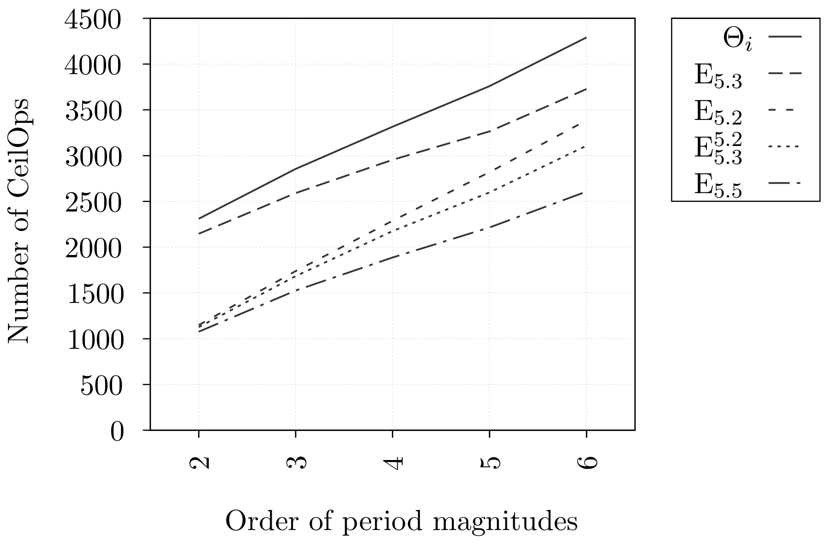

Figure 60: Server period orders of magnitude dependent average number of ceiling operations for the exact response time based schedulability test

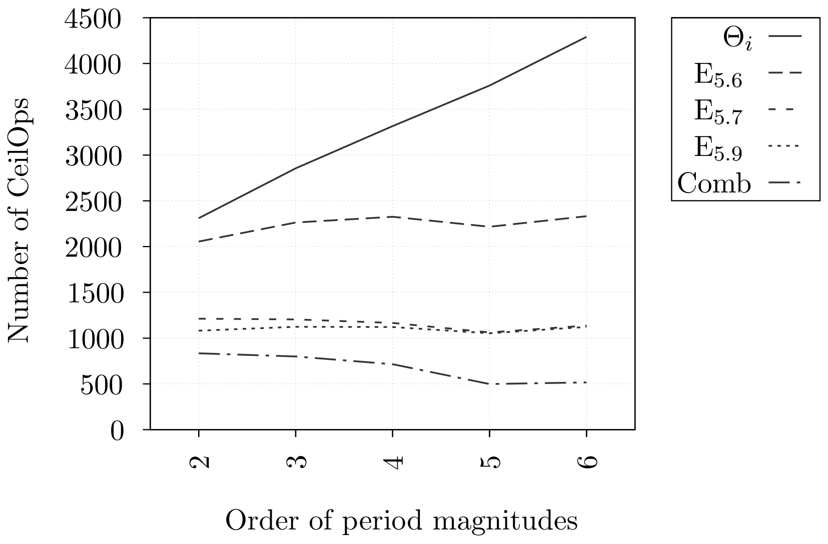

Figure 61: Server period orders of magnitude dependent average number of ceiling operations for the exact boolean schedulability test

priority scheduled systems since it is easier to schedule servers with a wider range of period values and therefore the sufficient test becomes more accurate.

The last part of this experiment is concerned with the effect of the server set size on the performance of the schedulability test. For this test we varied the

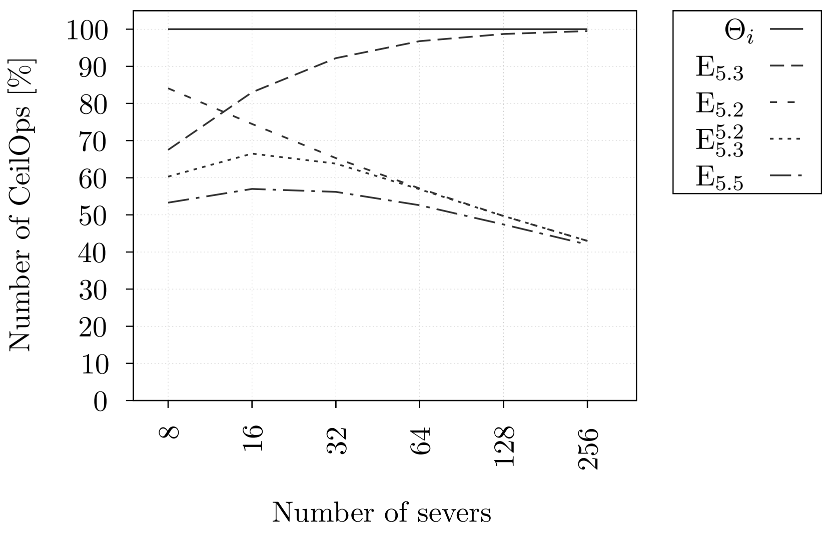

Figure 62: Server set size dependent average number of ceiling operations for the exact response time based schedulability test

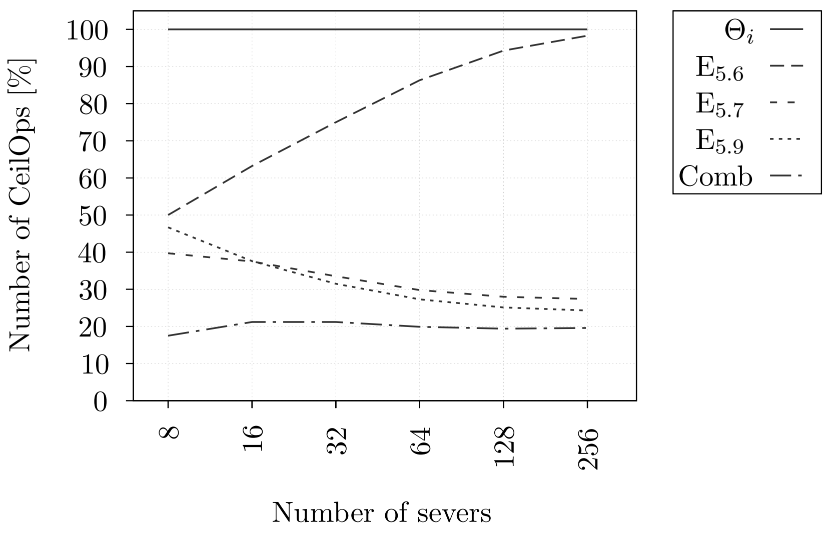

Figure 63: Server set size dependent average number of ceiling operations for the exact boolean schedulability test

number of servers in a server set from 8 up to 256 with system utilisation set to 95% and using four orders of server period magnitudes. It should be noted that as the number of servers in the system increases, the utilisation and the execution capacity of each server decreases. In Figures 62–63 the performance of the schedulability test utilising different initial values is compared with the measurements for the default initial value as a reference.

For the exact response time test Figure 62 indicates that initial values calculated by Equation 5.5 progressively improve the performance of the recurrence relation as the number of servers increase. This can be explained by an increasing number of initial values in each series generated by Equation 5.5. From the potential number of initial values at least one is a close approximation to the actual response time, hence significantly reducing the number of CeilOps.

In Figure 63 it can be observed that the improvement, obtained by the replenishment period based initial values, decreases with an increased number of servers. As the number of servers increases while the system utilisation is maintained at the same level, the difference between the replenishment period of two servers decreases and results in poor performance of the response time calculation. To the contrary, initial values considering the server's replenishment period and execution capacity perform better for the increasing number of servers at maintained system utilisation level, since the execution capacity of each server decreases in this case. However, the best performance is achieved again by the *Comb* method.

In summary, with regards to the online acceptance test, the most efficient optimisation steps are the use of the response time upper bound as a sufficient schedulability test in order to decide if a more exhaustive exact schedulability analysis is required. If the response time upper bound based test fails, an exact schedulability test is carried out by using the $\max(E_{5.3}, E_{5.7}, E_{5.9})$ as an initial values for the recurrence relation.

5.3.3 Experiment 9

Additionally to the standard implementation of the response time calculation, in Section 5.2.1 we proposed an implementation that improves the convergence of the recurrence relation. The tests conducted for this part of the experiment do not include the response time upper bound based optimisation of schedulability test, introduced by Equation 5.10.

For the sake of simplicity, only the default and the series of initial values defined by Equation 5.5 are used for the evaluation of this experiment. Figure 64 shows the performance results for the standard and incremental implementation of the recurrence relation on the basis of these two initial values. The graphs indicate

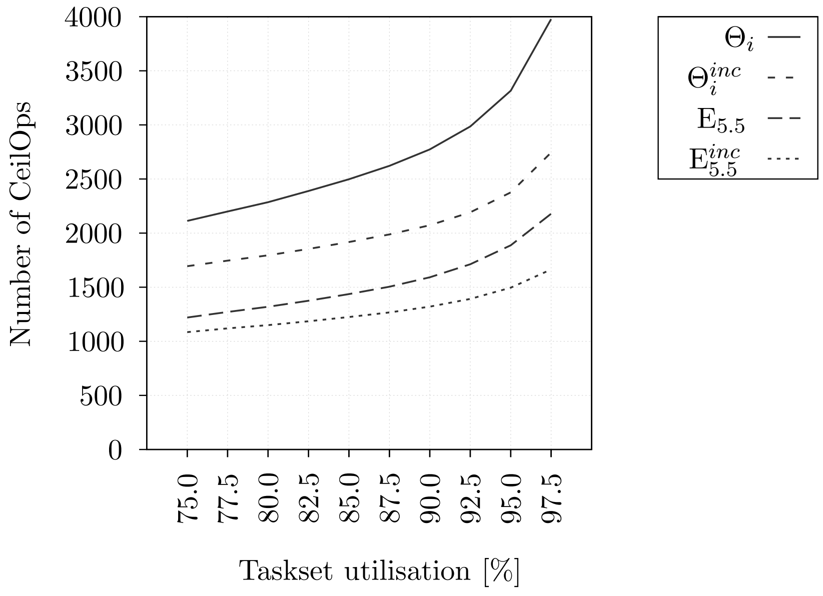

Figure 64: Comparing the implementation effect on the performance of the response time calculation

that on average, the incremental implementation (as defined by Algorithm 10) requires a smaller number of CeilOps to converge to the response time. However, the next experiment will show the whole effect of the incremental implementation on the performance of the response time calculation.

5.3.4 Experiment 10

In the final experiment of this chapter, the more abstract performance values are complemented by real execution time measurements, captured on an embedded board. The purpose of this experiment is to provide empirical evidence for the usefulness of optimisation proposals presented in this chapter.

In contrast to the evaluation on a desktop computer only a limited number of experiments could be performed on the used embedded platform due to the reduced size and capacity of hardware resources.

The performance measurements were carried out for server sets with an ITU of 75%, 85% and 95%. For each ITU setting 3,000 different random server sets, with 24 servers in each set, were generated. The server period values were distributed over four orders of magnitude. At each ITU, from the 3,000 examined server sets the worst-case value with the largest number of CeilOps for the schedulability test was selected to link the number of CeilOps with real execution time values. These worst-case values were obtained for the schedulability test starting with the default initial value.

Table 6 contains the execution times that were captured for the schedulability test starting with the different initial values considered in this chapter. Additional to the execution time of the schedulability test, the time required for the calculation of the various initial values is also listed. The sum of both execution times, the initial value calculation and the schedulability test, provides a critical argument in the selection of the most efficient approach for an online acceptance test.

Table 6: Associating the number of ceiling operations with execution time measurements

Initial value	Ceiling operations	Initial value (clk)	Sched. test (clk)	Total (clk)
Θ_i	6324	1371	163186	164557
$E_{5.3}$	5724	4390	147751	152141
$E_{5.2}$	3867	1688	100086	101774
$E_{5.3}^{5.2}$	3611	5179	93541	98720
$E_{5.5}$	3094	24548	80275	104823
$E_{5.6}$	4357	1688	112494	114182
$E_{5.7}$	1609	1688	41962	43650
$E_{5.9}$	1573	1587	41010	42597
Comb	722	3051 + 615	18516	22182

Table 6 shows that the response time based schedulability test is faster if the series of initial values, $E_{5.5}$, is used in contrast to $\max(E_{5.3}, E_{5.2})$. However, this performance improvement is lost, in fact the overall execution time of the schedulability test is worse for $E_{5.5}$ than $\max(E_{5.3}, E_{5.2})$ if the time required by the calculation of the corresponding initial values is taken into account as well.

For the exact boolean schedulability test, the method denoted as *Comb* in Table 6 required 3051 plus 615 clock cycles to calculate the initial value. We consider at this point the response time upper bound calculation consuming 3051 clock cycles for the 24 servers in the set, and the 615 clock cycles for the two lowest priority servers that fail the response time upper bound based sufficient

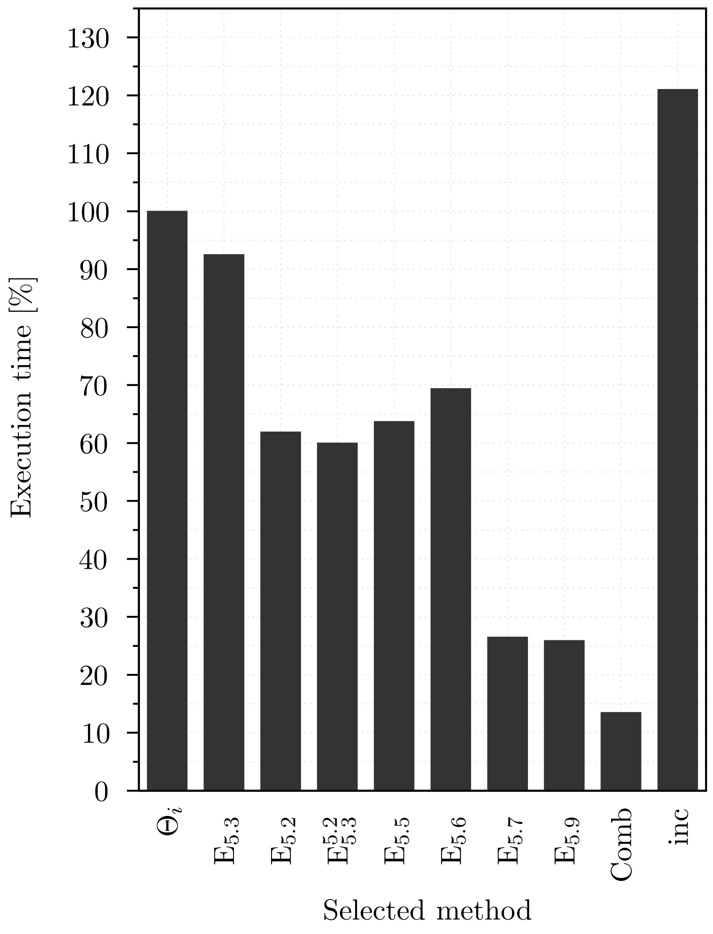

Figure 65: Execution time of response time calculation in contrast to classical approach

test, hence requiring initial values for the more time consuming response time analysis.

Figure 65 shows that the server period based initial values significantly reduce the execution time of the schedulability test. In combination with the response time upper bound test, denoted as *Comb*, the execution time of the schedulability test can be reduced to 13.5% of the time required by the approach using the default initial value. In other words, this is equivalent to the reduction of the execution time by a factor of approximately 7.5 in contrast to the default approach.

On the embedded platform, the performance of the standard (Algorithm 9) and the incremental (Algorithm 10) implementation of the response time calculation was evaluated. For the server sets with 95% ITU the number of the second *for*-loop iterations (line 9–13 in Algorithm 10) was reduced to a value between 68% and 81% in contrast to the standard implementation. However, the execution time measurements showed that the execution time of the incremental implementation was between 104% and 121% higher than for the standard implementation. The measurements revealed that on the used hardware platform the execution time of the *for*-loop in the standard implementation was 26 clock cycles and in the incremental implementation it increased to 39 clock cycles due to the additional store operation of the higher priority interference value. Though, it is possible that for different complier and hardware platforms the execution time difference between the standard and incremental implementation would change. Nevertheless, the presented results point to the importance of doing real measurements on the target hardware, to support the selection of effective optimisation steps.

5.3.5 Evaluation summary

The online acceptance test utilising the response time analysis is a necessary component of open real-time systems. The exact response time, more specifically the exact completion time of the server capacity supply, is not relevant for the purpose of the acceptance test. Therefore only an exact boolean schedulability test is required in order to determine whether a server set is schedulable or not.

The empirical evaluation indicates that the best performance for the exact boolean schedulability test can be achieved if a sufficient test, like the response time upper bound based test is executed to determine the schedulability of a server and an exact analysis is performed only if necessary. The experiments show that the schedulability of a large number of servers can be determined on the basis of the response time upper bound value. Furthermore, for the exact schedulability analysis the initial values presented in Equation 5.7 and 5.9 can significantly reduce the number of CeilOps required by the recurrence relation to converge to the response time value of the considered server.

However, the incremental implementation of the response time calculation as an alternative approach does not provide any benefit on the used embedded hardware platform. Most probably it will not lead to any performance benefit on most of the other platforms either, due to the additional memory write operations in the second *for*-loop. Therefore, it is preferable to use the standard implementation with the aforementioned response time upper bound and initial value enhancements.

5.4 Summary

With the focus on open real-time systems we are interested in an acceptance test that can determine if the set of servers established after the submission of a new application could be feasibly scheduled or not. That means, in the new server set, each server can supply its full capacity to the associated tasks before the server's next periodic replenishment event occurs. Such an acceptance test must only implement an exact boolean schedulability algorithm that provides accurate information about the schedulability of the servers in the system, but not about their exact response times.

The evaluation of the various optimisation trials showed, if an exact boolean schedulability result is sufficient then the best performance can be achieved if the response time analysis is supplemented by a test that can efficiently determine the response time upper bound. That means, the exact schedulability analysis using the response time calculation is carried out only if the sufficient schedulability test using the response time upper bound fails.

In the presence of flexible applications we are also concerned with the efficient distribution of the processing resource among the running applications. Therefore, the acceptance test is considered as an integral part of a more complex module that is also responsible for the distribution of spare processor capacity and not as a standalone component. A composite spare capacity distribution algorithm, containing the online acceptance test, is the topic of the following chapter where it will be defined and evaluated.

Chapter 6

Runtime server parameter adaptation

In embedded real-time systems being developed today, it is common to find requirements for flexibility and support for dynamic behaviour that are key driving factors in the design of their architectures and scheduling methods. Manufacturers of these embedded and resource constrained systems are faced with the difficulty of providing guarantees on the real-time behaviour of their applications while at the same time handling flexibility and dynamic changes to the applications being executed. Traditional real-time scheduling focuses on worst-case behaviour and using it for static configurations implies that a large amount of capacity, that is rarely required, is statically allocated in order to manage dynamic system changes. This conflicts with the common requirement to be able to use the maximum possible amount of the available resources.

In this context of requirements for flexibility and support for dynamic behaviour it is possible to design applications that adapt their processing to the assigned resources, adjusting the quality of the response to the available resources. In a system developed with this adaptivity in mind, it is possible to maximise resource usage while trying to provide the best possible QoS.

The complexity of modern embedded systems is also driving the need to independently develop applications or application components that may join and leave the system during runtime. The available processing capacity should be dynamically adapted to these changing situations. In most platforms these dynamic changes may be frequent, but not as fast as the regular application periods. We may have new applications that stay in the system for seconds or minutes, while their own internal periods may be in the milliseconds range.

Since the adaptation of server parameters to the available processing resource is an online task, there will be a trade-off between the selection of server parameters that maximise the QoS and the maximal affordable processing time for this task.

These mixed application and system requirements give us the motivation for an online anytime algorithm that is capable of distributing the spare capacity available on the processor. Such an algorithm is introduced and defined in the following sections.

6.1 Spare processor capacity

In this chapter we address long term major changes of the system state, like the submission or start of applications upon certain events, but also the termination of applications after the assigned processing tasks have been finished. In order to increase the processing resource utilisation in contrast to static partitioning, we exploit the flexibility of applications, and adapt and distribute the processing resource according to certain attributes. The algorithm introduced for this purpose is denoted as the *Spare Capacity Distribution (SCD)* algorithm.

The SCD algorithm is based on the idea of maximising the utilisation of the processing hardware resource, while optimising the QoS of the applications. Nogueira and Pinho [76] introduced a related anytime algorithm in the context of distributed real-time systems. Their algorithm allows to determine for each application a set of processing nodes and the corresponding processing capacity reservations in the system that can be collectively used in order to maximise QoS output of the applications. The reservation of processing capacity and nodes for each application is evaluated based on general QoS attributes (e.g. frame rate and colour depth for audio and video processing, etc.). In contrast to this, the SCD algorithm approaches the spare capacity distribution problem at lower level by utilising the server's *importance* Im_i and *weight* We_i attributes [4, 47]. These attributes are part of the application contract (see contract model in Section 2.4.4) and allow the applications to influence the outcome of the spare capacity distribution.

These two parameters, importance and weight, will be set by the system designer or integrator before the application is submitted into the system. Importance and weight will be used during the runtime to determine the processor's spare capacity distribution among the active servers in the system by defining their significance to the system and relative to each other.

The precedence, in which the spare processor capacity is assigned to servers is determined by the importance level Im_i. For the spare capacity distribution, servers with the same importance level are logically combined into groups. A group $\mathbb{S}_l$ is the set of all servers having the same importance level l (see Equation 6.1).

$$\mathbb{S}_l = \{S_i \in \Gamma | Im_i = l\} \tag{6.1}$$

The weight attribute We_i influences the fraction of the spare processor capacity that a server will get at the considered importance level. The fair share value [52] denoted as H_i is used as a factor to determine the fraction of additional capacity that a server S_i will get when a certain amount of spare capacity is distributed at the currently examined importance level Im_C (see Equation 6.2). In this equation, Im_j denotes S_j's importance level.

$$H_i = \frac{We_i}{\sum\limits_{\forall j: Im_j = Im_C} We_j} \tag{6.2}$$

The general idea of the SCD algorithm is to maximise the processing resource utilisation by modifying the temporal attributes of servers during runtime (i.e. the execution capacity and replenishment period, and as a consequence their priorities as well). These modifications are carried out with the constraint that the entire system remains schedulable, i.e. temporal requirements are not violated.

We use bisection in order to search for a feasible spare capacity distribution. Defining the search as an incremental process, provides the foundation for a simple and efficient implementation of the presented algorithm in a runtime framework. Furthermore, utilising the efficient response time analysis (presented in Chapter 5) to test for schedulability, no schedulability losses are incurred by the test itself since this method is known to be exact.

We note that changes to the set of the servers in the system are only permitted if the changed set remains schedulable using the minimal temporal requirements of the servers.

The transition to a system state where the new temporal values and priority ordering can be applied and used, will be performed at a feasible time instant (for example at an *idle instant* [98], or a time instant defined by the idle instant optimised protocol [49]). The exemplary application skeleton presented in Section 2.4.5 uses for instance the idle instant [98] to coordinate the temporal parameter changes.

The topics presented in previous chapters provide fundamental building blocks to enable the realisation of an efficient runtime algorithm for server parameter selection and consequently exploit the processor's spare capacity.

6.2 Spare capacity distribution algorithm

The exploitation of the processor's spare capacity in fixed-priority scheduled open real-time systems is the main topic of this section. We introduce an anytime algorithm as part of the online system functionality, focusing on the efficiency and effectiveness in the design of the algorithm. The intention of the SCD algorithm is to allocate as much as possible of the processor's spare capacity, expressed as processor utilisation U_s, to servers in the system. An implicit constraint for the SCD algorithm is that the set of servers with their prospective new temporal parameter values is still schedulable.

As indicated in Section 6.1, spare capacity distribution is an incremental process, assigning additional processor capacity only to servers at an examined importance level Im_C. The objective of the SCD algorithm is to find a feasible amount of spare processor capacity, expressed as processor utilisation, that can be distributed at once among the servers at the processed importance level.

Since the presented algorithm is a search based approach, feasibility of various utilisation probe values U_p for the available processor spare capacity need to be tested. An efficient approach to find a feasible probe value

$U_p \in \{0, 0 + \delta U_p, \dots, U_s - \delta U_p, U_s\}$ is provided by bisecting the interval $[0, U_s]$ and applying the binary search algorithm to it. The binary search algorithm is fast, has minimal memory requirements and the required processing resources are also low.

For a given set of servers, the maximal distributable spare capacity can be found within $1 + \lceil log_2 N \rceil$ iterations, where N is the number of potential values for U_p. Given the granularity δU_p of the interval $[0, U_s]$, the number of potential values for U_p can be calculated as $N = U_s/\delta U_p$. For example, a typical value of 1% for δU_p and the possible maximum value for U_s of 100%, limits the number of potential values N for U_p to 100. By applying the binary search on the 100 possible values, a feasible U_p can be determined by checking the schedulability of $1 + log_2 100 = 8$ different spare capacity distribution scenarios.

A server S_i is defined to be available/active for spare capacity distribution if it is able to increase its utilisation due to the following conditions:

- during the last iteration of the spare capacity distribution, S_i did not render the system unschedulable,
- S_i has not reached its predefined maximal utilisation and can therefore utilise a higher spare capacity allocation.

Each major iteration of the SCD algorithm is limited to a group $\mathbb{S}_l$ of servers. Usually several major iterations of the SCD algorithm are required until a solution is found for the given set Γ of servers.

In order to determine the optimality of the SCD algorithm the following assumption[6] is made: The spare capacity supplied at higher importance levels is considered infinitely more valuable than at lower importance levels. This implies that different importance levels are incomparable. The SCD algorithm starts with the spare capacity distribution at the highest importance level. Spare capacity is supplied by decreasing the periods and increasing execution capacity of servers at this importance level. Only after all possible spare capacity has been allocated at the highest importance level, by exploiting the possible temporal parameter ranges of the servers, does the algorithm consider the next highest level and so on. Hence, under the assumption that importance levels are incomparable, the SCD algorithm provides the optimal spare capacity distribution.

[6]The assumption is partly motivated by the objective of the FRESCOR project.

As the SCD algorithm evaluates different spare capacity distributions, the exact schedulability test for fixed-priority scheduled uniprocessor real-time systems [32] (introduced and evaluated in Chapter 5) is utilised.

The schedulability test in the SCD algorithm has two main tasks. First, before the SCD algorithm is applied to the servers in the system, the schedulability using their minimal timing requirements is ensured. The second task is to determine the schedulability of the servers in the system during the execution of the SCD algorithm, using different utilisation probe values U_p for the spare capacity distribution.

In the remainder of this section, we describe in detail the proposed online spare capacity distribution algorithm.

As input, the SCD algorithm requires a priority ordered list of servers along with their temporal attributes. This *priority ordered list ($\mathbb{P}$)* contains servers that want to benefit from the additional assignment of spare capacity. $\mathbb{P}$ is also used as the output list.

The SCD algorithm starts with the servers' temporal parameters, set to their minimum timing requirements. The minimum timing requirement is either the minimum execution capacity $\Theta_{i(min)}$ and largest period $\Pi_{i(max)}$ in the case of a continuous server, or (Θ_j, Π_j) pair with the smallest utilisation in the case of a discrete servers. A soon as the SCD algorithm finds an intermediate or final solution, the new temporal values of the servers are stored in $\mathbb{P}$.

With the SCD algorithm, at each importance level, the search for the largest feasible U_p is carried out in order to determine a feasible spare capacity distribution (see Algorithm 11).

Each major iteration of the SCD algorithm is limited to servers at a particular importance level, starting at the highest level.

The steps of the SCD algorithm that are executed at every importance level are as follows:

1. Calculate the fair share values.

   The fair share value H_i of every active server S_i at the currently processed importance level is calculated using Equation 6.2. This value is used to determine the fraction of capacity that will be assigned to the active servers.

2. Search for a feasible spare capacity distribution.

   The algorithm considers only servers that are active (i.e. capable of accepting more than their current utilisation) at the currently examined importance level. Using binary search (represented by the *while*-loop condition in Algorithm 11), the distributable spare capacity U_p is narrowed down to the largest value at which the system is schedulable, but where an additional amount (δU_p) of spare capacity assignment (i.e. $U_p + \delta U_p$) would lead to an infeasible schedule.

   For each utilisation probe U_p, the temporal parameters of active servers at the currently examined importance level are recalculated using Algorithm 12, where $U_{i(new)}$ denotes the increased utilisation of S_i by ΔU_i.

   $$\Delta U_i = U_p \cdot H_i \tag{6.3}$$

   We note that the SCD algorithm primarily uses the server utilisation values in order to determine a feasible spare processor capacity distribution. The specific server temporal parameter values are calculated only after the spare capacity is distributed in terms of server utilisation.

3. Increase the utilisation of all active servers.

   If a schedulable spare capacity distribution has been found, the new server temporal parameters are stored in the output list $\mathbb{P}$ regardless of whether these values are intermediate or final.

One possible approach to determine the execution capacity and period of a continuous server S_i (with an increased utilisation equal to $U_{i(new)}$) is presented in Algorithm 12. If possible, the server's smallest period $\Pi_{i(min)}$ is used and its capacity is calculated accordingly. If it is not feasible to use its smallest period, then the minimum capacity $\Theta_{i(min)}$ is fixed first and the period is calculated accordingly. Under both circumstances, the calculated values are restricted to the specified interval for the capacity and the period, respectively.

For a discrete server, the utilisation values of its different discrete (Θ_j, Π_j) parameter pairs are calculated. Then the (Θ_j, Π_j) pair with the biggest utilisation value, which is less than or equal to $U_{i(new)}$, is selected.

```
   InOut: $\mathbb{P}$: Priority ordered list of servers
1  foreach importance level $l$ (in decreasing order) do
2      Determine $H_i$ for all active servers $S_i^l := S_i : S_i \in \mathbb{S}_l$;
3      while not all possible $U_p$ values checked do
4          Calculate $\Delta U_i$ (see Equation 6.3) and increase the current
           utilisation of $S_i^l$ to $U_{i(new)}$ by $\Delta U_i$;
5          Determine $S_i^l$ new parameters using Algorithm 12;
6          Determine new priority ordering for the schedulability test;
7          if all servers are schedulable then
8              Store new priority ordering and temporal values of all active
               $S_i^l$ in $\mathbb{P}$;
9          end
10     end
11 end
```

Algorithm 11: Spare capacity distribution

```
1 if $(\Theta_{i(min)}/\Pi_{i(min)}) > U_{i(new)}$ then
2     $\Theta_i = \Theta_{i(min)}$;
3     $\Pi_i = \min\big((\Theta_{i(min)}/U_{i(new)}),\ \Pi_{i(max)}\big)$;
4 else
5     $\Pi_i = \Pi_{i(min)}$;
6     $\Theta_i = \min\big((\Pi_{i(min)} \cdot U_{i(new)}),\ \Theta_{i(max)}\big)$;
7 end
```

Algorithm 12: Execution capacity and replenishment period calculation

The spare capacity distribution achieved by Algorithm 11 can be, however, further improved. Therefore, we denote Algorithm 11 as coarse-grained, and introduce an extended version of the previously defined algorithm, denoted as the fine-grained SCD algorithm.

The core component of both SCD algorithms is the search for a single utilisation value that can be distributed at one go among a selected set of servers. Additional to this, the fine-grained SCD algorithm selectively deactivates servers in the group $\mathbb{S}_l$, to which the spare capacity distribution is applied. By doing this, the SCD algorithm has finer control over the utilisation of individual servers. In contrast to the coarse-grained SCD algorithm, the fine-grained version additionally increases the utilisation of individual servers. The pseudo-code for the fine-grained SCD algorithm is depicted in Algorithm 13.

```
InOut: $\mathbb{P}$: Priority ordered list of servers
1  Initialise temporary storage $\mathbb{P}'$ with $\mathbb{P}$;
2  foreach importance level $l$ (in decreasing order) do
3      while at least one server in $\mathbb{S}_l$ is active do
4          Determine $H_i$ for all active $S_i^l$;
5          Determine $U_p$ and store in $\mathbb{P}'$ the intermediate server parameter
           values that were calculated based on $U_p$;
6          if $U_p$ is feasible using values from $\mathbb{P}'$ then
7              Increase utilisation of all active $S_i^l$ by $\Delta U_i$;
8              foreach active $S_i^l$ do
9                  Save $S_i^l$'s current temporal attributes;
10                 Increase $S_i^l$'s utilisation by $\Delta U_i^b$;
11                 if schedule is infeasible then
12                     Restore $S_i^l$'s previously saved temporal attributes and
                       deactivate it;
13                 end
14             end
15             Store new temporal values in $\mathbb{P}$;
16         else
17             foreach active $S_i^l$ do
18                 Deactivate $S_i^l$;
19             end
20         end
21     end
22 end
```

Algorithm 13: Fine-grained spare capacity distribution

The steps executed at every importance level by the coarse-grained version of the SCD algorithm are extended by an additional step for the fine-grained version. The four major steps of the fine-grained SCD algorithm are listed below with a detailed description of the additional step:

1. Calculate the fair share values (Algorithm 13 - line 4).
2. Search for a feasible spare capacity distribution (Algorithm 13 - lines 5–6).
3. Increase the utilisation of all active servers (Algorithm 13 - line 7).
4. The fine-grained SCD, in addition to the previous steps, selectively increases the server utilisation (Algorithm 13 - lines 8–14).

In the case that the distribution of U_p spare capacity is feasible but the magnitude of $U_p+\delta U_p$ spare capacity leads to an infeasible schedule, usually means that the infeasible schedule was caused only by a subset of the active servers. This implies that other active servers could make use of even higher spare capacity assignments if the few servers that led to an infeasible schedule were not considered further in the spare capacity distribution. Therefore, after U_p amount of spare utilisation has been distributed, one at a time, each server's utilisation increase is overwritten by ΔU_i^b as depicted in Equation 6.4.

$$\Delta U_i^b = (U_p + \delta U_p) \cdot H_i \tag{6.4}$$

For each individual server at the examined importance level, if its utilisation increase by ΔU_i^b does not lead to an infeasible schedule, the increased server utilisation is retained. Servers that cause a schedule failure will be deactivated for further spare capacity distribution and they will restore their utilisation increment to the value which was calculated using the value U_p instead of $U_p + \delta U_p$.

We note that the usage of shared resources and the consequent blocking factors have no direct impact on the SCD algorithm itself. Therefore, there are neither assumptions nor restrictions on the usage of shared resources, since the corresponding blocking factors affect only the schedulability test.

For the sake of clarity, the functionality of the SCD algorithm, the impact of the different server attributes on the final spare capacity distribution is illustrated with a specific example in the following section.

6.3 Example

The functional principle of the SCD algorithm is demonstrated based on the fine-grained version since it also includes all steps of the coarse-grained version.

The numerical values were chosen such that emphasising certain aspects of the SCD algorithm and the related server attributes is facilitated.

The example contains four servers in the system (three discrete and one continuous, see Table 7). All four servers have the same importance level. Using the server replenishment periods, the server priorities are determined according to the rate-monotonic priority scheme.

Table 7: Temporal attributes of servers

S_i	Θ_{min}	Π_{max}	Θ_{max}	Π_{min}	We_i	Continuous / Discrete Type	Discrete Temporal Attributes (Θ_j, Π_j)
1	50	500	100	200	2	Continuous	N/A
2	100	1000	200	300	1	Discrete	(100,1000); (100,800); (150,750); (175,725); (200,500); (200,300)
3	50	1000	100	1000	1	Continuous	N/A
4	50	1000	200	200	1	Continuous	N/A

The first column in Table 7 shows a unique identifier for each server. It is used to identify the servers as the SCD algorithm progresses.

Furthermore, to show the effect of the weight attribute on the spare capacity distribution, corresponding values have been specified (see Table 7). All servers, except for S_1, have the same weight and will get the same amount of utilisation increment. Since the weight of S_1 is twice as big as the weight of the other servers, this server will receive double the amount of utilisation increment compared to the other servers in the system.

After the schedulability of the server set has been determined by using the minimal resource requirements of the servers, i.e. $(\Theta_{min}, \Pi_{max})$, Table 8 illustrates the changes of the current execution capacity Θ_{cur} and period Π_{cur} of each server while the SCD algorithm distributes the available spare capacity. The first column shows the largest feasible value of U_p (i.e. the amount of spare capacity) determined by the binary search algorithm. That means, a valid schedule is obtained by the SCD algorithm if U_p is distributed among the servers.

The initial utilisation of the system using the minimal temporal requirements of the four servers is 30% (see Table 7). The first feasible U_p value that is found by the binary search algorithm is 56%. This is the amount of spare utilisation that can be safely distributed among the active servers. The column with the

Table 8: Spare capacity distribution example

U_p	S_i	P_i	H_i	ΔU_i	Effective utilisation increase $\left(\Delta U_i^b\right)$	Θ_{cur}	Π_{cur}	Resulting resource utilisation	Active for SCD
56%	1	1	0.4	22.4%	22.55%	100	235	89%	No
	2	3	0.2	11.2%	10.00%	150	750		Yes
	3	4	0.2	11.2%	5.00%	100	1000		No
	4	2	0.2	11.2%	11.45%	50	304		Yes
8%	1	1	N/A	N/A	N/A	100	235	93.6%	No
	2	3	0.5	4.00%	0.00%	150	750		No
	3	4	N/A	N/A	N/A	100	1000		No
	4	2	0.5	4.00%	4.56%	50	238		Yes

title ΔU_i shows the calculated utilisation increase based on U_p for each server. It can be observed that the calculated utilisation increase for S_1 is twice as big as for the other servers as specified by its *weight* attribute.

An increase of one or more server's ΔU_i value by δU_p will lead to an infeasible schedule. Which servers render the system unschedulable is determined in the next step and they are taken out of consideration for further spare capacity distribution. The last column shows the servers that are still active for SCD after their utilisation has been increased to a certain value. In the above example with $U_p = 56\%$, S_1 is deactivated because any additional resource reservation for S_1 would result in an infeasible system schedule. S_3 is also deactivated, however, due to reaching its maximal utilisation.

The next feasible U_p value that is found by the binary search algorithm is 8%. At this point only S_2 and S_4 are left over for spare capacity distribution. Since only these two servers are considered by the SCD algorithm, their fair share values are recalculated. This time, it is S_2 whose utilisation increase causes an infeasible schedule and it will be deactivated for the next run of the SCD. Since it is a discrete server, the utilisation increase to its next utilisation boundary suddenly consumes a bigger amount of processor utilisation and leads to an unschedulable system.

In the last step, the SCD algorithm tries to increase the utilisation of S_4. However, even the smallest increase of S_4's utilisation would lead to an unschedulable system. Therefore, S_4 is also deactivated for SCD. There are two reasons why S_4's utilisation cannot be increased anymore. First, the utilisation probe U_p has a certain granularity. Second, the fair share value of S_4 increased to 1.0 since it is the only remaining active server. The fair share value of 1.0 and the granularity of 1% cannot provide a utilisation increment for S_4 that is small enough to create a feasible schedule. Finally, the SCD algorithm terminates since there are no more servers left that could utilise additional spare capacity assignment without leading to an infeasible system schedule.

Table 7 also shows that the effective utilisation increase, that each server undergoes, can deviate from the calculated utilisation increase. The effective utilisation increase that each server will get, depends on various factors:

1. the utilisation increase is limited because the server reaches its maximal utilisation or,

2. in the case of discrete servers, the utilisation increase is dependent on utilisation boundaries. The calculated utilisation increase will always be less than or equal to the discrete server's next highest discrete utilisation value. The utilisation boundaries of discrete servers are determined by their predefined (Θ_j, Π_j) temporal attribute values.

Due to the finite granularity of the algorithm, the *breakdown utilisation* of a set of servers (largest utilisation of the processor, determined by the spare capacity algorithm, at which the schedule of the servers becomes infeasible) is not necessarily the largest possible value that can be determined analytically.

Furthermore, Table 7 demonstrates the anytime behaviour of the SCD algorithm. The utilised processor capacity successively improves as the SCD algorithm progresses. The first determined intermediate result is the schedulability of the server set using its minimal resource requirements. Next, a further improvement is achieved by determining server parameters that enable 89% processor utilisation. After the completion of the SCD algorithm, the server set utilises 93.6% of the processor capacity. In summary, interrupting the SCD algorithm before its completion but after the schedulability of the server set with the corresponding minimal resource requirements is determined, provides the last determined intermediate result reflecting the last determined processor capacity distribution among the active servers.

6.4 Runtime server parameter change

Temporal parameter changes of tasks within an application are usually the consequence of events that are sent from the underlying operating system or system software to the corresponding application. The change of task parameters implies that tasks adapt their behaviour to the altered parameter values, reflecting certain mode of operation.

When the SCD algorithm, as part of the system software, determines the prospective resource reservation for applications, the corresponding server parameter changes have to be coordinated in a deterministic way. That means, not just the application has to perform a mode transition as suggested in Section 2.6 and 2.4.5, but also the system has to determine a feasible time instant when server parameters can be modified and become operative without jeopardising the system schedulability.

Since the server parameter adaptation is considered as a system functionality, the corresponding software module, implementing the control logic for coordinated change of server parameters, is best accommodated in a resource managing middleware as suggested in [48] and depicted in Figure 1.

Considering the temporal specification of tasks, the deterministic change of their parameters was addressed by mode-change protocols (see Section 2.6). We noted in Section 2.3 that from the schedulability analysis point of view, servers can be treated as periodic tasks. Thus, server parameter changes are analogous to task parameter changes. That means, certain mode-change protocols developed for tasksets can also be reused to coordinate the parameter change of servers. The purpose of a predictable server parameters change, coordinated by a mode-change protocol, is to ensure that the applications' temporal constraints are maintained at all times.

Due to its simplicity we utilise the idle instant protocol [98] to implement a predictable server parameter change procedure. An implementation of the idle instant protocol is also proposed and illustrated by Crespo et al. [25]. The rationale for the choice of the idle instant protocol is that if in a follow-up work of this thesis global shared resources are also considered, the idle instant protocol can still be used to coordinate the server parameter changes. Although protocols belonging into the category of asynchronous protocols enable a faster change of task parameters upon a mode-change event, however if applied on servers, these protocols would require an extensive research, analysis and adaptation. Shared resources, for example, represent one obstacle for the unmodified reuse of existing mode-change protocols, since the applications and their use of shared resources are not known in advance in open real-time systems.

In order to complete the implementation specific details, additionally to the presented implementation example for flexible applications and tasks (see Section 2.4.5) we sketch internals of the system level component, responsible for server parameter changes.

Based on the proposal in [25], an aperiodic server with a priority lower than all periodic servers can be created. Whenever an idle instant occurs the aperiodic server is scheduled and according to the idle instant protocol it can change the parameters of all servers in the system. Additionally, the idle instant event is propagated to the applications in order to allow them the modification of task parameters and to adapt their behaviour to the new resource reservations. However, a few constraints need to be considered for the implementation:

- The server parameter changes carried out within the aperiodic server have to be atomic in order to maintain a consistent system state.
- If the system is at 100% utilisation, the idle instant is not long enough to schedule the aperiodic server and execute all mode-change related work. Therefore, the resource managing middleware has to make sure that the aperiodic server is scheduled at the occurrence of the idle instant in any case if there are pending changes that would impact the server parameter values. Furthermore, the scheduling of periodic servers has to be disabled once the mode-change related modifications are started by the aperiodic server since the *old* server parameters and schedule became obsolete. The scheduling of periodic servers can be re-enabled after the *new* server parameters are set and the mode-change related work has been completed.
- To maintain the valid behaviour of executed tasks, also in the case of a mode-change, either the idle instant of the taskset and the associated server has to coincide or the taskset has to support the asynchronous change of task parameters to enable a feasible transition to the new parameter values.

After introducing an efficient schedulability test and incorporating it into a runtime system level module to exploit spare processor capacity and to increase the processor usage, the efficiency and effectiveness of the integrated online spare capacity distribution algorithm is evaluated in the following section.

6.5 Evaluation

To evaluate the performance of the SCD algorithm, server sets were generated where the variation of particular isolated test-case parameters was examined. For each of these server sets 100,000 different configurations were created. Each configuration consists of the predefined number of servers (i.e. 5, 10, 15, . . . , 50) for which randomly generated server parameters (i.e. execution capacity and replenishment period) were created.

The approach similar to the random task parameter generation [15] (see Section 3.6.1) for a given maximal utilisation (Initial Target Utilisation (ITU)), was also used in this evaluation to generate the random server parameter values.

The ITU was chosen appropriately in order to enable the observation of the SCD algorithm's performance in different scenarios where more or less spare capacity was available. The chosen ITUs were 30%, 50% and 80%. Before the server sets were processed by the SCD algorithm, it was ensured that the generated sets were schedulable at the chosen ITU, i.e. using the minimal resource requirements for each server. Test-cases with initially unschedulable server sets were not considered in the measurement and they were replaced with a new and schedulable server set.

The period and execution capacity of a server is derived from its randomly generated utilisation value. First, the server's period is chosen according to a uniform random distribution from a randomly selected period magnitude range (i.e. $[10^3, 10^4]$, $[10^4, 10^5]$, $[10^5, 10^6]$ or $[10^6, 10^7]$). Given the server's utilisation and period, the calculation of its execution capacity is straight forward.

The server weight and importance level parameter were randomly chosen from the integer range of [1,5].

In order to examine the advantage that arises as a result of the flexibility of servers, we define an upper and lower bound on their utilisation ranges. For each server the initially generated random utilisation values (the ITUs) are considered in the tests as their lower utilisation bounds. This lower utilisation bound is used to derive the server's minimum execution capacity and maximum replenishment period. The minimum execution capacity is then multiplied and the maximum period is divided by a factor in order to determine the server parameters (i.e. maximum execution capacity and minimum period) for its upper utilisation bound. A factor of 2.0 for the 30% ITU, and a factor of 1.5 for 50% and 80% ITU was used.

For discrete servers, additional to their lower and upper utilisation bounds, a random number of intermediate utilisation values was generated. The number of intermediate values was in the range of 1 to 3.

The following experiments evaluate the data that was collected from various measurements by varying the parameters that were mentioned at the beginning of this section.

The diagrams show the progress of the spare capacity distribution as a function of the number of ceiling operations needed by the schedulability analysis. These measurements give an insight into the behaviour and complexity of the algorithm under different circumstances.

The presented experiments contain a mixed set of servers (both continuous and discrete). The only exception is the experiment in which the impact of the different temporal attribute types (i.e. discrete or continuous) on the SCD algorithm is examined.

For the sake of clarity, the diagram types that are used to support the empirical evaluation, are introduced on the basis of Figures 66–67. Of main interest are the following properties of the SCD algorithm: how fast can the spare capacity be distributed and the number of *ceiling operations (CeilOps)* required by the algorithm to terminate. The average increase of processor's utilisation as the SCD algorithm progresses, is depicted in Figure 66. The percentage of test-cases terminated by a certain number of CeilOps is depicted in Figure 67. This figure can also be interpreted as the probability that the SCD algorithm will terminate within a given number of CeilOps for a given number of servers.

For all experiments, the SCD algorithm was executed until it terminated by itself. The processor's utilisation achieved in this way is the highest schedulable utilisation of the active servers in the system.

In order to help readability, the abbreviations listed in Table 9 are used in the following diagrams of this chapter to denote the test-case configurations for the depicted results.

Table 9: List of abbreviations used in the evaluation of the runtime server parameter adaptation

Abbreviation	Description
$5, 10, \cdots, 50$	The numbers denote the server set size, i.e. the number of mixed (continuous and discrete) servers used for a specific test configuration.
$10_{itu}, 25_{itu}, 50_{itu}$	The larger number denotes the server set size and the subscript refers to the configured *Initial Target Utilisation* (i.e. *itu = 30* $\widehat{=}$ 30% ITU, *itu = 50* $\widehat{=}$ 50% ITU and *itu = 80* $\widehat{=}$ 80% ITU) of the server set.
$10_{stype}, 25_{stype}, 50_{stype}$	The larger number denotes the server set size and the subscript refers to the type of servers in the examined set (i.e. *stype = m* $\widehat{=}$ mixture of continuous and discrete servers, *stype = d* $\widehat{=}$ only discrete servers and *stype = c* $\widehat{=}$ only continuous servers in the set).
$10_{ilvl}, 25_{ilvl}, 50_{ilvl}$	The larger number denotes the server set size and the subscript refers to the assignment of servers to importance levels (i.e. *ilvl = s* $\widehat{=}$ all servers in the test are assigned to the same importance level and *ilvl = d* $\widehat{=}$ the servers in the test are randomly assigned to the 5 available importance levels).
$10_{algo}, 25_{algo}, 50_{algo}$	The larger number denotes the server set size and the subscript refers to the used *Spare Capacity Distribution* algorithm (i.e. *algo = g* $\widehat{=}$ coarse-grained and *algo = f* $\widehat{=}$ fine-grained SCD algorithm).

6.5.1 Experiment 11: Impact of the server set size

In this experiment the main focus is on the effect that the server set size has on the SCD algorithm. In Figures 66–67 we examine the progress of the processor utilisation increase achieved by the SCD algorithm and the probability for the same algorithm to regularly terminate within a give time.

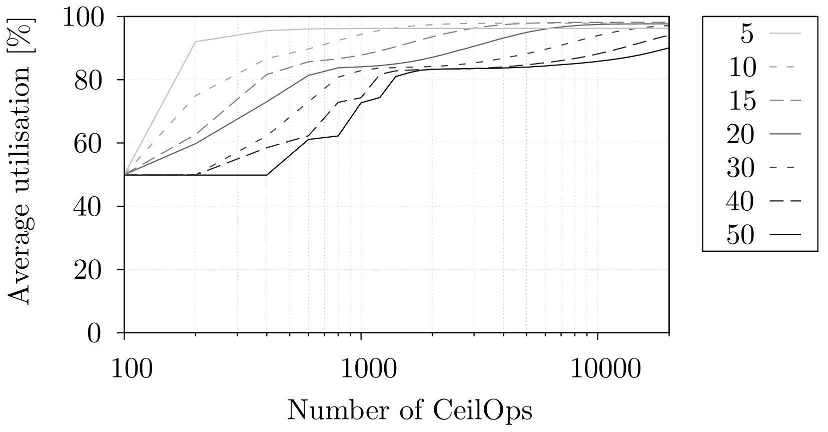

Figure 66: Server set size impact on the average processor utilisation for 5, 10, ..., 50 servers

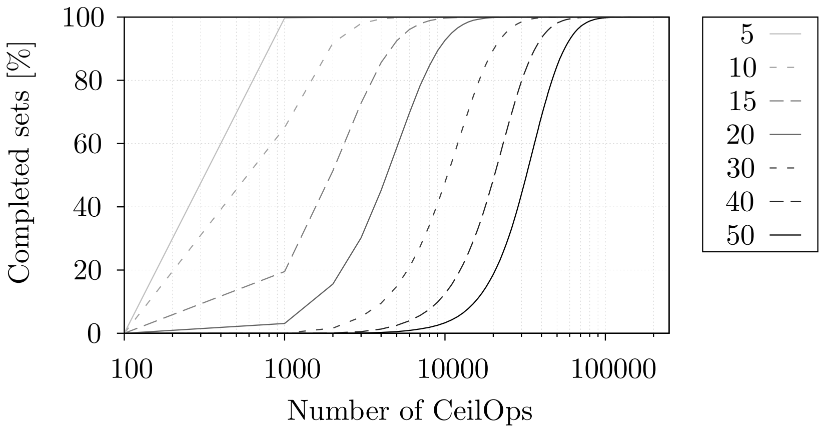

Figure 67: Server set size impact on the SCD algorithm termination rate for 5, 10, ..., 50 servers

In Figure 66, the rate of the average processor capacity exploitation, expressed as processor utilisation increase, for the test-cases starting from an ITU of 50% is depicted. The figure shows that the rate of the utilised processor capacity slows down with an increasing number of servers in the system.

Figure 67 shows that the probability for the SCD algorithm terminating regularly within a certain time interval, at this point expressed as number of CeilOps, also increases with the number of servers. The number of CeilOps will be mapped to absolute time values in a later experiment.

Supported by the diagrams, it can be stated that a larger number of CeilOps are required to find a feasible spare capacity distribution by each major iteration of the SCD algorithm, as the number of servers in the system increases. This effect can be attributed to the schedulability test examining an increasing number of servers.

6.5.2 Experiment 12: Impact of the server set's initial utilisation

In open real-time systems, that are dynamic by nature, the processor's utilisation at which applications are submitted into the system, cannot be predicted in advance. Therefore, this experiment examines the impact of the initial processor's utilisation on the progress of the SCD algorithm and the results are depicted in Figures 68–69.

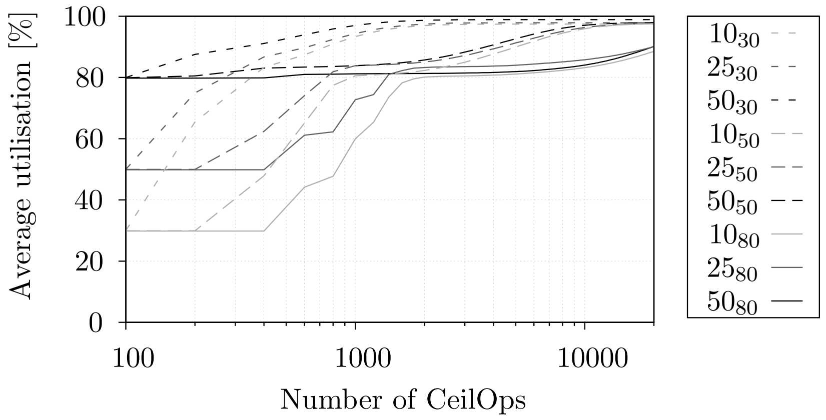

Figure 68: Impact of ITU on the average processor utilisation, starting at 30%, 50% and 80%

In Figure 68 it can be observed that the ITU has an effect on the processor's utilisation increase only at the beginning of the SCD algorithm. That means, after a certain number of CeilOps, at smaller ITU a larger chunk of spare processor capacity can be distributed without violating the schedulability of

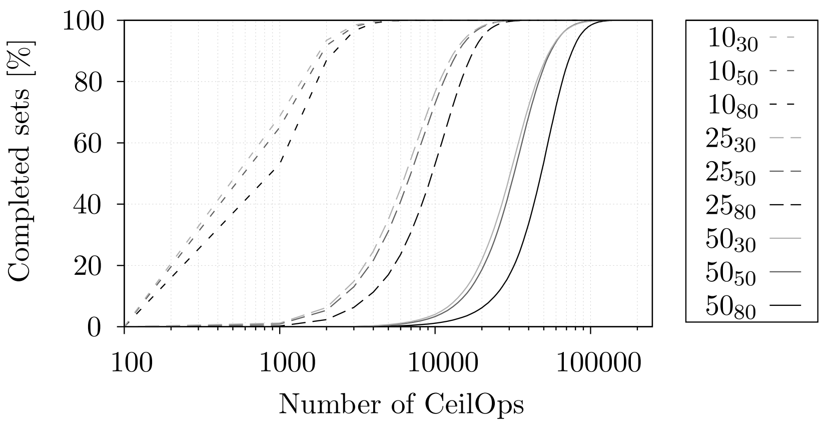

Figure 69: Impact of ITU on the SCD algorithm termination rate, starting at 30%, 50% and 80%

the system. Thus, the ITU has an impact on the SCD algorithm's efficiency only if the execution time of the algorithm is very limited. In the long term (i.e. approximately after 2,000 CeilOps) the ITU becomes irrelevant and the utilisation increase is dominated by the number of servers in the system.

The second interesting property of the SCD algorithm is its termination within a certain number of CeilOps. In Figure 69 it can be observed that the algorithm's termination is mainly influenced by the number of servers and the system's ITU has only a negligible effect on the algorithm's performance. As Figure 68 already indicated, the processor utilisation for a given number of servers in the system converges towards a similar level after a certain number of CeilOps. Hence, after this point the progress of the SCD algorithm, including the termination, becomes similar for different ITUs as well.

6.5.3 Experiment 13: Impact of the server parameter type

The server's temporal attributes can be either of continuous or discrete type. Hence, there can be three different server sets in the system. Only continuous, only discrete or a mixed set of servers. The next experiment analyses the consequences of the different server sets (see Figures 70–71).

This experiment shows that the graphs of different temporal attribute types are almost overlapping. Consequently the server's temporal attribute type has

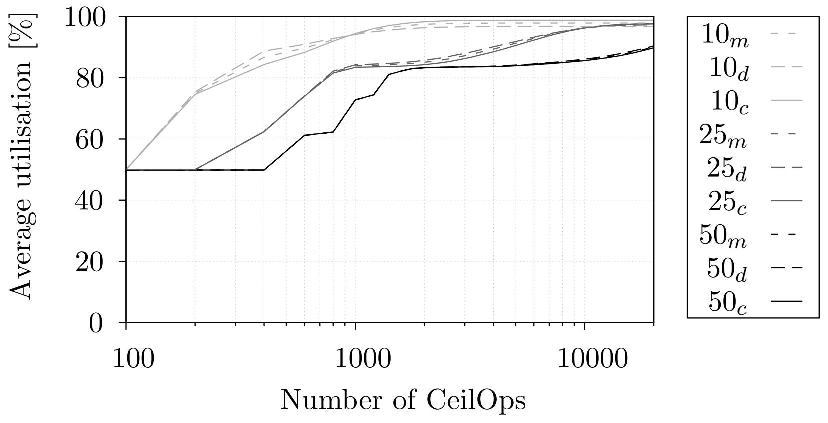

Figure 70: Measurements of the average processor utilisation for continuous, discrete and mixed server types

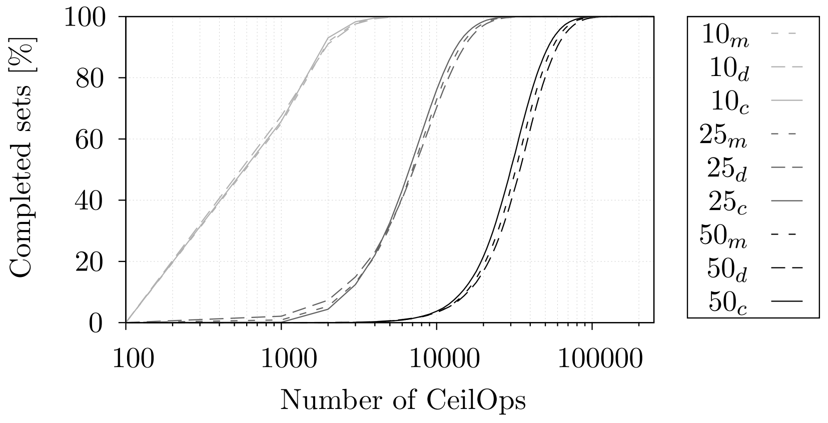

Figure 71: Measurements of the SCD algorithm termination rate for continuous, discrete and mixed server types

only a minor effect on the increase of the processor's utilisation (Figure 70) and the runtime of the SCD algorithm (Figure 71). The more servers are in the system the less relevant is the type of their temporal attributes, since for the generated test-cases the different utilisation values of the discrete servers are becoming finer. The conclusion is that a finer specification of the discrete server utilisation values facilitates a behaviour of the SCD algorithm that is similar to the continuous case.

6.5.4 Experiment 14: Impact of server arrangements in importance levels

In this experiment the effect of the importance level, an SCD algorithm specific attribute, is evaluated. We examine the difference between the scenario when all servers are assigned to the same importance level, and when they are randomly distributed over the maximum of five importance levels (Figures 72–73).

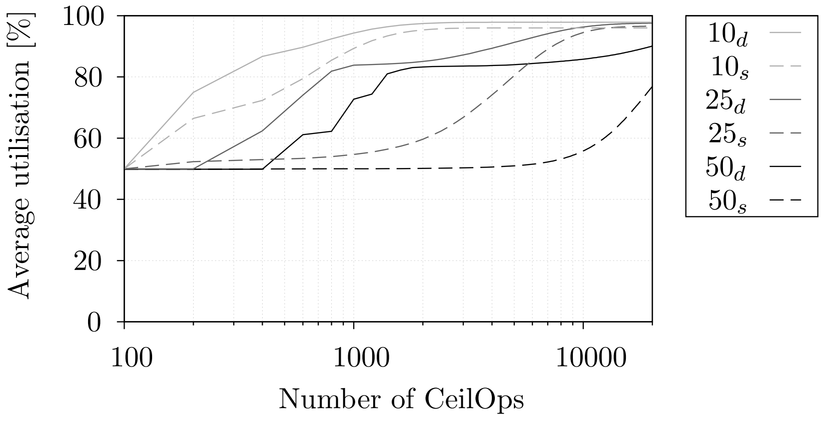

Figure 72: Average processor utilisation based on the importance level allocation

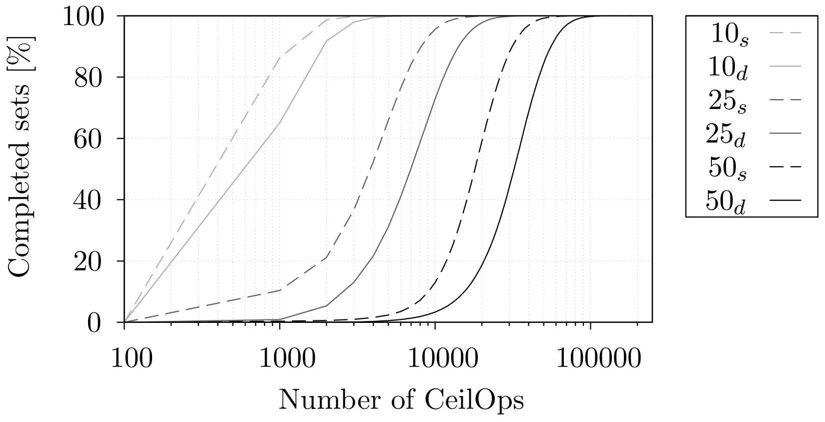

Figure 73: SCD algorithm termination rate based on the importance level allocation

Figure 72 shows, that the use of a single importance level has, similar to the effect of the ITU, a negative impact on the speed at which processor spare capacity can be distribute for small CeilOps. The processor's utilisation increase is much slower at an early stage of the SCD algorithm, though in long term, the processor's utilisation values for both scenarios converge towards a similar level. We note, however, that there is a slight utilisation loss in the case of a single importance level.

On the other hand, there is a slight decrease in the number of CeilOps required by the algorithm to regularly terminate if a single importance level is used (see Figure 73). This is the result of omitting the schedulability test for the empty importance levels.

6.5.5 Experiment 15: Performance of different SCD algorithm versions

This experiment compares the efficiency the two different implementation proposals, i.e. the coarse- and the fine-grained SCD algorithm (see Figure 74–76).

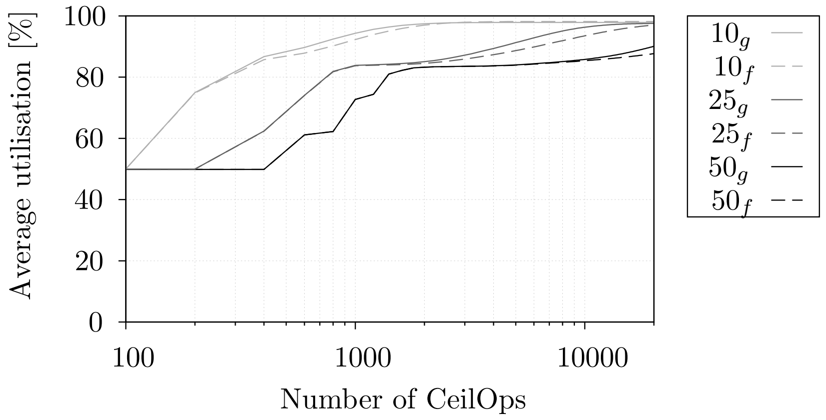

Figure 74: Average processor utilisation comparison between the coarse-grained and fine-grained SCD algorithm

Considering the speed at which the processor's capacity is distributed among the servers in the system, shows that there is hardly any difference between the coarse- and the fine-grained versions of the SCD algorithm (see Figure 74). In fact, the coarse-grained version increases the processor utilisation slightly faster since minor utilisation increases are left out of consideration in contrast the fine-grained version.

The smaller number of CeilOps required by the coarse-grained algorithm (Figure 75) is apparent since the additional schedulability tests, involved in the utilisation increase of individual servers, are omitted.

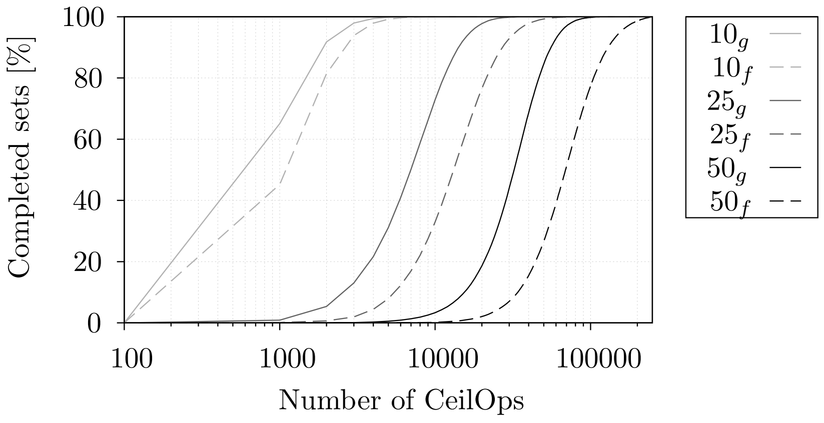

Figure 75: SCD algorithm termination rate comparison between the coarse-grained and fine-grained SCD algorithm

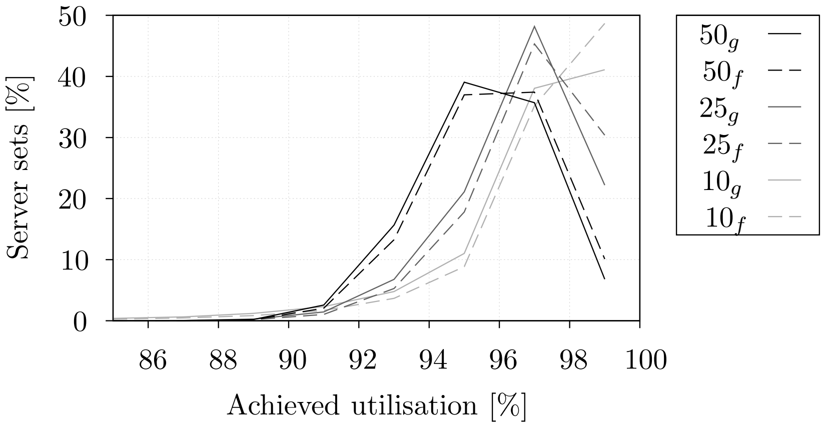

Figure 76: Achieved resource utilisation histogram for the coarse-grained and fine-grained SCD algorithm

The resource utilisation achieved by the two algorithms after the termination is very similar for systems with more then 25 servers (see Figure 76). However, the fine-grained SCD algorithm can achieve a higher utilisation for more server sets if the set contains less than 25 servers. This is due to the higher effort invested at each importance level to distribute the processor's capacity in finer chunks to the servers in the system. This property of the algorithm becomes more significant as the number of servers in the system decreases. Nevertheless, these noticeable differences occur in the range when the achieved processor utilisation is already high, that is to say, approximately between 95% and 100%.

Usually the advantage of the marginally higher utilised and distributed spare processor capacity is not significant enough to justify the application of the more complex fine-grained SCD algorithm.

6.5.6 Experiment 16: SCD algorithm execution time upper bound

The main purpose of this experiment is to determine which parameters influence the SCD algorithm's execution time and to provide information facilitating the calculation of the required resource reservation within the system software for the SCD algorithm. Additionally, using a pragmatic approach, a linear upper bound equation is derived for the execution time of the SCD algorithm.

During the design phase of real-time systems, information is required about the expected complexity of the SCD algorithm. In the following, results obtained by previous experiments are extended by mapping the number of CeilOps onto absolute execution times. As a result, a generic approach to derive an upper bound equation will be defined, allowing engineers during the design phase to determine the required amount of execution capacity for the SCD algorithm. Furthermore, a specific upper bound equation is presented for the used hardware platform, demonstrating the suggested approach for the derivation of SCD algorithm's execution time upper bound equation.

In order to exclude operating system and other undesirable overhead during the measurements, an embedded system was configured to execute only the SCD algorithm. This provided the facility to obtain absolute values for the execution time of the SCD algorithm. The embedded hardware platform used for this experiment consisted of an MPC555 microcontroller running at 40 MHz system clock.

To create the target executable file, the same setup and configuration was used as described in Section 5.3. Accordingly, the number of test cases was also reduced to 3,000 randomly generated server sets for each fixed number of servers set size (i.e. 5, 10, 15,. . . , 45, 50 servers). The ITU (i.e. 30%, 50% or 80%) and the type of the server sets (only continuous, only discrete or mixed type) were randomly created for each of the 3,000 server sets. Although the number of examined server sets was reduced in contrast to the measurements performed on the desktop computer, sufficient data was captured to analyse the behaviour of the SCD algorithm on real hardware.

Since it has been concluded that the coarse-grained algorithm has a better performance than the fine-grained version, the measurements on the embedded hardware platform and the corresponding evaluation are limited to only the coarse-grained SCD algorithm.

As an initial step, the dependency of the SCD algorithm's execution time on the number of CeilOps and servers in the system is examined. Figure 77 shows for different numbers of servers the execution time plotted against the number of CeilOps and the corresponding regression lines. Since the scatter of the measured values along the x-axis is very narrow for the test-cases with 5, 10 and 15 servers, their regression lines are omitted.

To determine the necessary values for the linear upper bound equation, the least-squares linear regression method was applied to the collected data. Table 10 summarises the parameters of the regression lines from Figure 77. The data in Table 10, as well as visual inspection of Figure 77 indicate that the slope of each regression line is very similar. This observation suggests a linear dependency of execution time on the number of CeilOps.

Table 10: Regression line parameters

Server#	Function	Slope (10^{-3})	Y-axis offset
50	$f_{50}(x)$	1.5425	99.287
45	$f_{45}(x)$	1.5520	87.238
40	$f_{40}(x)$	1.5579	75.519
35	$f_{35}(x)$	1.5722	64.308
30	$f_{30}(x)$	1.5862	53.022
25	$f_{25}(x)$	1.6263	42.277
20	$f_{20}(x)$	1.6386	32.486

But a single linear equation is not sufficient to express the execution time of the SCD algorithm. Since the regression lines do not overlap but have an offset between them, the dependency of this offset on the number of servers has been examined as well. In Figure 77 the intersection points of the regression lines with z-y-plane shows that the offset also increases linearly with the number of servers in the system.

The equation describing the y-axis intersection of the execution time regression lines as a function of the number of servers, was also determined via linear regression (see Equation 6.5).

$$y_0(v) = 2.381 \cdot v - 21.397 \tag{6.5}$$

The former analysis reveals that the execution time depends mainly on two parameters. Figure 77 shows the execution times plotted along the y-axis. The x-axis represents the number of CeilOps and the z-axis the number of servers. The observable linear behaviour of the plotted data along the x-axis as well as along the z-axis suggests the definition of the execution time upper bound as a plane equation with the number of CeilOps and servers as independent variables.

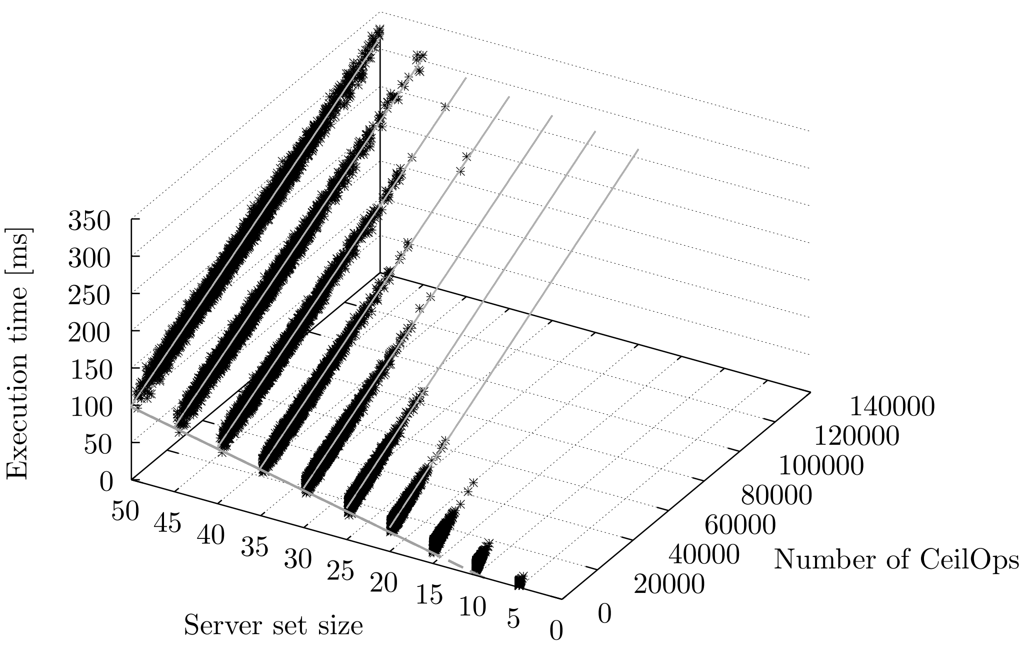

Figure 77: Execution time samples

The general form of the plane equation is defined as $C(n, v) = a_1 \cdot n + a_2 \cdot v$, with $C(n, v)$ representing the execution time, n the number of CeilOps and v the number of servers.

Next, the coefficients, a_1 and a_2, for the plane equation have to be specified. They are derived from the data that was obtained by measurements on the real embedded hardware platform.

The first coefficient, a_1, is determined by calculating the average slope of the execution time regression lines. The average value is approximately $1.58224 \cdot 10^{-3}$, but for ease of use, this value is rounded up to $1.6 \cdot 10^{-3}$.

Finally, the value of the second coefficient, a_2, is computed. Based on Equation 6.5 and on the data of Figure 77, the value of 2.8 is determined for a_2. This value is obtained by the application of pragmatic steps in order to simplify Equation 6.5. The aim is to preserve just a single factor that expresses the dependency between the number of servers, and the y-axis intersection of the lines that represent the execution time upper bounds for each set of servers. In order to achieve this, the value of coefficient a_2 is increased from the value starting at 2.381 (as specified by Equation 6.5) until the regression lines in Figure 77 become the upper bound on the measured execution times for the corresponding set of servers (i.e. every execution time sample is below the upper bound).

Using this information, an execution time upper bound equation for the SCD algorithm can be derived (see Equation 6.6). The equation represents a pragmatically derived execution time upper bound for the MPC555 microcontroller that was used to carry out the performance measurements.

$$C(n, v) = 0.0016 \cdot n + 2.8 \cdot v \tag{6.6}$$

To demonstrate how the execution time upper bound increases with system complexity, the upper bound is plotted as a function of the number of servers in the system and the number of CeilOps. The data generated by the application of Equation 6.6 spans a plane in three dimensional space. The plane in Figure 78 illustrates the execution time upper bound of the SCD algorithm. For the sake of clarity contour lines are rendered at 50 ms intervals.

In order to determine the required resource reservation for the SCD algorithm, the expected maximal number of servers in the system and the granted maximal number of CeilOps for the runtime of the algorithm has to be specified. The parameter specifying the granted maximal number of CeilOps influences the probability of the SCD algorithm to terminate regularly for the assigned processing resource reservation. The probability of termination within a certain number of CeilOps has already been investigated during the empirical evaluation in the previous experiments of this section and is summarised in Table 11.

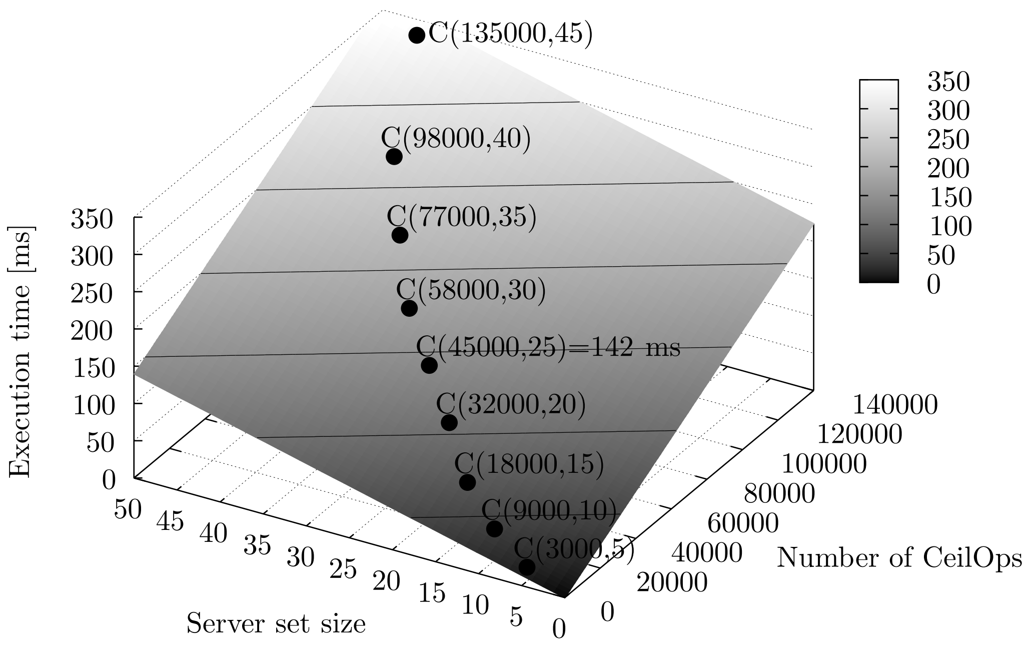

Figure 78: Execution time upper bound

Table 11 shows the maximal number of CeilOps that were required by the SCD algorithm to terminate for 99.99%, 99.90%, 99.00% and 90.00% of the test-cases. There is a slight difference in the number of required CeilOps for the algorithm among the test-cases that were performed with 30%, 50% and 80% ITU. For further evaluation the largest observed values were chosen.

Table 11: Number of CeilOps for the SCD algorithm

Server#	99.99%	99.90%	99.00%	90.00%
5	3000	2000	1000	1000
10	9000	6000	4000	3000
15	18000	14000	10000	6000
20	32000	24000	18000	12000
25	45000	36000	27000	18000
30	58000	49000	38000	26000
35	77000	66000	51000	36000
40	98000	86000	68000	49000
45	135000	108000	85000	62000
50	157000	132000	107000	77000

Based on the data that was collected during the empirical evaluation and the performance measurements, the required execution capacity for the SCD algorithm can be derived for a real-time system using an MPC555 microcontroller and a specified maximum number of servers.

As an example, we now determine for two different configurations the required execution capacity of an aperiodic server implementing the SCD algorithm. First, the execution capacity for the SCD algorithm's resource reservation on a system with a maximum of 5 servers is calculated. The execution capacity is chosen such that the algorithm can terminate in 99.99% of the cases. Applying the information from Table 11 in Equation 6.6, provides an execution capacity estimate of $C(3000, 5) = 0.0016 \cdot 3000 + 2.8 \cdot 5 = 18.8ms$. For the second example, we assume a system with at most 25 servers. Again, the execution capacity should allow the SCD algorithm to terminate in 99.99% of the cases. Hence, the estimated execution capacity is $C(45000, 25) = 0.0016 \cdot 45000 + 2.8 \cdot 25 = 142ms$. The calculated values for the examples are also illustrated in Figure 78.

The two coefficients, a_1 and a_2, of the linear equation depend on various hardware factors, like the availability and size of data caches, data bus bandwidth, external memory speed, etc. They also depend on the location (i.e. internal or external memory) of the relevant processing data. Therefore, the linear equation representing an upper bound for SCD algorithm's execution time, has to be individually derived for each hardware platform. This can however easily be performed using a suitable program for calibration, such as the one used to generate the evaluation results presented here.

6.5.7 Evaluation summary

The efficiency of the SCD algorithm has been examined in two different contexts. First, the algorithm underwent an empirical evaluation, followed by performance measurements on an embedded platform. The behaviour of the SCD algorithm was investigated in response to varying input parameters and the empirical evaluation reveals that the performance of the SCD algorithm is mainly influenced by the number of servers in the system. In addition to the number of servers, the choice of the algorithm has the biggest effect on the runtime of the algorithm. Regarding the two slightly different versions of the algorithm, in average case the coarse-grained SCD algorithm outperforms the fine-grained version, by achieving similar resource utilisation within a shorter time.

The performance evaluation of the SCD algorithm shows that, for example, in a system consisting of up to 25 mixed servers, the algorithm terminates in 99.99% of the cases within 45,000 CeilOps, and within the same number of CeilOps an average processor utilisation of 98% is reached. Based on the upper bound equation (Equation 6.6), the worst-case execution time for 45,000 CeilOps and 25 servers is equivalent to 142 ms on an MPC555 microcontroller with a system clock of 40 MHz. With focus on systems that undergo changes in long-term, for example, on average every minute, the 142 ms for the spare capacity distribution would imply that 0.24% of the entire processor capacity is consumed by this system task.

On faster processors, the cost for one ceiling operation of the SCD algorithm, as well as the overall execution time, decreases. Therefore the complexity of a system, measured in terms of the number of active servers, for which the SCD algorithm can be considered applicable, increases. For example, a processor with approximately 10 times the performance of a 40 MHz MPC555 might be used in avionics or telecommunications applications that need to support 10 to 25 applications or servers, respectively. In this case, such a processor would require at most 15 ms to execute the SCD algorithm.

6.6 Summary

In order to utilise the adaptability of flexible real-time applications and their corresponding servers, an easily and efficiently implementable online algorithm for the distribution of the processor's spare capacity was presented. The contribution can be summarised as:

- definition of an anytime spare capacity distribution algorithm that can generate useful results even if the algorithm's execution time is limited,
- enabling simultaneous runtime adaptation of continuous and discrete server temporal parameters (i.e. execution capacity and replenishment period),

- preventing schedulability test based inefficiency due to application of an efficient exact boolean schedulability test.

The definition of the iterative algorithm was complemented by measurements on a desktop computer as well as on a small embedded hardware platform to verify the efficiency and applicability of the SCD algorithm in real systems.

Although the distribution of the processor's spare capacity is not a trivial task, the performance measurements and evaluation showed that it can be feasibly integrated into real systems. The provided information allows the system designer to determine the processing resource needs and tailor the resource reservation for the SCD algorithm, and to trade off the algorithm's performance against the reserved processor capacity.

Chapter 7

Conclusion and future work

The main focus of the research undertaken was to address the problem of server parameter calculation and selection in open fixed-priority scheduled real-time systems. In the following sections the contribution of the thesis is summarised and an outlook is given on future work that could extend the capabilities of the proposed approach and facilitate the application of the server parameter selection in new domains of real-time systems.

7.1 Summary

The central proposition of the thesis (see Section 1.4) claimed that for flexible systems an efficient temporal partitioning is achievable by selecting optimal server parameters, hence enabling efficient processor utilisation. A summary of the chapters will relate the thesis' contribution to the main objective stated in the central proposition.

Following the introduction and the examination of work related to open real-time systems, flexible applications and the server parameter selection was investigated in Chapter 1 and 2. In Chapter 3, initial considerations were provided for the selection of optimal server parameters resulting in the lowest possible processor utilisation for a fixed-priority scheduled taskset. Essential analysis and tools that allow the possible interval of the optimal server period to be limited, have been introduced. The presented analysis enables the calculation of an upper and a lower bound on the optimal server period.

The optimal server period interval limited by the lower and upper bounds is used in Chapter 4 as a starting point to determine the optimal server parameters. Based on the server period upper bound, as one optimal parameter candidate, an efficient iterative algorithm is specified. The iterative algorithm is able to determine the optimal server parameters for a taskset with fixed temporal parameters. Tasksets containing flexible tasks are converted to additional pseudo tasksets where flexible tasks are assumed to have a fixed value for their temporal parameters. Each pseudo taskset can be related to an application mode of operation with corresponding fixed task parameter values. Hence, the iterative algorithm is always faced with a set of tasks that have fixed parameters and can determine the optimal server parameters without modification of the algorithm.

The evaluation showed that the iterative algorithm provides a good trade off between the algorithm's performance and the achieved server parameter optimisation. In contrast to related approaches, our iterative algorithm might be slightly slower under certain circumstance but up to approximately 4% less processor utilisation is reserved for certain tasksets by using the optimal server parameters.

The calculation of optimal server parameters is just the first step towards efficient processor utilisation in an open real-time system. During runtime, these systems are faced with a changing set of applications and therefore require an efficient test that can be utilised not only as an online acceptance tests but also as part of an algorithm to distribute the processor's spare capacity. The different possibilities for improving the performance of existing exact boolean schedulability tests were introduced and evaluated in Chapter 5. Based on the evaluation results the best performance for the online implementation of a schedulability test is achieved by the response time based test using the new initial values.

The best benefit of open real-time systems is achieved in conjunction with flexible applications and the resulting continuous and discrete servers, providing the appropriate resource reservation for the applications. Consequently, the processing time reservation for flexible applications can and must be adapted during runtime to efficiently utilise the processor capacity. The spare capacity distribution (SCD) algorithm introduced in Chapter 6 aims at maximising the processor utilisation by assigning as much as possible of the processors capacity to the active applications in the system. Hence, the major application of the efficient schedulability test is to determine the schedulability of different spare capacity distribution scenarios within the SCD algorithm.

In order to examine the efficiency and effectiveness of the offline optimal server parameter selection, as well as the online spare capacity distribution algorithm, both were characterised by performance measurements on the desktop computer and real embedded hardware platform.

Although the offline algorithm for optimal server parameter selection has a pseudo-polynomial complexity, the evaluation results with up to 35 tasks showed that the presented method is feasible for real-time applications with a realistic number of tasks and in most test-cases it had a better performance than simpler approximation methods.

The efficiency of the online schedulability test and the SCD algorithm were also evaluated based on empirical experiments. To obtain an insight into the performance behaviour of the introduced online algorithms, they were executed on a MPC555 microcontroller based embedded hardware platform. Based on the measurements that were captured on the embedded hardware platform, a method was presented to determine the required processing time reservation for the SCD algorithm in a real system.

The performance results obtained on the desktop computer and the embedded hardware, confirmed the effectiveness and efficiency of the improved schedulability test and the SCD algorithm. The empirical experiments examined the performance of the algorithms even beyond values that seem to be realistic for the envisaged category of embedded real-time systems. Conducting experiments with up to 50 servers demonstrated that the introduced optimisation steps, by utilising new initial values, and the specified online algorithms, are practicable for the anticipated systems.

For fixed-priority scheduled real-time systems, a 100% utilisation of the processor capacity is difficult to achieve. Especially, in hierarchically scheduled systems, it is inevitable to lose a certain amount of utilisable processor capacity due to the capacity consumption of the context switch overhead between applications. Nevertheless, the experiments carried out with the SCD algorithm showed that the presented simple algorithm is capable of distributing nearly 100% processor utilisation — including the utilisation consumed by the context switch overhead — among the applications in the system.

The specification of the FRESCOR middleware, on which the presented research of this thesis is based on, expresses general requirements to provide support for shared resources, but also resource management support for other than uniprocessor systems. Especially, the partitioning problem in multiprocessor system was covered [45] only at a very basic level and mainly from the implementations point of view. The rudimentary investigation of these topics during the FRESCOR project offers the opportunity to link the research of this thesis with new challenges as part of the future work.

7.2 Future work

The methods presented in the previous chapters mainly focus on uniprocessor systems without shared resources. The most evident extension to the offline and online server parameter selection methods introduced in this thesis would take into consideration local as well as global shared resources in a fixed-priority scheduled open real-time system.

One option for the offline server parameter selection that analyses the temporal requirements of a single taskset (presented in Chapter 3 and 4) could take into account the access to shared resources by incorporating it as blocking time on tasks. The method, as suggested by Almeida and Pedreiras [6], considers blocking time B_i for a task τ_i by increasing the submitted load for priority level i by B_i. Given this extended load function, the server parameter selection algorithm could determine the optimal values in the presence of shared resources.

In Section 2.2 we distinguished for open real-time systems between local and global shared resources. Local shared resources do not need any dedicated consideration since the critical code section accessing these resources can be interrupted when the server's execution capacity is exhausted. However, the access to global shared resources cannot be interrupted at any arbitrary time. Davis and Burns [27] introduced the *Hierarchical Stack Resource Policy (HSRP)* that allows a server to consume more capacity than was initially assigned to it. Their analysis in conjunction with the HSRP can be utilised to extend the methods for online acceptance test and spare capacity distribution presented in Chapter 5 and 6. However, the interface between the applications and the resource managing middleware would also have to be extended in order to make the information about the longest critical section (non interruptible code section accessing a shared resource) within an application available to the online acceptance test and spare capacity distribution module.

Another promising extension to the presented server parameter selection methods is the consideration of other than fixed-priority, like dynamic priority, scheduled tasksets. The challenge with dynamic priority scheduled tasksets is to find an efficient method that enables the determination of representative demand points that can be used by the offline server parameter selection algorithm to calculate the optimal server parameter values. As the *Earliest Deadline First (EDF)* scheduling is known to be optimal, it is expected to achieve a further reduction of unused processor capacity that is reserved for an application.

Temporal partitioning of communication media is also an important aspect especially in distributed real-time automotive and aerospace systems. For the *Controller Area Network (CAN)* a dedicated schedulability analysis has been devised by Davis et al. [31] to determine if timing guarantees for CAN messages can be met or not. Follow-on research may utilise the existing analysis dedicated for communication networks and provide a method for the calculation of communication server parameters, considering their non-preemptible property.

Another very interesting future topic is the temporal partitioning of multiprocessor or multi-core systems. The *survey of hard real-time scheduling on multiprocessor systems* [30] highlighted the complexity of the scheduling problem in the context of multiprocessor systems. Especially certain system model constraints have to be accepted in order to achieve optimality without clairvoyance. Furthermore, anomalies [30] like the increase of task periods or decrease of the assumed worst-case execution time resulting in unschedulable tasksets may occur in the case of multiprocessor systems, preventing the straightforward reuse of uniprocessor algorithms. The application of temporal partitioning concepts to multiprocessor systems would, however, counteract the identified anomalies and it would partially enable the reuse of uniprocessor algorithms. Nevertheless, the domain of multiprocessor systems still contains difficult challenges, like an efficient schedulability analysis or an efficient partitioning algorithm.

In this thesis, optimal server parameter selection focuses on minimizing the processor reservation for a given taskset. Yet, in the context of multiprocessor systems, the term *optimal* would have to be defined in order to enable appropriate adjustments to the presented algorithm and to achieve the envisaged quality of of server parameters.

With respect to the available spare capacity, the SCD algorithm could still be used on a per processor bases. The utilisation of each processor, with an assigned set of flexible servers, can be maximised by the defined SCD algorithm including minor modifications.

In summary, multiprocessor systems will have a major impact on the offline optimal server parameter selection and efficient schedulability tests. As part of the future work, these topics require extensive research and analysis, taking into consideration the constraints prevalent in multiprocessor systems.

7.3 Final word

This thesis addressed the problem of temporal partitioning of fixed-priority scheduled uniprocessor systems. The problem was approached at two levels.

First, an offline method was introduced that can be used during the application design in order to determine the minimal processor reservation for a given set of flexible tasks. The minimal processor reservation is expressed by the execution-time server parameters and the predetermined temporal guarantees are ensured during runtime by the server mechanism. Second, in order to reduce unallocated spare processor capacity, the spare capacity is exploited during runtime by the SCD algorithm and distributed among the applications in the system.

The work presented in this thesis is intended to be applicable in real systems and to contribute to the adoption of more efficient flexible real-time systems in a range of application domains.

References

[1] L. Abeni and G. C. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 4–13, Madrid, Spain, Dec 1998.

[2] Aeronautical Radio, Inc., Annapolis, Maryland, USA. *ARINC Specification 653P1-2: 653P1-2 Avionics Application Software Standard Interface, Part 1 - Required Services*, March 2006.

[3] A. Aguilar-Soto and G. Bernat. Bicriteria fixed-priority scheduling in hard real-time systems: Deadline and importance. In *Real-Time and Network Systems*, number 25–34, 2006.

[4] M. Aldea Rivas, G. Bernat, I. Broster, A. Burns, R. Dobrin, J. M. Drake, G. Fohler, P. Gai, M. González Harbour, G. Guidi, T. L. J. Javier Gutiérrez Garcia, G. Lipari, J. L. M. P. José M. Martínez, J. C. Palencia Gutiérrez, and M. Trimarchi. FSF: A real-time scheduling architecture framework. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 113–124, 2006.

[5] L. Almeida, S. Fischmeister, M. Anand, and I. Lee. A dynamic scheduling approach to designing flexible safety-critical systems. In *Proceedings of the 7th ACM & IEEE international conference on embedded software*, pages 67–74, 2007.

[6] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: Response-time analysis and server design. In *Fourth ACM International Conference On Embedded Software*, pages 95–103, 2004.

[7] N. C. Audsley. *Flexible Scheduling of Hard Real-Time Systems*. PhD thesis, University of York, 1993.

[8] N. C. Audsley, A. Burns, M. F. Richardson, K. W. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[9] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Incorporating unbounded algorithms into predictable real-time systems. *Computer Systems Science and Engineering*, 8(2):80–89, Apr 1993.

[10] AUTOSAR GbR. *AUTOSAR: Specification of Operating Systems*, February 2009.

[11] T. P. Baker. Stack-based scheduling for realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

[12] E. Bini and S. K. Baruah. Efficient computation of response time bounds under fixed-priority scheduling. In *International Conference on Real-Time and Network Systems*, pages 95–104, 2007.

[13] E. Bini and G. C. Buttazzo. The space of rate monotonic schedulability. In *IEEE Real-Time Systems Symposium*, pages 169–178, 2002.

[14] E. Bini and G. C. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers*, 53(11):1462–1473, 2004.

[15] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[16] E. Bini, G. C. Buttazzo, and G. M. Buttazzo. Rate monotonic analysis: The hyperbolic bound. *IEEE Transactions on Computers*, 52(7):933–942, 2003.

[17] R. J. Bril, W. F. J. Verhaegh, and E.-J. D. Pol. Initial values for online response time calculations. In *Euromicro Conference on Real-Time Systems*, pages 13–22, 2003.

[18] A. Burns and A. Wellings. *Immediate ceiling priority protocol*, chapter 13.11.1, pages 491–493. Addison-Wesley Longman Publishing Co., Inc., 3rd edition, 2001.

[19] G. C. Buttazzo. Research trends in real-time computing for embedded systems. *ACM Special Interest Group on Embedded Systems Review*, 3(3):1–10, 2006.

[20] G. C. Buttazzo and E. Bini. Optimal dimensioning of a constant bandwidth server. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 169–177, 2006.

[21] G. C. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 286–295, Washington, DC, USA, 1998. IEEE Computer Society.

[22] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3):289–302, 2002.

[23] D. Chen, A. K. Mok, and T.-W. Kuo. Utilization bound revisited. *IEEE Transaction on Computers*, 53(3):351–361, 2003.

[24] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in Real Time Systems*. John Wiley and Sons Ltd, 2002.

[25] A. Crespo, M. T. de Esteban, I. Ripoll, V. B. Tortosa, and M. Frödin. Execution platforms implementation and evaluation of the processor contract model ii. Technical Report D–EP.3v2-1, Universitat Politècnica de València, 2008.

[26] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 389–398, 2005.

[27] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 257–267, 2006.

[28] R. I. Davis and A. Burns. An investigation into server parameter selection for hierarchical fixed priority pre-emptive systems. In *16th International Conference on Real-Time and Network Systems*, pages 19–28, 2008.

[29] R. I. Davis and A. Burns. Response time upper bounds for fixed priority real-time systems. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, 2008.

[30] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *To appear in ACM Computing Surveys*, 2011.

[31] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien. Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.

[32] R. I. Davis, A. Zabos, and A. Burns. Efficient exact schedulability tests for fixed priority real-time systems. *IEEE Transactions on Computers*, 57(9):1261–1276, 2008.

[33] R. I. Davis, A. Zabos, A. Burns, R. Guerra, and G. Fohler. Optimal contract calculation techniques. Technical Report D–AT.4, University of York, 2008.

[34] D. de Niz, L. Abeni, S. Saewong, and R. R. Rajkumar. Resource sharing in reservation-based systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 171–180, 2001.

[35] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *IEEE Real-Time Systems Symposium*, pages 308–319, 1997.

[36] Z. Deng, J. W.-S. Liu, and J. Sun. A scheme for scheduling hard real-time applications in open system environment. In *Proceedings of the 9th Euromicro Workshop on Real-Time Systems*, pages 191–199, Toledo, Spain, Jun 1997.

[37] Z. Deng, J. W.-S. Liu, L. Zhang, S. Mouna, and A. Frei. An open environment for real-time applications. *Real-Time Systems*, 16(2–3):155–185, 1999.

[38] A. Easwaran, M. Anand, and I. Lee. Compositional analysis framework using EDP resource models. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 129–138, 2007.

[39] A. Easwaran, I. Shin, O. Sokolsky, and I. Lee. Incremental schedulability analysis of hierarchical real-time components. In *Proceedings of the 6th ACM & IEEE International conference on embedded software*, pages 272–281, 2006.

[40] X. A. Feng and A. K. Mok. A model of hierarchical real-time virtual resources. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pages 26–35, 2002.

[41] N. Fisher. An FPTAS for interface selection in the periodic resource model. In *17th International Conference on Real-Time and Network Systems*, pages 127–135, 2009.

[42] G. Fohler. Realizing changes of operational modes with pre run-time scheduled hard real-time systems. In *Proceedings of Second International Workshop on Responsive Computer Systems*, 1992.

[43] G. Fohler. Changing operational modes in the context of pre run-time scheduling. In *IEICE TRANSACTIONS on Information and Systems*, volume E76-D, pages 1333–1340, 1993.

[44] G. Fohler. Requirements analysis version 2. Technical Report D–RA.2, Technische Universität Kaiserslautern, 2008.

[45] P. Gai. Multiprocessor execution platforms. Technical Report D–EP.7v2, Evidence S.r.l., 2009.

[46] T. M. Ghazalie and T. P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 9:31–67, 1995.

[47] M. González Harbour and M. T. de Esteban. Architecture and contract model for integrated resources. Technical Report D–AC.2v1, Universidad de Cantabria, 2007.

[48] M. González Harbour and M. T. de Esteban. Architecture and contract model for integrated resources ii. Technical Report D–AC.2v2, Universidad de Cantabria, 2008.

[49] M. González Harbour, D. S. López, and M. T. de Esteban. Mode change protocol for budget changes in contract-based scheduling. Technical report, Universidad de Cantabria, 2008.

[50] S. Hua and G. Qu. A new quality of service metric for hard/soft real-time applications. In *ITCC*, pages 347–351, 2003.

[51] M. Joseph and P. K. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[52] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, 1988.

[53] T.-W. Kuo and C.-H. Li. A fixed-priority-driven open environment for real-time applications. In *IEEE Real-Time Systems Symposium*, pages 256–267, 1999.

[54] C.-G. Lee, L. Sha, and A. Peddi. Enhanced utilization bounds for QoS management. *IEEE Transactions on Computers*, 53(2):187–200, 2004.

[55] T.-M. Lee, W. Wolf, and J. Henkel. Dynamic runtime re-scheduling allowing multiple implementations of a task for platform-based designs. In *DATE*, pages 296–301, 2002.

[56] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 201–209, 1990.

[57] J. P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks fixed-priority preemptive systems. In *IEEE Real-Time Systems Symposium*, pages 110–123, 1992.

[58] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium*, pages 166–171, 1989.

[59] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the 8th IEEE Real-Time Systems Symposium*, pages 261–270, 1987.

[60] B. Leiner, M. Schlager, R. Obermaisser, and B. Huber. A comparison of partitioning operating systems for integrated systems. In *Proceedings of the 26th International Conference on Computer Safety, Reliability and Security*, pages 342–355, 2007.

[61] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4):237–250, Dec 1982.

[62] C. Lin, T. Kaldewey, A. Povzner, and S. A. Brandt. Diverse soft real-time processing in an integrated system. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 369–378, 2006.

[63] C. Lin, P. Marti, S. A. Brandt, S. Banachowski, M. Velasco, and J. Fuertes. Improving control performance using discrete quality of service levels in a real-time system. In *Work-in-Progress Session of the IEEE Real-Time Technology and Applications Symposium*, 2004.

[64] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *Euromicro Conference on Real-Time Systems*, pages 151–158, 2003.

[65] G. Lipari and E. Bini. A methodology for designing hierarchical scheduling systems. *Journal of Embedded Computing*, 1(2):257–269, 2005.

[66] G. Lipari, E. Bini, and G. Fohler. A framework for composing real-time schedulers. *Electronic Notes in Theoretical Computer Science*, 82(6):133–146, 2003.

[67] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[68] J. Liu. *Real-Time Systems*. Prentice-Hall, Inc., 2000.

[69] W.-C. Lu, K.-J. Lin, H.-W. Wei, and W.-K. Shih. Period-dependent initial values for exact schedulability test of rate monotonic systems. In *IEEE Parallel and Distributed Processing Symposium*, pages 1–8, 2007.

[70] R. Marau, P. Leite, M. Velasco, P. Martí, L. Almeida, P. Pedreiras, and J. M. Fuertes. Performing flexible control on low-cost microcontrollers using a minimal real-time kernel. *IEEE Transactions on Industrial Informatics*, 4(2):125–133, May 2008.

[71] C. W. Mercer, R. Rajkumar, and H. Tokuda. Applying hard real-time technology to multimedia systems. In *Proceedings of the Workshop on the Role of Real-Time in Multimedia/Interactive Computing Systems*, 1993.

[72] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems. Technical Report CMU–CS–93–157, Carnegie Mellon University, 1993.

[73] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 90–99, 1994.

[74] A. K. Mok. *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

[75] A. K. Mok, X. A. Feng, and D. Chen. Resource partition for real-time systems. In *IEEE Real Time Technology and Applications Symposium*, pages 75–84, 2001.

[76] L. Nogueira and L. M. Pinho. Time-bounded distributed qos-aware service configuration in heterogeneous cooperative environments. *Journal of Parallel and Distributed Computing*, 69:491–507, 2009.

[77] R. Obermaisser, P. Peti, B. Huber, and C. E. Salloum. DECOS: An integrated time-triggered architecture. *Elektrotechnik und Informationstechnik*, 123(3):83–95, 2006.

[78] P. S. M. Pedro and A. Burns. Schedulability analysis for mode changes in flexible real-time systems. In *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, pages 172–179, 1998.

[79] A. Rahni, E. Grolleau, and M. Richard. An efficient response-time analysis for real-time transactions with fixed priority assignment. *Innovations in Systems and Software Engineering*, 5(3):197–209, 2009.

[80] R. R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, pages 150–164, 1998.

[81] R. R. Rajkumar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek. A resource allocation model for QoS management. In *IEEE Real-Time Systems Symposium*, pages 298–307, 1997.

[82] R. Ramaker, W. Krug, and W. Phebus. Application of a civil integrated modular architecture to military transport aircraft. In *Digital Avionics Systems Conference*, pages 2.A.4–1–2.A.4–10, 2007.

[83] J. Real and A. Crespo. Offsets for scheduling mode changes. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 3–10, 2001.

[84] J. Real and A. Crespo. Mode change protocols for real-time systems: a survey and a new proposal. *Real-Time Systems*, 26(2):161–197, 2004.

[85] J. Real and A. J. Wellings. The ceiling protocol in multi-moded real-time systems. In *Proceedings of the 1999 Ada-Europe International Conference on Reliable Software Technologies*, pages 275–286, 1999.

[86] D. Rosu, K. Schwan, S. Yalamanchili, and R. Jha. On adaptive resource allocation for complex real-time application. In *Proceedings of the 18th IEEE International Real-Time Systems Symposium*, pages 320–329, 1997.

[87] S. Saewong, R. R. Rajkumar, J. P. Lehoczky, and M. H. Klein. Analysis of hierarchical fixed-priority scheduling. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, pages 173–181, 2002.

[88] L. Sha, J. P. Lehoczky, and R. R. Rajkumar. Solutions for some practical problems in prioritizing preemptive scheduling. In *Proceedings of the 7th IEEE Real-Time Sytems Symposium*, pages 181–191, 1986.

[89] L. Sha, R. R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sep 1990.

[90] L. Sha, R. R. Rajkumar, J. P. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, 1(3):243–264, 1989.

[91] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, pages 2–13, 2003.

[92] I. Shin and I. Lee. Compositional real-time scheduling framework. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 57–67, 2004.

[93] I. Shin and I. Lee. Compositional real-time scheduling framework with periodic model. *ACM Transactions on Embedded Computing Systems*, 7(3):30:1–30:39, May 2008.

[94] M. Sjödin and H. Hansson. Improved response-time analysis calculations. In *IEEE Real-Time Systems Symposium*, pages 399–408, 1998.

[95] B. Sprunt. *Aperiodic task scheduling for real-time systems.* PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1990.

[96] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

[97] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, 1995.

[98] K. W. Tindell and A. Alonso. A very simple protocol for mode changes in priority preemptive systems. Technical report, Universidad Politecnica de Madrid, 1996.

[99] K. W. Tindell, A. Burns, and A. J. Wellings. Mode changes in priority pre-emptively scheduled systems. In *IEEE Real-Time Systems Symposium*, pages 100–109, 1992.

[100] E. Wandeler and L. Thiele. Optimal TDMA time slot and cycle length allocation for hard real-time systems. In *Proceedings of the 2006 Conference on Asia South Pacific Design Automation*, pages 479–484, 2006.

[101] C. B. Watkins and R. Walter. Transitioning from federated avionics architectures to integrated modular avionics. In *Digital Avionics Systems Conference*, pages 2.A.1–1–2.A.1–10, 2007.

[102] A. Zabos and A. Burns. Towards bandwidth optimal temporal partitioning. Technical Report YCS-2009-442, University of York, 2009.

[103] A. Zabos, R. I. Davis, A. Burns, and M. González Harbour. Spare capacity distribution using exact response-time analysis. In *17th International Conference on Real-Time and Network Systems*, pages 97–106, 2009.

[104] S. Zilberstein and S. J. Russell. Anytime sensing planning and action: A practical model for robot control. In *International Joint Conference on Artificial Intelligence*, pages 1402–1407, 1993.