A Hierarchical Scheduling Model for Dynamic Soft-Realtime Systems*

Vladimir Nikolov[1], Stefan Wesner[2], Eugen Frasch[3], and Franz J. Hauck[4]

1 **Institute of Information Resource Management, Ulm University, Ulm, Germany**
vladimir.nikolov@uni-ulm.de
2 **Instistute of Information Resource Management, Ulm University, Ulm Germany**
stefan.wesner@uni-ulm.de
3 **Institute of Distributed Systems, Ulm University, Ulm, Germany**
eugen.frasch@uni-ulm.de
4 **Institute of Distributed Systems, Ulm University, Ulm, Germany**
franz.hauck@uni-ulm.de

Abstract

We present a new hierarchical approximation and scheduling approach for applications and tasks with multiple modes on a single processor. Our model allows for a temporal and spatial distribution of the feasibility problem for a variable set of tasks with non-deterministic and fluctuating costs at runtime. In case of overloads an optimal degradation strategy selects one of several application modes or even temporarily deactivates applications. Hence, transient and permanent bottlenecks can be overcome with an optimal system quality, which is dynamically decided. This paper gives the first comprehensive and complete overview of all aspects of our research, including a novel CBS concept to confine entire applications, an evaluation of our system by using a video-on-demand application, an outline for adding further resource dimension, and aspects of our prototype implementation based on RTSJ.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems, D.4.1 Scheduling

Keywords and phrases real-time, scheduling, hierarchical, dynamic, ARTOS

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.7

1 Introduction

This work summarizes the scientific results of the ARTOS [5] project. Its objective was to develop and integrate adaptive resource-management mechanisms in a generic framework for soft real-time systems supporting a dynamic set of applications and tasks. As target platforms general purpose systems as traditional x86, ARM and PPC based platforms were supposed, which may range from small embedded and mobile devices, to desktop PCs and large scale servers. The result is an open and dynamic execution environment for soft real-time applications, which can be integrated on different system levels and used for a variety of scenarios.

* This research has been supported by Deutsche Forschungsgemeinschaft (DFG) under Grant No. HA2207/8-1 (ARTOS) and BMBF under Grant No. 01/H13003 (MyThOS).

29th Euromicro Conference on Real-Time Systems (ECRTS 2017).
Editor: Marko Bertogna; Article No. 7; pp. 7:1–7:23

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The demand for such adaptive platforms is currently on the rise. A good example comes from the automotive sector. The consolidation of functions with mixed criticality on single powerful ECUs (electronic control units) is a prevailing requirement for manufacturers of head-unit and entertainment systems [14, 35]. For example, the central console of various brands already provides diverging functions concurrently, e.g. for telematics, connectivity, navigation, display, etc. In the near future the functional scope shall be dynamically extendable "over-the-air" without maintenance effort in a garage. Even passengers will be able to download and activate new third-party apps on their backseat entertainment devices. Many of them will be executed concurrently on single embedded devices and ECUs. The required resource-management mechanisms could be provided by the OS, a middleware or the application itself. Certainly, a working solution would fit a variety of further use-cases in the fields of embedded systems, robotics and even complex applications in data centres, as has been shown in [30].

Since applications and activities may freely arrive or depart during runtime, the system has to adapt itself to a varying computational load, while sustaining a steady application performance and best possible overall quality. Moreover, the exact resource requirements of the applications are assumed to be *unknown* until runtime and also may *fluctuate* during their execution. Reasons for such cost variations can be found not only in the non-deterministic behavior of the executional hardware (e.g. effects of non-uniform memory access, caching, virtual memory etc.). They are also caused by data dependencies and varying algorithmic complexity of tasks, or even on a software compositional level where applications' logical control flow may be strongly distracted by dynamic service discovery, qualification and execution. The latter particularly applies for dynamically composed apps in service-oriented environments. Such a system requires not only mechanisms for cost monitoring and admission control, but also adaptive resource reservation and degradation techniques for transient and permanent overload management.

The main scientific contribution of ARTOS is its approximation and scheduling framework which is subject of this paper. In ARTOS we primarily focused on CPU time management for a single processor. Anyway, plenty of the provided mechanisms are applicable to other resource types as well, e.g. network, memory or energy usage. We are currently working on an extension of our model for distributed and parallel applications (i.e. a multi-processor version) and multiple resource types.

However, the problem of optimal and adaptive resource distribution for a varying set of unknown real-time applications is hard even for the uniprocessor case and a single resource (i.e. CPU time). An appropriate cost approximation model for applications and tasks must hold as a decision basis independently from the time resolution of their events. An execution schedule has to be dynamically generated and updated on task arrival and departure, but (and even more important) also in case of cost fluctuations. On the other side, costs for (online) feasibility analysis have to be omitted and its occurrence clearly defined and planned. Furthermore, cost reservations for tasks and applications are crucial for their prioritization and isolation, especially in the case of overload and faults. On the other side, for optimal system utilization and admission the reservations have to be adjustable as well. However, unpredictable bottlenecks due to potential system over-provisioning are still possible and an appropriate overload management strategy is necessary in order to keep the system and the applications in a consistent state. Its reactiveness determines the probability of temporal faults and thus the overall system quality of service.

According to Buttazzo et al. [10] there are three basic mechanisms to handle overload situations, i.e. modify the periods of tasks (*elastic scheduling* [11]), omit certain jobs (*job*

skipping [20]) and degrade tasks workload on the algorithmic level. We developed a new generalized application model that supports different *operational modes*, and thus quality levels, of applications. The actual modes are centrally decided and adjusted by the system during runtime following priorities and their measured resource demands. Therefore, an optimization approach was developed which allows for a trade-off between the best possible system quality and its optimal utilization. This mechanism implements a sophisticated overload management strategy in our system based on a controlled and directed application degradation. The effective realization of the modes is finally up to the application developers, and can be implemented with any or as a combination of the previously mentioned techniques. In [28] we already presented a realization of multi-mode tasks with multiple versions.

The required monitoring, reservation and automatic degradation mechanisms for the applications were integrated as parts of a *hierarchical scheduling model*. In contrast to our previous publications on this topic (e.g. [31, 29, 27]), we present here the first comprehensive and complete overview of our approach, including recent results and improvements as well as several novel contributions. These will be explicitly accentuated in the paper. Our scheduler was integrated into a component-based and service-oriented software framework, which offers a generic platform for dynamic and open soft real-time systems [4]. Our framework was implemented based on the Real-Time Specification for Java (RTSJ) on top of a Linux system, a modified OSGi framework and Aicas's JamaicaVM [19]. The JVM offered us basic support for (a)periodic activities, deadlines, cost monitoring, events (e.g. overrun, deadline miss) and a fixed-priority–based scheduling. Our close cooperation with Aicas helped us to extend RTSJ 1.0.2 and its implementation in JamaicaVM where necessary [28], in order to improve the runtime support for our scheduler. However, our evaluation finally proves the functionality and adaptational capabilities of our system to unpredictable and varying load [26].

This paper is structured as follows: Section 2 introduces our system model and Section 3 gives a basic overview of our scheduler's functionality. Section 4 illustrates our cost approximation model and scheduling on the task-level. In Section 5 capacity reservation mechanisms for the applications and their modes are explained. Section 6 describes the decision of the optimal modes selection. Section 7 discusses our system implementation and in Section 8 different evaluation scenarios and results are presented. In Section 9 related work is assessed and discussed. Finally, in Section 10 we draw conclusions and present future work directions.

2 System Model and Scheduling Problem

We assume a dynamic set of n concurrent applications A_1, $A_2, ..., A_n$, and for each A_i there is a set of modes $M_{i,1}, M_{i,2}, ..., M_{i,m_i} \in \mathcal{M}_i$, which can be switched dynamically. Each application further consists of a set of k tasks $\tau_{i,k} \in \mathcal{T}_i$ also having different modes, e.g. with different periods $T_{i,k}$, deadlines $D_{i,k}$ and estimated costs $C_{i,k}$ with respect to each application mode $M_{i,j}$. We primarily focused on periodic tasks[1], which better suit our actual audio/video processing use-cases. Thus, an application mode manifests as a preconfigured *set of* particular *task modes*. In general, a higher application mode has a higher computational demand $B_{i,j}$ but delivers a better quality, e.g. a better video resolution. The quality is formalized by application-specific utility functions $\mathrm{u}_i(M_{i,j})$ given by developers. Their values are precomputed and normalized in the system as a relation of quality benefits between the different application modes, i.e. $U_{i,j} = \mathrm{u}_i(M_{i,j})/u_i(M_{i,|\mathcal{M}_i|})$. Each application A_i contains a

[1] Aperiodic tasks require a special treatment in periodic systems, e.g. aperiodic server mechanisms like Deferrable Servers [37].

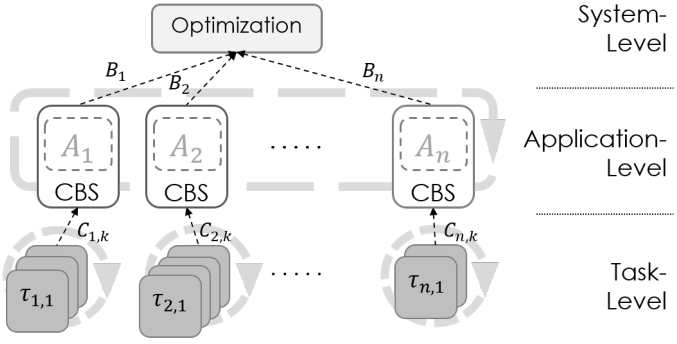


Figure 1 Scheduling Model Architecture.

mode $M_{i,0}$ with $B_{i,0} = \mathrm{u}_i(M_{i,0}) = 0$ which corresponds to the application being temporarily stopped[2]. Adding such a mode ensures the presence of an optimal resource distribution [31] and implements a fine-grained and dynamic admission control. Applications can be weighted by users or the system itself with importance factors a_i. These factors affect most obviously each applications prominence during the resource distribution process. For further discussion on conceptual and technical mechanisms for the realization of that model, we refer to [28].

2.1 The Scheduling Problem

Based on the previous system definitions our scheduler has to (a) monitor the actual resource requirements of the applications, (b) dynamically select an optimal modes configuration and (c) ensure schedulability of all application tasks. Obviously, a straight-forward approach would be to perform $|\mathcal{M}_1| \times |\mathcal{M}_2| \times \ldots \times |\mathcal{M}_n|$ feasibility tests for all application mode combinations and to select a feasible configuration that gains the maximal overall quality and optimally utilizes systems resources. However, since tasks may have deadlines not equal to their periods (i.e. $D_{i,k} \leq T_{i,k}$)[3] the complexity for solving the famously NP-hard feasibility problem escalates even further. Even with a fast approximative algorithm (e.g. [3]) solving the whole problem online may lead to inacceptable overhead. Also, it is not clear *when* and *how often* the feasibility of the actual configuration should be checked. For example, on each update of tasks (average) cost estimations? Without further precautions the latter may lead to permanent mode switches and reconfigurations of the system. Indeed, we assume that mode switches create additional *reconfiguration costs* for the apps, e.g. for data conversion, task adaptation, etc., and therefore they should be *omitted as far as possible*. Hence, a defensive reconfiguration strategy is needed, which tries to keep a stable system state as long as possible, but also is reactive enough to quickly detect and adapt on substantial state changes and errors.

3 General Scheduler Overview

Our hierarchical approximation and scheduling model is aimed to *temporarily* and *spatially* distribute the complexity of schedulability tests [29], i.e. the tests are performed only *when*

[2] Its tasks are blocked in a consistent state and temporarily removed from scheduler's control.
[3] $D_{i,k} \geq T_{i,k}$ was not considered due to implementational issues.

and *where* it is necessary in the system. Figure 1 depicts the general architecture of the scheduler. It follows abstractions of tasks, applications and operational modes on different levels. Application behaviour is monitored and statistically approximated on the lowest level. These approximations are forwarded to the upper levels, similar to a cascaded control system. Thereby, the information is further abstracted until on the uppermost level global decisions about the optimal configuration of application modes can be made. The basic idea is to assume schedulability as long as task approximations are valid and to postpone feasibility tests as long as their behaviour does not change considerably. If a task approximation gets invalid its cost estimate $C_{i,k}$ is updated and a local feasibility test is performed only for the concerned application A_i. This results in a new approximation of application's computational demand $B_{i,j}$ for its actual mode $M_{i,j}$. In this case, finding the optimal configuration of application modes on the system level is similar to filling a knapsack with items of different weights $B_{i,j}$ and values $U_{i,j}$.

On the lowest level, application-local schedulers control the execution of tasks and approximate their current costs. These cost estimations take also the observed jitter into account by allocating a dynamic resource buffer (cf. Section 4). As long as the approximations are accurate, no state update is forwarded to the upper scheduler layers. Only if a task bursts out of its current approximation bounds, a new cost estimation value $C_{i,k}$ is computed and notified on the application level.

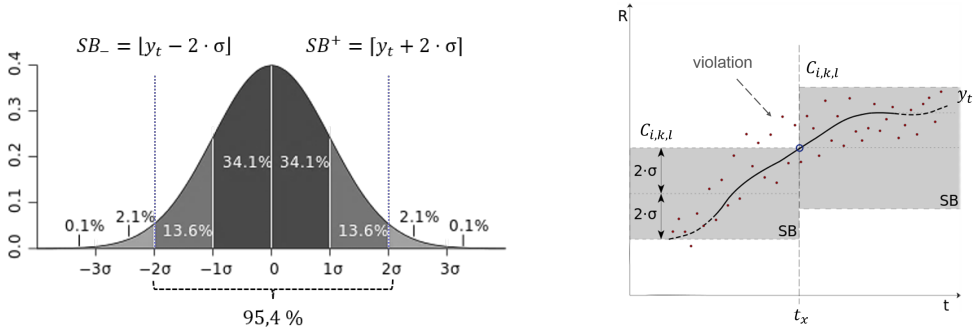
On the next higher level apps are isolated and scheduled with the help of *Constant Bandwidth Servers* (CBS) [1]. The CBS enforce resource reservations called *bandwidths*. We extended the original CBS-algorithm for encapsulation of whole applications. This requires a special methodology for dimensioning of the servers, which will be explained later in Section 5. In this paper, we present the latest version of our resource allocation mechanism. However, when a task cost estimation $C_{i,k}$ is updated, a local feasibility analysis results in a new bandwidth reservation $B_{i,j}$ for its application A_i. Thus, $B_{i,j}$ represents the required amount of resources in order to meet the deadlines of A_i's tasks. The new reservation is then forwarded to the optimization on the system level.

The objective of the optimization is to allocate the available system resources optimally. This is done by selecting and activating a set of app modes that maximize the system's overall utility constrained by its limited resource amount. For this purpose we developed a knapsack-based algorithm which is solved online via dynamic programming (see Section 6).

4 Task Approximation and Scheduling

As already explained, application-local schedulers control the activation of tasks $\tau_{i,k} \in \mathcal{T}_i$ according to a particular scheduling discipline. In fact, any arbitrary policy could be used, even different mechanisms for different applications at the same time. However, the task schedulers are an intrinsic part of our custom CBS mechanism. They determine the feasibility analysis mechanisms applied on the application level. In ARTOS we decided for EDF-based task schedulers. EDF allows for exact schedulability analysis even in case of a high system utilization. Since our framework was developed on top of a fixed priority scheduler (see Section 7), we used a modified version of the *priority shifting* mechanism proposed by Zerzelidis et al. [39, 41] in order to emulate EDF. Our current implementation with RTSJ is briefly discussed in Section 7.2 and further presented and evaluated in [28, 26].

Task schedulers also implement the approximation directives of our model. We instrument the thread-specific POSIX real-time clocks of the Linux kernel and sample the *pure* CPU costs C_t of each task job, i.e. scheduling and blocking effects are not included in C_t. These



(a) Standard deviation confidence intervals

(b) Safety Buffer with $2 \cdot \sigma$

Figure 2 Safety Buffers.

samples are then exponentially smoothed:

$$y_t = \alpha \cdot C_t + (1 - \alpha) \cdot y_{t-1} \tag{1}$$

The exponential smoothing implements a low-pass filter with an infinite impulse response. We chose a smoothing factor of $\alpha = 0.125$ which implies a stronger emphasis on the history $(1 - \alpha) \cdot y_{t-1}$ and therewith a relatively stable average value with respect to high-frequent jitter. However, for our actual task-cost estimation we need a measure for the observed jitter per task. The approximation is established with a dynamic *safety buffer* (SB) for guaranteed resources, which is defined as:

$$\mathrm{SB}_-^+ = [y_t \pm z \cdot \sigma_n] \tag{2}$$

where y_t is an expectation value based on the smoothed average from Equation 1 and σ_n is the standard deviation of the measured task costs C_t in a time window of n task periods. The actual task costs estimation is equal to the upper bound of the safety buffer $C_{i,k} = \mathrm{SB}^+$, while the lower bound SB_- prevents a constant overprovisioning of the costs. With this method fluctuations of the task costs can be algorithmically expressed and the factor z determines the quality of their approximation. According to the confidence intervals of normally distributed random variables a factor of for example $z = 2$ already covers ca. $95,4\%$ of the measured task costs (see Figure 2a). Hence, the actual task behaviour is approximated from the observed history of n periods with a quality of $95,4\%$.

Normal distribution is based on the central limit theorem which proves that the overlap of several independent stochastic processes is almost normally distributed [18]. Regarding the non-deterministic factors influencing the particular execution time of each task job, e.g. cache and TLB occupation, NUMA effects, potential blocking effects and spinning on OS-internal locks during system calls, this approximation model seems to be suitable. Thus, even if task costs are not normally distributed they are calibrated with the standard deviation leading to a certain deviation value σ_n.

In our model the cost estimations $C_{i,k}$ are not computed and updated on each task period. This would be costly because of the computation of σ_n for the last n period samples. In fact, only the exponentially smoothed values y_t are updated and periodically checked against their actual approximation bounds SB_-^+. These checks are part of the rescheduling process at the task level and have a low and constant complexity as well as memory demand. Consequently, each task verifies its own approximation limits on every periodic release. An established

SB is valid as long as the smoothed task costs y_t remain within its bounds (see Figure 2b). In case of $y_t \geq \mathrm{SB}^+$ or $y_t \leq \mathrm{SB}_-$ violations of the $2 \cdot \sigma$ SB-intervall must have occurred with a higher rate than expected, i.e. the task behaviour has changed so far, that it is not approximated correctly anymore. In this case a new deviation σ_n has to be computed and the buffer limits SB_-^+ must be updated (according to Equation 2). This results in a new cost estimation $C_{i,k} = \mathrm{SB}^+$ which is then notified to the application level. However, this strategy works on task-period granularity and approximates cost fluctuations precisely at the expense of a higher system reconfiguration probability. Other strategies may involve more seldom checks in order to reduce reconfigurational overhead and costs, however, at a risk for inaccurate task approximations and temporal faults.

5 CBS Extensions and Dimensioning

This chapter describes novel contributions of our work. As already explained, applications are isolated with separate constant bandwidth servers. A CBS is defined by the ratio of a reserved resource capacity R_i per server period P_i, i.e. a server *bandwidth* $B_i = R_i/P_i$. Since the bandwidths are automatically enforced by the CBS scheduling, applications are protected from overruns of other apps. Moreover, if an app needs more resources than actually reserved, it will not affect other apps but suffer alone from its deficiency. Hence, a mechanism is needed that 1) divides the available system bandwidth between the apps and 2) updates their reservations according to their actual demands. The first mechanism is implemented by the optimization, which decides the configuration of application modes on the system level. The second mechanism is conditioned by the app-local feasibility analysis which is triggered upon task cost updates (cf. Section 4).

Traditionally, the CBS were not designed to serve whole applications with multiple tasks and throughout literature they are mostly used for encapsulation of single tasks. Such a flat model would exacerbate the resource distribution between the applications since it requires additional application abstractions combining the reservations of their tasks. Moreover, it would also create a higher scheduling overhead for a considerable higher amount of servers. However, the original work of Abeni and Buttazzo [1] lists several clauses guiding the definitions and scheduling of the CBS. Indeed, we could not identify any reason or limitation why a CBS could not work well with multiple tasks in parallel. Consequently, we introduce the following specializations and enhancements of the algorithm:

1. **Internal Scheduling Discipline:** The server provides an internal queue for incoming jobs of *different* tasks. The ordering of the queue follows a particular scheduling strategy which does *not* have to be *non-preemptive*. At any time instant there is only one job of the same task in the queue – no interleaved job execution. The latter corresponds to the typical realization of tasks as threads within a particular OS.
2. **Server Dimensioning:** Regarding its capacity R_i and period P_i, a server must be dimensioned so that all internal tasks get enough computational time in order to meet their *own* deadlines. The feasibility of the internal task set must be approved for the allocated server bandwidth.

In ARTOS the first enhancement is fulfilled by the application local task schedulers. Each task scheduler maintains its own run queue ordered by EDF and the system maintains an absolute deadline for each task job. However, there are several points where the scheduling on task-level is interlinked with the CBS on application-level, i.e. on:

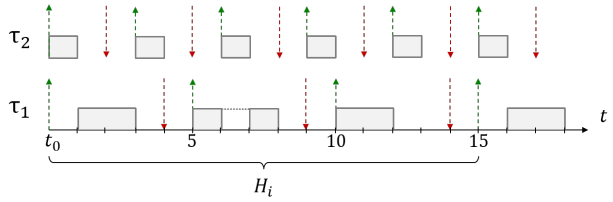


Figure 3 EDF activation example.

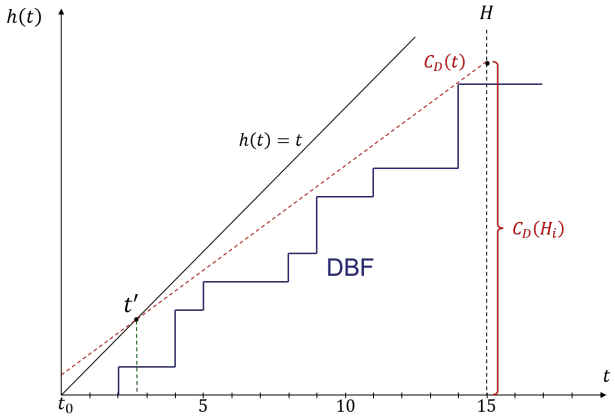


Figure 4 Example for app-local feasibility analysis.

- *Server Deactivation*: Each task scheduler tracks the termination of task jobs with no further pending jobs and notifies a server suspension (cf. Clause 8 in [1]) to the application level.
- *Server Activation*: On each task-job activation, the task scheduler tracks if the CBS is currently inactive and applies the *wake-up rule* if required (cf. Rule 7 in [1]).

Further details on the realization of the interconnection of task- and application-level scheduling can be found in [28, 26].

The second requirement is more complex and therefore described in more detail here[4]. It implies a relationship between the bandwidth of a server, i.e. its period P_i and capacity R_i, and the feasibility of its internal tasks with respect to their own relative deadlines. This relationship is established in our model based on tasks' processor demand [6]. The basic idea is to extract a required server capacity respecting tasks' actual cost estimations $C_{i,k}$ and internal deadlines $D_{i,k}$, which is then split over several server periods. Here we exploit the property of EDF that as long as task costs (estimations) remain unchanged their activation sequence remains identical within their hyperperiod H_i.

5.1 Dimensioning of the server capacity R_i

Figure 3 depicts an example of two tasks $\tau_{i,k} \in \mathcal{T}_i = (T_{i,k}, C_{i,k}, D_{i,k})$ with $\tau_{i,1} = (5, 2, 4)$ and $\tau_{i,2} = (3, 1, 2)$. Both tasks belong to the same application A_i and they are currently executed in a particular application mode $M_{i,j} \in \mathcal{M}_i$. The partial processor utilization of the task set is $U_i = \sum_k C_{i,k}/T_{i,k} = 0,73$ and the hyperperiod $H_i = 15$ is the LCM of the

[4] Again, we present here the latest version of our CBS dimensioning strategy.

task periods. In Figure 4 the cumulative demand bound function (DBF) [6] of the tasks is depicted as a step function. For simplicity we assume a synchronous task start, which is generally known as the worst case [17]. According to Baruah et al. [6] feasibility tests must be performed for all points of discontinuity of the DBF, i.e. the deadline points of the tasks, until $t' = U_i/(1 - U_i) \cdot max(T_{i,k} - D_{i,k}) = 2.75$ which is the intersection point of DBF's approximation tangent $C_D(t) = t \cdot U_i + U_i \cdot max(T_{i,k} - D_{i,k})$ and the angle bisector $h(t) = t$. The task set is feasible if for none of these points the according value of the DBF is greater than $h(t) = t$. As a required capacity for both tasks we use $C_D(H_i)$, which is the value of the approximation tangent in H_i. Hence, as long as the cost estimations of both tasks do not change (cf. Section 4), they need at least a capacity of $C_D(H_i) = 11.73$ in order to meet their deadlines in H_i and in every further repetition of H_i. The value of $C_D(H_i)$ is then normalized with the selected server period P_i and stored as R_i, i.e.:

$$R_i = P_i \cdot \frac{C_D(H_i)}{H_i} . \tag{3}$$

5.2 Dimensioning of the server period P_i

The choice of an appropriate server period is not as obvious as it might seem. In [26] we show based on several experiments, that the server period has to be less or equal the smallest task period within an application A_i, i.e. $P_i = min(T_{i,k})$. Only then enough server preemption points are created, so that their internal tasks can be activated timely in order to meet their internal deadlines. The normalization of the required capacity in Equation 3 can now be interpreted as follows:

- The system bandwidth is dissected over time in a capacity R_i per period P_i. The remainder capacity $P_i - R_i$ is available for all other apps in the system and their own server capacities.
- The capacity $R_i = c_g + c_r$ consist of a base part c_g for the task $\tau_{i,min}$ with the minimal period and a residual fragment c_r for all remaining tasks $\tau_{i,k}$ of A_i with greater periods.
- The base part $c_g = C_{i,min}$ guarantees the completion of task $\tau_{i,min}$ within each server period and, thus, within its own period and deadline.
- The residual fragment c_r consists of the sum of the partial costs of all remaining tasks of A_i, with respect to their own periods and deadlines, relative to the chosen server period, hence $c_r = \sum_{i=1}^{k} \frac{P_i \cdot C_{i,k}}{T_{i,k}} - C_{i,min}$.

Depending on the application-internal task activations and server preemptions during runtime, the actual job of $\tau_{i,min}$ or any other task of A_i may be split over multiple server periods. However, since R_i is warranted for each P_i, meeting of all task deadlines in A_i is guaranteed as long as its task costs do not change.

5.3 Interpretation of the Server Dimensioning

After setting $P_i = min(T_{i,k})$ the server capacity is computed and normalized according to Equation 3. Here the task hyperperiod H_i is inserted in the linear equation of the approximation tangent $c_D(t)$ of tasks DBF and normalized with the chosen period P_i. With the following transformation of the formula:

$$R_i = P_i \cdot \frac{c_D(H_i)}{H_i} \Rightarrow P_i \cdot \frac{(H_i \cdot U_i + U_i \cdot max(T_{i,k} - D_{i,k}))}{H_i} \Rightarrow$$

$$R_i = P_i \cdot U_i \cdot (1 + \frac{max(T_{i,k} - D_{i,k})}{H_i}) \tag{4}$$

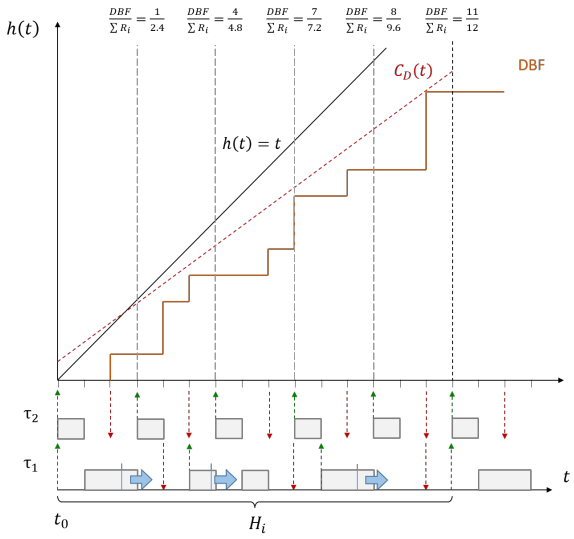


Figure 5 Visual interpretation of the server dimensioning.

it can be easily seen, that for tasks with deadlines equal to periods ($D_{i,k} = T_{i,k}$) the term $max(T_{i,k,l} - D_{i,k,l})/H_i$ gets zero and the required capacity is determined by the partial utilization $U_i = \sum_k \frac{C_{i,k}}{T_{i,k}} = B_i$ of A_i. According to Baruah et al. [6] in case of $D_{i,k} = T_{i,k}$ this necessary feasibility test is also sufficient. With $P_i \cdot U_i$ a base capacity for the $D_{i,k} = T_{i,k}$ case is reserved, which would be sufficient for the tasks in each hyperperiod. With $P_i \cdot U_i \cdot max(T_{i,k,l} - D_{i,k,l})/H_i$ a cumulative fraction of additional capacity per period P_i is reserved which in case of $D_{i,k} \leq T_{i,k}$ is necessary to meet the shorter task deadlines.

For our particular task example in Section 5.1 applying the defined server dimensioning rules leads to a server period $P_i = min(T_{i,k}) = 3$ and a server capacity $R_i = P_i \cdot \frac{c_D(H_i)}{H_i} = 2.4$ time units. Respectively, the allocated server bandwidth is $B_i = R_i/P_i = 0.8$. Figure 5 allows for a graphical interpretation of our reservation mechanism.

The time scale is subdivided in server periods of 3 time units. At each period an increase of 2.4 units of processor demand can be served. Figure 5 shows the relation between the cumulative processor demand increase DBF and the overall provided capacity $\sum R_i$ at the end of each server period. As a feasibility condition $\frac{DBF(t)}{\sum_t R_i} \leq 1$ must hold at the end of all server periods throughout H. For example, at the end of the second server period the task set demands for $DBF = 4$ time units, while the server guarantees so far $\sum R_i = 4.8$ TUs. Because of the capacity limitation of $R_i = 2.4$ per $P_i = 3$ the execution of some task jobs may be intercepted and postponed until the next server period. Since for all existing servers in the system the condition $\sum B_i \leq 1$ must hold true, the remainder capacity of $R_{rmd} = (1 - B_s) \cdot P_i = 0.6$ time units defines the longest possible preemption time of the server and its tasks in our scenario. For example, in Figure 5 in the first period a chunk of 0.6 time units of task costs is shifted to the second period, where afterwards a chunk of 0.2 is moved to the third, a.s.o. These costs are automatically reflected by the increase of tasks' processor demand at the respective server period. The actual preemption phases depend on the activation sequence of all servers and their internal tasks within the system. However, since the CBS scheduling (cf. [1]) automatically enforces the reserved bandwidths, meeting tasks' deadlines depends merely on the coverage of their DBF with sufficient capacity for each server period, independent of their actual activation and preemption sequence.

Our reservation mechanism automatically ensures that coverage by applying the presented processor demand approximation mechanism. However, other approximations such as the presented one by Albers et al. [3] and Chakraborty et al. [13] are possible and part of our future investigations.

6 Mode Decisions and Optimization

Mode decisions are reduced to a knapsack-based optimization problem, which is solved online and produces one of many exact solutions. Although the knapsack problem is NP-hard it can be solved via dynamic programming with pseudo-polynomial complexity. Our solution is described in detail in [31, 29, 26]. It computes an optimal mode selection in one pass and tends to distribute the resources uniformly, i.e. it prefers solutions with a higher admission rate and lower app modes instead of fewer active apps in higher modes gaining the same overall quality. The optimization is triggered when app's priorities or resource estimations $B_{i,j}$ change, or when new apps are activated or depart from the system. Our evaluation in [31] and [26] shows a linear increase of the computation time, when the amount of apps, modes, or system's resource resolution R increase. Thus, the overall computational complexity of the problem is $\mathcal{O}(n \cdot |R|) \cdot \mathcal{O}(|M_{i,j}|)$, where $\mathcal{O}(n \cdot |R|)$ denotes the amount of computed table entries and $\mathcal{O}(|M_{i,j}|)$ is the computational overhead per entry.

In the remainder of this chapter, as a novel contribution, we want to particularly focus a potential extension of our algorithm for further resource types with a multi-dimensional knapsack problem (MKP). Assuming one further dimension for application-specific network traffic[5] and demands, for example represented as mode specific bandwidths $N_{i,j}$, the problem could be defined as:

$$\begin{aligned} \text{Find a selection} \quad J = \; & \{M_{1,j_1}, M_{2,j_2}, ..., M_{n,j_n}\}, \\ \text{maximizing} \quad & \textstyle\sum_{i=1}^{n} a_i \cdot \mathrm{u}_i(M_{i,j_i}), \\ \text{subject to} \quad & \textstyle\sum_{i=1}^{n} B_{i,j_i} \leq R \\ \text{and} \quad & \textstyle\sum_{i=1}^{n} N_{i,j_i} \leq N. \end{aligned}$$

Since we work with bandwidth values, the numerical ranges of both resource dimensions R and N as also their knapsack size is assumed to be equal to 0..1 (i.e., 0 to 100 %). For the solution two matrices $\mathrm{d} \in \mathbb{Z}^3$ and $\mathrm{J} \in \mathbb{Z}^3$, are used for the computation. While d stores the values of partial iteration steps for all three dimensions, i.e. n-applications, R CPU resources and N network resources, J holds the optimal mode selection for each iteration step. For $i \in \{1, 2, ..., n\}$, $r \in \{0, 1, ..., R\}$, $k \in \{0, 1, ..., N\}$ and $M_{i,j} \in \mathcal{M}_i$ let

$$\mathrm{d}(i, r, k) = \textstyle\sum_i a_i \cdot \mathrm{u}_i(M_{i,j_i}) \tag{5}$$

denote the maximum utility such that $\sum_i B_{i,j_i} \leq r$ and $\sum_i N_{i,j_i} \leq k$. Let further

$$\mathrm{J}(i, r, k) = \{M_{1,j_1}, M_{2,j_2}, ..., M_{i,j_i}\} \tag{6}$$

be an optimal selection satisfying the given constraints. At each step the matrix $\mathrm{d}(i, r, k)$ holds the maximum possible utility for the first i applications and the resource boundaries of r and k, while $\mathrm{J}(i, r)$ stores the selection which led to the gained maximal utility.

[5] In practice, up- and down-link should be approximated separately, but we omit this here.

The matrices are initialized with $\mathrm{d}(0,r,k) = 0$ and $\mathrm{J}(0,r,k) = \emptyset$ for all $r = 0,1,...,R$ and $k = 0,1,...,N$. Then for $i \in \{1,2,...,n\}$, $r \in \{0,1,...,R\}$ and $k \in \{0,1,...,N\}$ the following recursion applies:

$$\mathrm{d}(i,r,k) = \max_{M_{i,j} \in \mathcal{M}_i} \{\mathrm{d}(i-1, r-B_{i,j}, k-N_{i,j}) + a_i \cdot \mathrm{u}_i(M_{i,j})\} \tag{7}$$

while with

$$\mathrm{J}(i,r,k) = \mathrm{J}(i-1, r-B_{i,j_i}, k-N_{i,j_i}) \cup \{M_{i,j_i}\} \tag{8}$$

an optimum realizing selection is given. Checking if $r - B_{i,j}$ and $k - N_{i,j}$ still reside within the matrix $\mathrm{d}(i,r,k)$ ensures observation of the constraints $\sum_{i=1}^{n} B_{i,j_i} \leq R$ and $\sum_{i=1}^{n} N_{i,j_i} \leq N$. In the following the pseudo code of the algorithm which was presented in [31, 29, 26] is extended with the new definitions:

Algorithm 1 Optimization Algorithm Pseudo Code

```
1: for r = 0 → R and k = 0 → N do
2:    d[0][r][k] ← 0, J[0][r][k] ← ∅
3: end for
4: for i = 1 → n do
5:    d[i][0][0] ← 0
6:    for r = 1 → R do
7:       for k = 1 → N do
8:          max ← 0
9:          for M_{i,j} ∈ M_i do
10:            b ← u[i−1][r − R_{i,j}][k − N_{i,j}] + a_i · U_i(M_{i,j})
11:            if b > max then
12:               max ← b
13:               J[i][r][k] ← J[i−1][r − R_{k,j}][k − N_{i,j}] ∪ {M_{i,j}}
14:            end if
15:         end for
16:         u[i][r][k] ← max
17:      end for
18:   end for
19: end for
20: return J[n][R][N]
```

As can be seen, we basically extended our previous algorithm with another dimension N. That leads to a further nested iteration loop over its value range (Line 7) and respective resource probing (Line 10). Consequently, with each new resource X the complexity of the algorithm is extended by a factor $|X|$, i.e. the numerical resolution of that resource. I.e., for our network resource based extension the computational complexity of our algorithm is $\mathcal{O}(n \cdot |R| \cdot |N|) \cdot \mathcal{O}(|M_{i,j}|)$, where $\mathcal{O}(n \cdot |R| \cdot |N|)$ denotes the amount of computed table entries and $\mathcal{O}(|M_{i,j}|)$ is the computational overhead per entry. MKPs generally suffer from the so called "curse of dimensionality", i.e. a substantial effort added with each further dimension. For that reason approximative dynamic programming approaches (like [33]) might help to solve the problem faster but at a risk of a suboptimal solution. However, such an approximative approach is part of our future work.

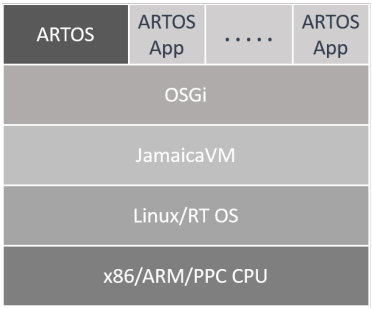


Figure 6 ARTOS Software Stack.

7 Implementation

We implemented our scheduling and application model and integrated them within a Java-based application framework and runtime system. In this Section, we describe the general hard- and software configuration, which was used also during our evaluation and experiments. We will then accentuate on some particular peculiarities of our implementation.

7.1 System Configuration

Figure 6 outlines the general software stack of our system. Our application and scheduling model were implemented with RTSJ 1.0.2 [8], which offers basic mechanisms for real-time application programming, i.e. periodic/aperiodic realtime threads, a FIFO priority-based scheduler, priority inheritance and ceiling monitor protocols, a bounded garbage collection interference, asynchronous events with controlled handler execution, cost monitoring and optional enforcement, server-like mechanisms and processing groups, etc. In order to support a dynamic provisioning and execution of applications and code, we decided for OSGi as a middleware. Its lightweightness, compositional abilities and service support allow for an easy integration of executable components (bundles) and Java-based code during runtime, without the necessity for a system restart. However, the ARTOS application API and scheduling code can be exported as bundles and, thus, integrated as an abstraction layer into any particular OSGi framework. In our experiments we used *Concierge* [15] which is a performance-optimized OSGi-R3-compliant framework for small and embedded devices. Applications which make use of our application and scheduling API *(ARTOS Apps)* are deployed as bundles within a running system and automatically registered within our scheduling primitives when activated.

JamaicaVM [19], which is developed by Aicas GmbH, is one of many RTSJ-compliant real-time JVMs. We chose this particular JVM for our experiments because it implements the full scope of the RTSJ spec, i.e. also its optional features like cost monitoring for threads and thread groups. These mechanisms were necessary for our custom CBS implementation on the application level of our scheduler. Beyond traditional interpreted execution JamaicaVM provides an *ahead-of-time* native compiler, which allows for better integration and runtime performance of Java programs and code especially on embedded devices. The compiled native binary of ARTOS contains a stripped version of the required RTSJ environment and runs as a root process on a particular OS and hardware platform. Thereby, RTSJ threads are mapped 1:1 to processes and threads of the underlying OS. Furthermore, the priority space of RTSJ can be mapped directly to a particular scheduling policy and respective priority

range[6] in a user-defined way. On this way, interference of other system and user processes can be avoided and controlled. On the other side, priority modifications of RTSJ real-time threads directly affect their native priorities and scheduling order within the underlying OS. JamaicaVM supports in general x86, ARM, and PPC platforms and a variety of operating systems, like Linux, VxWorks, QNX, etc. These configuration variants underline the broad range of applications supported alone by our system prototype. For our experiments we used a traditional Desktop PC with an Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz processor and 16 GB RAM. As an OS, we used a Linux 3.10.65 kernel, which was extended with the `CONFIG_PREEMPT_RT` [34] patch. Similar x86- and Linux-based configurations are applicable for a variety of embedded devices and they neatly integrate with JamaicaVM.

A traditional Linux kernel allows a process to preempt another process only at certain circumstances, i.e. (a) when the CPU executes user-mode code, (b) while leaving kernel-space, after a system call or interrupt has been handled, (c) when the kernel code blocks on a mutex or (d) explicitly yields control to another process. Thus, a long running low-priority process within the kernel may delay a request of a high-priority thread for hundreds of milliseconds, which would compromise the performance of any real-time system. With the `CONFIG_PREEMPT` option Linux allows kernel code to be entirely preemptable, except of particular spin-lock–protected regions and during interrupt handling. However, this lowers preemption latencies to a couple of milliseconds. For even lower latencies the `CONFIG_PREEMPT_RT` patch makes the kernel fully preemptable. For this, interrupt handlers are wrapped in kernel threads and processed with own fixed priorities next to user processes. Furthermore, the patch transforms particular kernel-intern locks to preemptive `rtmutexes` [36], which are enriched with priority-inversion avoidance protocols, like *priority inheritance*. JamaicaVM again builds on these mechanisms for its object monitors, threading system and internal scheduling.

7.2 Implementation Peculiarities

The implementation of our model based on RTSJ is presented more detailed in [28, 26]. However, for periodic tasks we used `RealtimeThread`s bound to special `PeriodicParameters` containing start, period, costs[7] and a relative deadline ($D_{i,k} \leq T_{i,k}$) definitions for each task. Task modes were realized with an RTSJ technique called Asynchronous Transfer of Control (ATC). ATC allows the definition of dynamically selectable and asynchronously interruptible code instances. Thus, multiple versions of the same task (i.e. modes) can be provided in parallel and are decided on each job activation. In case of a deadline miss or mode switch, the actual task job can be immediately interrupted. Beside basic task and mode abstractions our API provides semantics for applications consisting of several independent tasks with multiple modes. These apps are integrated with OSGi's module and lifecycle management mechanisms.

However, although RTSJ syntactically provides abstractions for custom thread scheduling, the implementation of such mechanisms is restricted [40]. As a consequence, we emulate EDF on top of RTSJ's fixed priority scheduler with a modified version of the *priority shifting* mechanism in [39, 41]. Thereby, the tasks perform a cooperative scheduling by shifting their own priorities and the priorities of their neighbours between three virtual priorities `preempted`, `active` and `scheduling`. First, an absolute deadline is maintained for each task and updated on each periodic release. For each application the tasks are ordered into a

[6] E.g. in Linux `SCHED_FIFO` [38] supports 100 fixed priorities whereas RTSJ prescribes at least 28 priorities.
[7] Since we did not rely on RTSJ's cost enforcement, the task costs are only informative.

separate run queue relative to their current absolute deadlines. On each suspension and activation a task $\tau_{i,k}$ is allowed to reassign the virtual priorities of all tasks in the same application A_i according to their current absolute deadlines, i.e. to select one active and to preempt all other tasks. For this, all tasks suspend at the end of each period and, thus, release again with a `scheduling` priority, capable to shortly preempt their neighbours and to re-schedule. The virtual application priorities are dynamically mapped and shifted by the system between certain regions within the fixed priority space, according to an application scheduling discipline. The latter is defined by the application level of our scheduler, i.e. the CBS scheduling mechanism.

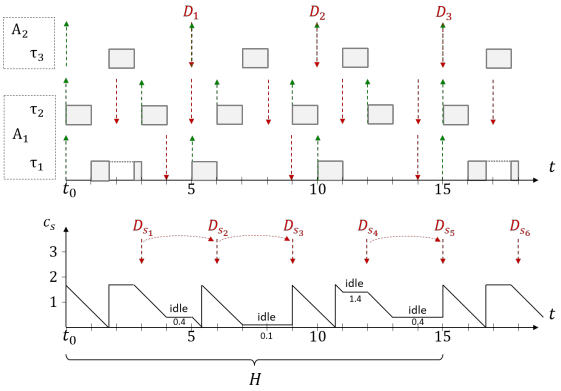
RTSJ provides basic support for custom server mechanisms through (implicit) *processing groups*. Each group is assigned a *budget* and a *reservation period*. The real-time JVM monitors the costs of each group and will suspend all threads if their budget is depleted. However, the budget is automatically replenished at the beginning of the next reservation period and the threads are then resumed. Consequently, this mechanism is not appropriate for a CBS, but for example for a Deferrable Server [37]. We extended the processing groups specification with custom budget replenishment and retrieval techniques for user-defined server mechanisms. The extensions were included in the latest RTSJ 2.0 version. With the help of Aicas, a modified version of the JamaicaVM supporting the suggested mechanisms was developed and used as a prototype for our evaluation. Our particular CBS implementation and scheduling are explained in detail in [28]. For scheduling we used a similar priority shifting mechanism like on the task-level, but now whole applications are moved between two *priority bands* (`ACTIVE` and `PREEMPTED`). As described in Section 5 our CBS scheduling is interlinked with the dispatching on task level where server activation and deactivation are traced per task and notified to the application scheduler for intervention. Budget depletion in turn is automatically tracked by the JVM for each processing group and notified via an asynchronous event. During runtime the CBS scheduling applies according to [1] and the extensions in Section 5 while the tasks perform their application-local cooperative scheduling.

8 Evaluation And Results

For our evaluation scenarios we developed special *artificial apps* which follow predefined load profiles, e.g. emulate certain costs, burst and jitter behaviour. Our experiments involved several tests with focus on (a) correct scheduling, i.e. the interplay between CBS and internal task scheduling, (b) systems adaptational capabilities and reactiveness, (c) suppression of cyclic task bursts and (d) the overall quality benefit of our model. The full spectrum of our experiments can be examined in [26], here we accentuate our latest results for (a) and (d).

8.1 Scheduling

We compared the theoretical schedules of different scenarios with the activation trace generated by our scheduler during runtime. For that we instrumented a `TraceMonitor` app provided by JamaicaVM. An example of its output can be seen in Figure 8. Our experiments involved: a single app with multiple tasks (task-level scheduling), multiple apps with single tasks (CBS scheduling) and multiple apps with multiple tasks (task and CBS scheduling). Here we show an example only for the last case, which involves scheduling on both task and application level. Two apps were involved: A_1 with two tasks $\tau_{i,k} \in \mathcal{T}_i = (T_{i,k}, C_{i,k}, D_{i,k})$:



(a) Theoretical schedule (all values in 10^2 ms)

t	Action
1.7	CBS_1 overrun and postponement of D_{s_1} to D_{s_2}
5.0	CBS_1 is *idle* with capacity rest of $c_s = 0.4$. Release of τ_1 and τ_3. Since $0.4 \leq (6-5) * 0.57$, τ_1 is served with rest capacity and D_{s_2}. For τ_3 the new deadline is D_2.
5.4	CBS_1 overrun and postponement of D_{s_2} to D_{s_3}. Since $D_{s_3} = 9 < D_2 = 10$, $\tau_1$and τ_2 in A_1 are then executed.
7	τ_2 suspends and CBS_1 remains *idle* with $c_s = 0.1$. A_2 and τ_3 are now active.
9	CBS_1 and CBS_2 are *idle* and all tasks suspend. τ_2 is activated and with $0.1 \geq (9-9) * 0.57$ $D_{s_4} = 12$ and $c_s = 1.7$ are set.
10.7	CBS_1 overrun and postponement of D_{s_4} to D_{s_5}. A_1 remains active since $D_{s_5} = D_3$.
11	A_1 suspends with $c_s = 1.4$. A_2 and τ_3 are now active.
12	τ_2 activates CBS_1 and with $1.4 \leq (15-12) * 0.57$ task τ_1 is served with its rest capacity and D_{s_5}.

(b) Comments on the execution

Figure 7 Artificial test scenario.

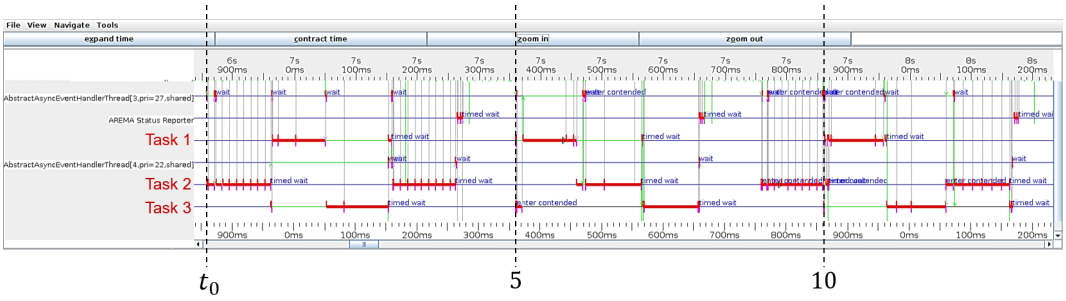


Figure 8 `TraceMonitor` output.

$\tau_1 = (5, 1, 4)$ and $\tau_2 = (3, 1, 2)$, and A_2 with one task $\tau_3 = (5, 1, 5)$[8]. The respective CBS configuration for both apps is shown in Table 1.

Figure 7a shows the theoretical execution plan of the applications and their tasks for a synchronous start, according to the original CBS rules in [1] and our hierarchical extensions in Sect. 5. On the bottom, the CBS state of A_1 is shown, the CBS of A_2 is of less interest, since it encapsulates only one task. Figure 7b shows a legend for certain time events and decisions. Several events occur in that example, e.g. server deadline postponements because of capacity depletion (e.g. $t = 1.7, 5.4, 10.7$) and server deactivations (e.g. $t = 4.0, 7.0, 11.0, 13.0$), as well as server activations with (e.g. $t = 0, 9, 15$) and without (e.g. $t = 5.0, 12.0$) capacity replenishment. The latter occur especially when the CBS wake-up rule applies. Even an application preemption can be seen at $t = 1.7$ because of a server deadline postponement in A_1. However, the schedule is valid and all tasks meet their own deadlines.

Figure 8 shows the real behaviour of our system (after some execution time). The task execution phases are outlined as red lines on the time line, and we also marked indices to the original time line of the theoretical schedule. The results show almost identical theoretical and runtime traces with slight deviations due to RTSJ's event handling procedure. Nevertheless, the dynamic resource budgeting for the apps assured a fault-free execution of their tasks in all experiments.

[8] All values are given in hundreds of milliseconds.

Table 1 CBS configuration for A_1 and A_2.

(in ms)	Period (P_i)	Capacity (R_i)	Bandwidth (B_i)
CBS_1	300	170	57 %
CBS_2	500	100	20 %

8.2 Adaptation

We emulated a sudden bottleneck caused by a burst or arrival of an artificial task. The reactiveness of our system is determined by the burst intensity and the smoothing factor α in Equation 1. A small factor requires a higher burst amplitude and a longer duration for its detection, but induces less reconfigurations. The actual adaptation duration is determined by the costs for 1) feasibility analysis, 2) optimization and 3) mode switches of the affected applications. The costs for feasibility analysis are non-deterministic and depend on the test limit t' (cf. Section 5.1), which in turn depends on the partial utilization U_i of the respective app and the maximal gap between tasks' periods and deadlines. The mode switch costs are application specific and non-deterministic as well.

8.3 Cyclic Bursts

Depending on the size of the history window for the standard deviation σ_n (cf. Equation 2) bursts may be considered as jitter within tasks' cost approximations. In this case the size of the approximation buffer SB^+_- increases. As a consequence, cyclic bursts are automatically approximated and recurring system reconfigurations suppressed [27]. The size of the σ_n window controls the frequency of the burst that can be balanced. Low frequent bursts need a bigger window but possibly lead to a stronger overprovisioning. Small windows in turn approximate more precisely, but lead to more frequent reconfigurations. An automatic adaptation of the window size according to the most prominent burst frequency after spectral analysis of tasks costs is currently in work. However, spare capacities caused by task costs overprovisioning can be instantly shared with other CBS based on a mechanism like CASH [12].

8.4 Overall quality benefit

We developed a real video-on-demand (VoD) streaming application based on our system model [26]. The VoD client was integrated within our framework and supported various modes with different video quality, e.g. different frame rate, resolution and frame quality. During runtime a bottleneck was created via an artificial app forcing the client to request a lower quality from the sender. The latter in turn immediately changes its video encoding format and adapts its sending rate to the requested frame rate. The experiments were done with and without the degradational mechanisms of our system. We supposed that a schedule that causes fewer deadline misses for the cost of lower video quality generally produces a better user experience (i.e. a higher utility), than one with the best video quality in presence of many temporal faults. For such a trade-off we compared frame-wise the peak signal to noise ratio (PSNR) of the received and decoded streams with their originals on the sender side. Our expectation was that, without degradation client's tasks will suffer more frequently from deadline misses leading to a higher signal corruption and thus lower PSNR value. With

Table 2 VoD application mode configurations.

Mode	Resolution	Framerate	CPU Bandwidth
1	480x270	10	10 %
2	1280x720	20	18 %
3	1920x1080	30	42 %

Table 3 PSNR results

Average PSNR	81.28
Maximum PSNR	99.00
Minimum PSNR	4.0
Average of Values < 99 dB	7.6

(a) without degradation

Average PSNR	72.08
Maximum PSNR	99.00
Minimum PSNR	7.06
Average of Values < 99 dB	44.66

(b) with degradation

degradation a lower overall quality is requested but there are no missing or corrupted parts in the received video, leading to a higher PSNR value.

In order to omit transmission errors and delay the experiments were performed within an isolated network. Table 2 shows the configuration of the quality levels of the VoD application and their estimated resource demands during runtime. According to the frame rate of each mode, clients tasks synchronize with periods equal to 100, 50 and 33 milliseconds and deadlines equal to periods ($D_{i,k} = T_{i,k}$). A miss may be caused merely by a deficit of execution time for the tasks. In such cases the client was instrumented to display and store a black frame for the respective video position, i.e. to mark a quality loss. In our experiments the VoD app was first started in its highest mode 3. A bottleneck was then caused with an artificial app which was activated at a certain time and led to a system overbooking of ca. 110 %. While without degradation the rest of the playback caused around 200 deadline misses within the VoD app, with degradation the app was immediately switched to mode 2 without further occurrence of temporal faults. Table II shows the results of the PSNR tests for the transmitted videos.

A bigger PSNR value shows a higher equivalence of the video signals, while a value of 99 dB in our example indicates almost identical parts. The results show that in both scenarios there were identical (max. PSNR) and completely different parts (min. PSNR). Obviously, the occurrence of deadline misses in our scenario without degradation was not frequent enough in order to drastically reduce the overall average video quality (avg. PSNR). Also, a lot of misses came at the expense of the artificial app itself which caused the bottleneck. With respect to that, the requested lower quality level in the degradation scenario reduced the PSNR to an even lower average value. However, for the parts where misses have occurred (avg. $<$ 99 dB) the average quality was improved by ca. 36 %. This value clearly depends on the applications' quality level definitions, the choice of one particular level as also the strength and the length of the bottleneck. Nevertheless, the results confirm that during transient bottlenecks our system preserves a better average quality than without its mode adaptation mechanisms.

9 Related Work

One of the first well-founded approaches for tasks with multiple modes and a quality driven scheduling are the *imprecise computations* (IC) [24]. A base-task quality is guaranteed by assuring mandatory parts of the tasks. This accords to an admission test which requires knowledge about their (worst-case) execution times. However, residual capacity is divided between optional task parts with the use of several heuristic approaches. Some for example tend to minimize the generated divergence from the best possible quality while others ascertain a particular probability for particular optional parts. The IC approach is easily reproducible with our model. However, what IC does not consider is a guideline when a real system must react and spend the effort for decisions and reconfigurations. Here we have a clear definition based on our approximation model. The consequence is that an IC-based system has to re-check its actual service quality periodically leading to a conflict between too frequent reconfigurations and a risk of suboptimality. Several system models base their mode-decision phase on IC. FCS [25] for example implements a feedback control loop, which periodically adjusts the overall system utilization and task quality levels.

Several related works have emerged out of the IRMOS and FRESCOR projects. In [2] and [32] a flat model of one task per CBS is presented with a control based approach for dynamically calibrating tasks' allocated bandwidths. A feedback loop per task is used to control its local scheduling error, i.e. the difference between its virtual finishing time and its job deadline. The objective of the controller is to keep the scheduling error per task with a certain probability within an acceptable range. For this, it uses a predictor to estimate tasks costs evolution and stepwise increases or reduces its allocated bandwidth. This includes also a recovery strategy with a finite error regeneration period for the case of a violation. A second loop with a supervisor is used to restrict an overbooking of systems overall available bandwidth. In case of an overload, the supervisor greedily resets reservations to at least minimal bandwidths for different classes of tasks. In [16] a version of the dual loop scheduler even supporting different task modes is presented. A global QoS controller periodically decides quality levels for the whole task set for different resource dimensions – even zero modes for the tasks are supported. Herefore, a greedy approach based on heuristics with a multipath search through the solution space is proposed.

The previous works ([2, 32, 16]) generally target at similar objectives as our model. However, our basic application abstraction provides a practical solution for tasks that have to be configured differently and synchronized accordingly for each particular application mode. This is not supported by the cited works where modes are effectively decided periodically on a per-task basis. However, consolidating tasks' requirements to common application bandwidths allows for a simple knapsack-based algorithm for the selection of optimal application modes (see Section 6). Since hereby the size of the problem domain is considerably reduced, this promises a lower complexity and a faster solution of the optimization problem. Instead of convergence to an acceptable result using a multipath search through solution space, our optimization directly computes an exact configuration, even in the multi-dimensional case. However, how the presented control mechanisms in [2, 32, 16] would apply for a set of tasks being concurrently served by the same CBS remains open for discussion. Since the tasks now interfere within the same server activation period, their scheduling errors have to be convoluted leading to a common server budget. For that, our approach proposes a simple mechanism based on tasks' processor demand and an application-local feasibility test, while the actual budget calibration steps are minimized. In fact, the algorithm in [16] penalizes applications frequently switching modes in order to minimize re-configuration costs and

attain a steady QoS. This may obstruct the quality of a potential optimal solution. Our model in turn provides an adaptive cost overprovisioning mechanism on task level in order to suppress frequent re-configurations, which is even capable to smooth periodic task bursts, as shown in Section 8. Thus, mode switches occur with the probability of tasks to unexpectedly burst out of their approximations or in case of recurring burst with a variable rate.

Recently, the `SCHED_DEADLINE` [21] scheduler in Linux was extended with a herarchical group scheduling mechanism based on the Linux *CGroups*. The originally flat model of this scheduler, which encapsulates each task in its own CBS, was extended with a second-level scheduler per CBS, i.e. while globally scheduled with EDF the servers allow for an internal fixed-priority–based task scheduling. However, while ongoing works focus on appropriate group scheduling in the presence of multiple processors, `SCHED_DEADLINE` still does not support dynamic reservations for the groups according to actual task demands. Consequently, server bandwidths are configured and allocated statically in advance, guided by respective CGroup definitions, while applications are admitted within the system. Thereby, a simple admission test based on partial task utilizations assures that the system is not overloaded.

In [23] a hierarchical CBS approach (H-CBS) is presented which guarantees temporal isolation between subsets of tasks (i.e. applications). The original CBS mechanism is extended by a virtual time and a scheduling deadline per task, and an excess capacity per task subset. Partial task utilizations are known and used for initial capacity setup, while the virtual times control the observance of task-specific reservations. However, all tasks are scheduled globally by EDF according to their scheduling deadlines, and when a task executes it uses the excess capacity of its subset (application). Meanwhile, H-CBS maintains different execution states for the tasks and according rules for updating their virtual time and deadline, as well as the excess capacity of the whole set. The algorithm ensures temporal isolation on task level as well as between different task subsets. However, the H-CBS algorithm seems still to support a flat task execution model, while the group relationship is encoded within the virtual time and excess capacity update rules for each task. In our approach we have one CBS per app instead with an internal scheduling discipline and run queue. However, it is not clear how the H-CBS mechanism would have to be extended in order to respect arbitrary task deadlines (e.g. $D_{i,k} \leq T_{i,k}$), since now temporal correctness on task level would be compromised. On the other side, no effort has been spent on re-calibrating the excess capacity according to actual task demands (initial task utilizations are still used for all updates). The partial utilization of a subset is assumed to be a static portion of the shared processor resource.

There is a magnificent number of works on optimal CBS dimensioning, for example such minimizing the amount of task preemptions [9] or avoiding the so called *deadline aging* problem [7]. What can be remarked here is that due to the optimal and dynamic adjustment of application capacities we observed a much lesser amount of CBS deadline postponements, which antagonizes the deadline aging problem.

10 Conclusion

In this paper we presented latest improvements of our hierarchical approximation and scheduling model for open soft real-time systems with a dynamic set of applications and tasks supporting multiple execution modes. These enhancements particularly apply to the dynamic server dimensioning and hence resource reservation strategy for applications. Moreover, we exemplified how our optimization mechanism, which is used for selection of an optimal application mode configuration, can be extended for further resource types, e.g. network bandwidth. We also explained the particular configuration and peculiarities

of our current implementation, which neatly integrates as a component-based real-time application framework on various OS and hardware architectures. Besides, latest evaluation results and experiments were discussed in particular proving the correctness of our scheduler and the adaptivity of our system on different load and jitter conditions. A concrete VoD application scenario was used to examine the overall quality benefit of our system especially in overload conditions. We are currently extending our model for distributed compute clusters and real-time HPC applications with multiple parallel tasks based on RT-DLT [22]. Our approach promises to enable a higher overall cluster utilization and throughput while retaining the best possible application quality and performance.

References

1 L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings 19th IEEE Real-Time Systems Symposium*, RTSS'98, pages 4–13. IEEE Computer Society, 1998. doi:10.1109/REAL.1998.739726.

2 L. Abeni, T. Cucinotta, G. Lipari, L. Marzario, and L. Palopoli. Qos management through adaptive reservations. *Real-Time Syst.*, 29(2-3):131–155, March 2005. doi:10.1007/s11241-005-6882-0.

3 K. Albers and F. Slomka. Efficient Feasibility Analysis for Real-Time Systems with EDF Scheduling. In *Proceedings of the Conference on Design, Automation and Test in Europe – Volume 1*, DATE'05, pages 492–497. IEEE Computer Society, 2005. doi:10.1109/DATE.2005.128.

4 AREMA – Adaptive Runtime Environment for Multimode Applications. https://sourceforge.net/projects/arema/, September 2016.

5 ARTOS. https://www.uni-ulm.de/en/in/vs/res/projects/artos/, April 2017.

6 S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings 11th Real-Time Systems Symposium*, pages 182–190, Dec 1990. doi:10.1109/REAL.1990.128746.

7 A. Biondi, A. Melani, and M. Bertogna. Hard Constant Bandwidth Server: Comprehensive formulation and critical scenarios. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, pages 29–37, June 2014. doi:10.1109/SIES.2014.6871182.

8 G. Bolella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.

9 G. Buttazzo and E. Bini. Optimal dimensioning of a constant bandwidth server. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 169–177, Dec 2006. doi:10.1109/RTSS.2006.31.

10 G. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency*. Springer US, 2005. doi:10.1007/0-387-28147-9.

11 G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Trans. Comput.*, 51(3):289–302, March 2002. doi:10.1109/12.990127.

12 M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proceedings of the 21st IEEE Conference on Real-time Systems Symposium*, RTSS'10, pages 295–304. IEEE Computer Society, 2000.

13 S. Chakraborty, S. Kunzli, and L. Thiele. Approximate schedulability analysis. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, pages 159–168, 2002. doi:10.1109/REAL.2002.1181571.

14 S. Chakraborty, M. Lukasiewycz, C. Buckl, S. Fahmy, N. Chang, S. Park, Y. Kim, P. Leteinturier, and H. Adlkofer. Embedded systems and software challenges in electric vehicles. In

Proceedings of the Conference on Design, Automation and Test in Europe, DATE'12, pages 424–429. EDA Consortium, 2012.

15 Concierge OSGi. http://concierge.sourceforge.net/, April 2009.

16 T. Cucinotta, L. Palopoli, L. Abeni, D. Faggioli, and G. Lipari. On the integration of application level and resource level qos control for real-time applications. *IEEE Transactions on Industrial Informatics*, 6(4):479–491, Nov 2010. doi:10.1109/TII.2010.2072962.

17 L. George, N. Rivierre, and M. Spuri. Preemptive and Non-Preemptive Real-Time Uni-Processor Scheduling. Research Report RR-2966, INRIA, 1996. Projet REFLECS.

18 N. Henze. *Stochastik für Einsteiger*. Springer Fachmedien, Wiesbaden, 10 edition, 2013. doi:10.1007/978-3-658-03077-3.

19 JamaicaVM. https://www.aicas.com/cms/en/JamaicaVM, February 2017.

20 G. Koren and D. Shasha. Skip-over: algorithms and complexity for overloaded systems that allow skips. In *Proceedings 16th IEEE Real-Time Systems Symposium*, pages 110–117, Dec 1995. doi:10.1109/REAL.1995.495201.

21 J. Lelli, C. Scordino, L. Abeni, and D. Faggioli. Deadline scheduling in the linux kernel. *Software: Practice and Experience*, 46(6):821–839, 2016. doi:10.1002/spe.2335.

22 X. Lin, A. Mamat, Y. Lu, J. Deogun, and S. Goddard. Real-time scheduling of divisible loads in cluster computing environments. *Journal of Parallel and Distributed Computing*, 70(3):296–308, March 2010. doi:10.1016/j.jpdc.2009.11.009.

23 G. Lipari and S. Baruah. A hierarchical extension to the constant bandwidth server framework. In *Proceedings Seventh IEEE Real-Time Technology and Applications Symposium*, RTAS'01, pages 26–35. IEEE Computer Society, 2001. doi:10.1109/RTTAS.2001.929863.

24 J. W. S. Liu, W. K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise computations. *Proceedings of the IEEE*, 82(1):83–94, Jan 1994. doi:10.1109/5.259428.

25 C. Lu, J. A. Stankovic, S. H. Son, and G. Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms*. *Real-Time Syst.*, 23(1/2):85–126, July 2002. doi:10.1023/A:1015398403337.

26 V. Nikolov. *Ein hierarchisches Scheduling Modell für unbekannte Anwendungen mit schwankenden Ressourcenanforderungen*. PhD thesis, Ulm University, 2016. doi:10.18725/OPARU-4099.

27 V. Nikolov, F. J. Hauck, and L. Schubert. Ein hierarchisches Scheduling-Modell für unbekannte Anwendungen mit schwankenden Ressourcenanforderungen. In *Betriebssysteme und Echtzeit*, Informatik aktuell, pages 49–58. Springer Berlin Heidelberg, 2015. doi:10.1007/978-3-662-48611-5_6.

28 V. Nikolov, F. J. Hauck, and S. Wesner. Assembling a framework for unknown real-time applications with rtsj. In *Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES'15, pages 12:1–12:10. ACM, 2015. doi:10.1145/2822304.2822318.

29 V. Nikolov, K. Kempf, F. J. Hauck, and D. Rautenbach. Distributing the Complexity of Schedulability Tests. In *21st IEEE Real-Time and Emb. Techn. and Appl. Symp.*, RTAS 2015, page 2. IEEE, 2015.

30 V. Nikolov, S. Kächele, F. J. Hauck, and D. Rautenbach. Cloudfarm: An elastic cloud platform with flexible and adaptive resource management. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, UCC'14, pages 547–553. IEEE Computer Society, 2014. doi:10.1109/UCC.2014.84.

31 V. Nikolov, M. Matousek, D. Rautenbach, L. D. Penso, and F. J. Hauck. ARTOS: system model and optimization algorithm. Technical Report VS-R08-2012, Inst. of Dist. Sys., University of Ulm, 2012.

32 L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. Aquosa – adaptive quality of service architecture. *Softw. Pract. Exper.*, 39(1):1–31, January 2009. doi:10.1002/spe.v39:1.

33 W. B. Powell. What you should know about approximate dynamic programming. *Naval Research Logistics (NRL)*, 56(3):239–249, 2009. doi:10.1002/nav.20347.

34 PREEMPT_RT. https://rt.wiki.kernel.org, January 2016.

35 Wind River. A Smart Way To Drive ECU Consolidation. https://www.windriver.com/whitepapers/automotive/a-smart-way-to-drive-ecu-consolidation/, January 2017.

36 RT-mutex implementation design. https://www.kernel.org/doc/Documentation/locking/rt-mutex-design.txt, January 2017.

37 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 Comp.*, 44(1):73–91, January 1995. doi:10.1109/12.368008.

38 The Open Group and IEEE. IEEE Std 1003.1. The Open Group technical standard base specification, Issue 6, Base Definitions. http://www.opengroup.org/onlinepubs/009695399/mindex.html, 2004.

39 A. Zerzelidis and A. J. Wellings. Getting more flexible scheduling in the RTSJ. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*, pages 8 pp.–, April 2006. doi:10.1109/ISORC.2006.38.

40 A. Zerzelidis and A. J. Wellings. Getting more flexible scheduling in the RTSJ. In *Proc. of the 9th IEEE Symp. on Obj.-Oriented Real-Time Distrib. Comp.—ISORC*, pages 3–10, 2006.

41 A. Zerzelidis and A. J. Wellings. A framework for flexible scheduling in the rtsj. *ACM Trans. Embed. Comput. Syst.*, 10(1):3:1–3:44, August 2010. doi:10.1145/1814539.1814542.