Software Tools for Technology Transfer — The final publication is available at link.springer.com

Temporal Isolation in Real-Time Systems*

The VBS Approach

Silviu S. Craciunas **Christoph M. Kirsch** **Hannes Payer** **Harald Röck** **Ana Sokolova**

University of Salzburg, Austria

Abstract. Temporal isolation in real-time systems allows the execution of software processes isolated from one another in the temporal domain. Intuitively, the execution of a process is temporally isolated if the real-time behavior of the process is independent of the execution of the other concurrently scheduled processes in the system. The article provides a comprehensive discussion of temporal isolation through variable-bandwidth servers (VBSs). VBS consists of an EDF-based uniprocessor scheduling algorithm and a utilization-based schedulability test. The scheduling algorithm runs in constant time modulo the time complexity of queue management. The schedulability test runs in time linear in the number of processes and enables admission of an individual process in constant time. The test is a sufficient condition for VBS to provide temporal isolation through lower and upper response-time bounds on processes. We present the VBS design, implementation, proofs, and experiments, followed by condensed versions of results on scheduler overhead accounting with VBS and on reducing power consumption in VBS systems.

1 Introduction

Temporal isolation can be seen as a quantified version of scheduler fairness that is particularly relevant to real-time systems. Temporal isolation enables real-time programming models that are compositional with respect to real-time behavior. It is thus an important prerequisite of robust and scalable engineering methodologies for real-time systems.

In this article, we extend and complete the discussion of temporal isolation through variable-bandwidth servers (VBSs) from [1]. The process model of VBS is based on the concept of actions as pieces of sequential process code. A VBS process is modeled as a potentially infinite sequence of actions, which allows us to define response times, and thus temporal isolation of processes, at the level of individual actions. The response time of an action is defined as the duration from the time instant when process execution reaches the beginning of the action (arrival) until the time instant when process execution reaches the beginning of the next action (termination). The VBS scheduler is based on the well-known EDF mechanism [2] and executes processes in temporal isolation. More precisely, with VBS the response time as well as the variance of the response time (jitter) of a given action is bounded independently of any other actions that run concurrently and is thus solely determined by the given action itself.

VBS can be seen as a more general form of constant-bandwidth servers (CBSs) [3]. CBS maintains a fixed rate of process execution, i.e., a CBS process is allowed to execute a constant amount of time (limit) in a time period of constant length (period) whereas a VBS process can change both the limit and the period at runtime. The only restriction on a VBS process is that its utilization, i.e., the ratio of its limit over its period, has to be less than or equal to a given utilization cap. This restriction enables a fast, sufficient schedulability test for VBS, which guarantees temporal isolation as long as the sum of the utilization caps of all processes is less than or equal to 100 %. Dynamically modifying the limit and period (rate) at which VBS processes execute enables different portions of process code to have different throughput

* This work has been supported by the EU ArtistDesign Network of Excellence on Embedded Systems Design, the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund S11404-N23), and an Elise Richter Fellowship (Austrian Science Fund V00125). We thank the anonymous reviewers for their comments and suggestions.

Correspondence to: ck@cs.uni-salzburg.at

(limit) and latency (period) requirements, which may be helpful in applications such as control loops [4].

In this article, we provide a detailed theoretical and practical view on VBS. After reiterating the basic concepts of VBS presented in [1], we extend the discussion twofold with material that has appeared in a technical report version [5] of [1]. In particular, we give detailed proofs of the schedulability test and include experiments analyzing the actual quality of temporal isolation which were not present in [1]. Next, we include implementation details that enable an efficient implementation of VBS in a real system. We present details of four alternative queue management plugins based on lists, arrays, matrices, and trees that trade off time and space complexity of the VBS scheduler. Finally, we present condensed versions of our previous results on scheduler overhead accounting with VBS [6] and on reducing power consumption while maintaining temporal isolation [7].

The structure of the rest of the article is as follows. We start by describing VBS conceptually in Sect. 2 and present the scheduling algorithm in Sect. 3, as introduced in [1], extended by the proofs of the results. In Sect. 4, we present implementation details of the scheduling algorithm including time and space complexity when using the four different queue management plugins. An extended version of the experimental results presented in [1] is discussed in Sect. 5. In Sects. 6 and 7 we round up the description by including the results on scheduler overhead accounting and power-aware scheduling with VBS. We present a detailed account of related work in Sect. 8. Secttion 9 gathers the conclusions and presents future work.

2 Variable-Bandwidth Servers

The timeline is represented by the set of natural numbers $\mathbb{N}$, i.e., we consider a discrete timeline. The main ingredients of the scheduling model are VBSs defined through virtual periodic resources and VBS-processes composed of sequential actions.

2.1 VBS and Processes

A virtual periodic resource [8,9] (capacity) is a pair $R = (\lambda, \pi)$ where $\lambda \in \mathbb{N}^+$ stands for limit and $\pi \in \mathbb{N}^+$ for period. If no confusion arises, we will say resource for virtual periodic resource. The limit λ specifies the maximum amount of time the resource R can be used (by a server and thus process) within the period π. We assume that in a resource $R = (\lambda, \pi)$, $\lambda \leq \pi$. The ratio

$$u = \frac{\lambda}{\pi}$$

is the utilization of the resource $R = (\lambda, \pi)$. We allow for an arbitrary set of resources denoted by $\mathcal{R}$.

A constant-bandwidth server (CBS) [3] is uniquely determined by a virtual periodic resource $R = (\lambda, \pi)$. A CBS serves CBS-processes at the virtual periodic resource R, that is, it lets a process execute for λ amount of time, within each period of length π. Hence, the process as a whole receives the constant bandwidth of the server, prescribed by the defining resource.

A variable-bandwidth server (VBS) is uniquely determined by the utilization ratio u of some virtual periodic resource. The utilization ratio prescribes an upper bound bandwidth cap. The server may execute processes that change the resources in time, as long as the resources have utilization less than or equal to the defining utilization. The notion of a process that can be served by a given VBS is, therefore, richer in structure. Note that a VBS can serve processes with any kind of activation. The server itself is periodic (with variable periodicity) but the processes need not be.

A VBS-process $P(u)$ served by a VBS with utilization u, is a finite or infinite sequence of (process) actions,

$$P(u) = \alpha_0 \alpha_1 \alpha_2 \dots$$

for $\alpha_i \in \mathrm{Act}$, where $\mathrm{Act} = \mathbb{N} \times \mathcal{R}$. An action $\alpha \in \mathrm{Act}$ is a pair $\alpha = (l, R)$ where l standing for load is a natural number, which denotes the exact amount of time the process will perform the action on the resource R, and $R = (\lambda, \pi)$ has utilization less than or equal to the utilization of the VBS, that is $\frac{\lambda}{\pi} \leq u$. If no confusion arises, we call VBS-processes simply processes, and we may also just write P instead of $P(u)$. By $\mathcal{P}$ we denote a finite set of processes under consideration.

Note that any action of a VBS-process is itself a finite CBS-process, hence a VBS-process can be seen as a sequential composition of CBS-processes. Moreover, note that the notion of load simplifies the model definition, although in the implementation it is in general not known a-priori.

Conceptually, the idea of decomposing a process into subtasks that run sequentially (the counterpart to actions in the VBS model) has appeared before, in the context of fixed-priority scheduling [10], and was extended in [4] for solving control-related issues.

As an illustration, let us now consider a simple theoretical example of processes and actions where the limit and period are expressed in seconds (which is the granularity of the time line).

Given a set $\mathcal{R} = \{(1,2), (1,4), (1,3)\}$ of resources, we consider a finite process $P(0.5)$ that first does some computation for 3 seconds with a virtual periodic resource $(1,2)$, then it works on allocating/deallocating memory objects of size 200 KB, which takes 2 seconds with the resource $(1,4)$, then it produces output of size 100 KB on an I/O device in 1 second with $(1,3)$, then again it computes, now for 2 seconds, with $(1,2)$ again. We can represent P as a finite sequence

$$\begin{aligned} P(0.5) &= \alpha_0 \alpha_1 \alpha_2 \alpha_3 \\ &= (3,(1,2))(2,(1,4))(1,(1,3))(2,(1,2)) \end{aligned}$$

on the $1s$-timeline. This process corresponds to (can be served by) a VBS with utilization $u = 0.5$ (or more).

2.2 Scheduling

A schedule for a finite set of processes $\mathcal{P}$ is a partial function $\sigma : \mathbb{N} \hookrightarrow \mathcal{P}$ from the time domain to the set of processes that assigns to each moment in time a process that is running in the time interval $[t, t+1)$. Here, $\sigma(t)$ is undefined if no process runs in $[t, t+1)$. Due to the sequential nature of the processes, any scheduler σ uniquely determines a function $\sigma_{\mathcal{R}} : \mathbb{N} \hookrightarrow \mathcal{P} \times \mathcal{R}$ which specifies the resource a process uses while being scheduled.

A schedule respects the resource capacity if for any process $P \in \mathcal{P}$ and any resource $R \in \mathcal{R}$, with $R = (\lambda, \pi)$ we have that for any natural number $k \in \mathbb{N}$

$$|\{t \in [k\pi, (k+1)\pi) \mid \sigma_{\mathcal{R}}(t) = (P, R)\}| \leq \lambda.$$

Hence, if the schedule respects the resource capacity, then the process P uses the resource R at most λ units of time per period of time π, as specified by its capacity.

Given a schedule σ for a set of processes $\mathcal{P}$, for each process $P \in \mathcal{P}$ and each action $\alpha_i = (l_i, R_i)$ that appears in P we distinguish four absolute moments in time:

- Arrival time a_i of the action α_i is the time instant at which the action arrives. We assume that a_i equals the time instant at which the previous action of the same process has finished. The first action of a process has arrival time zero.
- Completion time c_i of the action α_i is the time at which the action completes its execution. It is calculated as

$$c_i = \min\{c \in \mathbb{N} \mid l_i = |\{t \in [a_i, c) \mid \sigma(t) = P\}|\}.$$

- Finishing or termination time f_i of the action α_i is the time at which the action terminates or finishes its execution. We always have $f_i \geq c_i$. The difference between completion and termination is specified by the termination strategy of the scheduler. The process P can only invoke its next action if the previous one has been terminated. In the scheduling algorithm, we adopt the following termination strategy: an action is terminated at the end of the period within which it has completed. Adopting this termination strategy is needed for the correctness of the scheduling algorithm and the validity of the admission (schedulability) test.
- Release time r_i is the earliest time when the action α_i can be scheduled, $r_i \geq a_i$. If not specified otherwise, by the release strategy of the scheduler, we take $r_i = a_i$. In the scheduling algorithm, we will consider two release strategies, which we call early and late strategy.

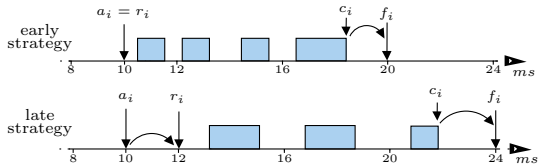


Fig. 1. Scheduling an action $\alpha = (5, (2, 4))$ [1]

We now present the two release strategies and elaborate the scheduling method via an example before continuing with the schedulability analysis.

In the late strategy, the release time of an action is delayed until the next period instance (of its resource) after the arrival time of the action. In the early strategy, the release time is equal to the arrival time, however, the limit of the action for the current period is adjusted so that it does not exceed its utilization in the remaining part of the current period. Our late strategy corresponds to the polling server [11] from classical scheduling theory, and the early strategy is similar in goal to the deferrable server [12]: it improves average response times since actions that arrive between period instances are not delayed.

Figure 1 presents the scheduling of an action $\alpha = (5, (2, 4))$ with load of 5 miliseconds, arriving at time 10, in both strategies. The resource used by the action has a period of 4 and a limit of 2 miliseconds. In the late strategy, the action is only released at time 12, which is the next period instance after the actual arrival time. Then, it takes three more periods for the action to finish. In the early strategy, the action is released at once, but in the remaining time of the current period (2 miliseconds) the limit is adjusted to 1, so that the utilization remains 0.5. In this situation, the scheduled response time in the early release strategy is one period shorter than in the late release strategy. In both cases, the action splits into a sequence of three tasks that are released in the consecutive periods. In the early strategy, these tasks are released at time 10, 12, and 16; have deadlines 12, 16, and 20; and durations 1, 2, and 2, respectively. In the late strategy, the tasks are released at times 12, 16, and 20; have deadlines 16, 20, and 24; and durations of 2, 2, and 1, respectively.

Using these notions, we define response time under the scheduler σ of the action α denoted by s_i, as the difference between the finishing time and the arrival time, i.e., $s_i = f_i - a_i$. Note that this definition of response time is logical in the sense that all possible side effects of the action should take effect at termination but not before. In the traditional (non-logical) definition, response time is the time from arrival to completion, decreasing response time (increasing performance) at the expense of increased jitter (decreased predictability).

Assume that the upper and lower response bounds b_i^u and b_i^l are given for each action α_i of each process P in a set of processes $\mathcal{P}$. The set $\mathcal{P}$ is schedulable with

respect to the given bounds if and only if there exists a schedule $\sigma : \mathbb{N} \hookrightarrow \mathcal{P}$ that respects the resource capacity and for which the actual response times do not exceed the upper response bounds, i.e., $s_i \leq b_i^u$, and are greater than the lower response bounds, i.e., $s_i \geq b_i^l$, for all involved actions α_i.

2.3 Schedulability Result

Given a finite set $\mathcal{P} = \{P_i(u_i) \mid 1 \leq i \leq n\}$ of processes with corresponding actions $\alpha_{i,j} = (l_{i,j}, R_{i,j})$ for $j \geq 0$, such that $P_i(u_i) = \alpha_{i,0}\alpha_{i,1}\ldots$ corresponds to a VBS with utilization u_i, we define the upper response time bound as

$$b_{i,j}^u = \pi_{i,j} - 1 + \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j}. \tag{1}$$

where $R_{i,j} = (\lambda_{i,j}, \pi_{i,j})$ with $l_{i,j}, R_{i,j}, \lambda_{i,j}$, and $\pi_{i,j}$ being as before the load, the resource, the limit, and the period for the action $\alpha_{i,j}$, respectively. Since an action $\alpha_{i,j}$ executes at most $\lambda_{i,j}$ of its load $l_{i,j}$ per period of time $\pi_{i,j}$, $\left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil$ is the number of periods the action needs in order to complete its load. In addition, in the response bound, we account for the time in the period in which the action arrives, which in the worst case is $\pi_{i,j} - 1$ if it arrives right after a period instance.

The lower response-time bound varies depending on the strategy used, namely

$$b_{i,j}^l = \begin{cases} \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j} \text{ , for late release} \\ \\ \left\lfloor \frac{l_{i,j}}{\lambda_{i,j}} \right\rfloor \pi_{i,j} \text{ , for early release.} \end{cases} \tag{2}$$

Note that the lower bound in the early release strategy is achieved only if $\lambda_{i,j}$ divides $l_{i,j}$, in which case $\left\lfloor \frac{l_{i,j}}{\lambda_{i,j}} \right\rfloor = \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil$. From these bounds, we can derive that the response-time jitter, i.e., the difference between the upper and lower bound on the response time, is at most $\pi_{i,j} - 1$ for the late release strategy and at most $2\pi_{i,j} - 1$ for the early release strategy.

The next schedulability/admission result justifies the definition of the response bounds and shows the correctness of our scheduling algorithm.

Proposition 1. *Given a set of processes $\mathcal{P} = \{P_i(u_i) \mid 1 \leq i \leq n\}$, as above, if*

$$\sum_{i=1}^{n} u_i \leq 1, \tag{3}$$

then the set of processes $\mathcal{P}$ is schedulable with respect to the resource capacity and the response bounds (1) and (2).

The proposition shows that it is enough to test whether the sum of the utilization (bandwidth) caps of all processes is less than one. The test is finite even though the processes may be infinite. In addition, the test is computable even if the actual loads of the actions are unknown, as it is often the case in practice. Hence, the standard utilization-based test for CBS-processes, holds also for VBS-processes. The test runs in constant time, meaning that whenever a new VBS-process enters the system, it is decidable in constant time whether it can be admitted and scheduled.

In order to prove Proposition 1, we first isolate a more essential schedulability property in the following section. The proof of Proposition 1 follows in Sects. 2.5 and 2.6.

2.4 Typed EDF

We describe a schedulability test for a particular dynamic EDF [2] scheduling algorithm, and prove its sufficiency.

Let $\tau = (r, e, d)$ be an aperiodic task with release time r, execution duration e, and deadline d, all natural numbers. We say that τ has type, or specification, (λ, π) where λ and π are natural numbers, $\lambda \leq \pi$, if the following conditions hold:

- $d = (n+1)\pi$ for the (uniquely determined) natural number n such that $r \in [n\pi, (n+1)\pi)$, and
- $e \leq (d - r)\frac{\lambda}{\pi}$.

Hence, a task specification is basically a periodic task which we use to impose a bound on aperiodic tasks. Note that if $r = n\pi$, then the duration e is limited to λ. A task of type (λ, π) need not be released at an instance of the period π, but its utilization factor in the interval of time $[r, d]$ remains at most $\frac{\lambda}{\pi}$.

Let S be a finite set of task types. Let I be a finite index set, and consider a set of tasks

$$\{\tau_{i,j} = (r_{i,j}, e_{i,j}, d_{i,j}) \mid i \in I, j \geq 0\}$$

with the properties:

- Each $\tau_{i,j}$ has a type in S. We will write $(\lambda_{i,j}, \pi_{i,j})$ for the type of $\tau_{i,j}$.
- The tasks with the same first index are released in a sequence, i.e., $r_{i,j+1} \geq d_{i,j}$ and $r_{i,0} = 0$.

The following result provides us with a sufficient schedulability test for such specific set of tasks.

Lemma 1. *Let $\{\tau_{i,j} \mid i \in I, j \geq 0\}$ be a set of tasks as defined above. If*

$$U = \sum_{i \in I} \max_{j \geq 0} \frac{\lambda_{i,j}}{\pi_{i,j}} \leq 1, \tag{4}$$

then this set of tasks is schedulable using the EDF strategy at any point of time, so that each task meets its deadline.

Proof. The proof builds upon the standard proof of sufficiency of the utilization test for periodic EDF, see, e.g., [13]. Assume the opposite, i.e., a deadline gets missed at time d by a task $\tau = (r, e, d) \in \{\tau_{i,j} \mid i \in I, j \geq 0\}$. Let t be the earliest moment in time such that in the interval $[t, d]$ there is full utilization and all executed tasks have deadlines that are not larger than d. Note that $t < d$ and t is a release time of some task.

Let $C(x, y)$ denote the computation demand of our set of tasks in an interval of time $[x, y]$. We have that $C(x, y)$ is the sum of the durations of all tasks with release time greater than or equal to x and deadline less than or equal to y.

Since a deadline is missed at d, we have

$$C(t, d) > d - t$$

i.e., the demand is larger than the available time in the interval $[t, d]$. We are going to show that $C(t, d) \leq (d - t)U$, which shows that $U > 1$ and completes the proof.

First we note that

$$C(t, d) = \sum_{i \in I} C_i(t, d)$$

where $C_i(t, d)$ is the computational demand imposed by tasks in $\{\tau_{i,j} \mid j \geq 0\}$ for a fixed $i \in I$.

In the finite interval $[t, d]$ only finitely many tasks in $\{\tau_{i,j} \mid j \geq 0\}$ are executed, say n tasks. Moreover, by the choice of t, none of these tasks is released at time earlier than t. Therefore, we can divide $[t, d]$ to subintervals

$$[t, d] = \bigcup_{k=0}^{n} [t_k, t_{k+1}]$$

where $t = t_0$ and $d = t_{n+1}$, and for all $k \in \{1, \ldots, n\}$, t_k is a release time of a task $\tau_k = (r_k, e_k, d_k)$ in $\{\tau_{i,j} \mid j \geq 0\}$. Since the tasks in $\{\tau_{i,j} \mid j \geq 0\}$ are released and executed in a sequence, we have that $d_k \leq t_{k+1}$. Let (λ_k, π_k) denote the type of τ_k. Moreover, we either have $t_1 = t_0$, or no task at all in $[t_0, t_1]$.

Denote by (λ_i^*, π_i^*) the "most expensive" task type in terms of utilization for $\{\tau_{i,j} \mid j \geq 0\}$, i.e.

$$\frac{\lambda_i^*}{\pi_i^*} = \max_{j \geq 0} \frac{\lambda_{i,j}}{\pi_{i,j}}.$$

We have $C_i(t_0, t_1) = 0$ and for $k > 0$,

$$\begin{aligned} C_i(t_k, t_{k+1}) &\leq (d_k - r_k)\tfrac{\lambda_k}{\pi_k} \\ &\leq (t_{k+1} - t_k)\tfrac{\lambda_k}{\pi_k} \\ &\leq (t_{k+1} - t_k) \cdot \tfrac{\lambda_i^*}{\pi_i^*}. \end{aligned}$$

Hence for all $k \in \{0, \ldots, n\}$, it holds that

$$C_i(t_k, t_{k+1}) \leq (t_{k+1} - t_k) \cdot \frac{\lambda_i^*}{\pi_i^*}.$$

Therefore,

$$\begin{aligned} C(t, d) &= \textstyle\sum_{i \in I} C_i(t_0, t_n) \\ &= \textstyle\sum_{i \in I} \sum_{k=0}^{n} C_i(t_k, t_{k+1}) \\ &\leq \textstyle\sum_{i \in I} \sum_{k=0}^{n} (t_{k+1} - t_k)\frac{\lambda_i^*}{\pi_i^*} \\ &= \textstyle\sum_{i \in I} (t_{n+1} - t_0)\frac{\lambda_i^*}{\pi_i^*} \\ &= (d - t)U. \end{aligned}$$

which completes the proof.

The schedulability test (4) computes the maximal utilization from the tasks in $\{\tau_{i,j} \mid j \geq 0\}$, given by the "most expensive" type. Since there are finitely many types, even though the set $\{\tau_{i,j} \mid j \geq 0\}$ may be infinite, the test is computable. Clearly, the test is conservative. For finite or "periodic" sets $\{\tau_{i,j} \mid j \geq 0\}$ one could come up with a more complex sufficient and necessary utilization test based on the overlap of the tasks. We leave such an investigation for future work.

We now present the proof of Proposition 1 first for the upper and then for the lower response-time bounds. [1]

2.5 Upper response-time bound

Each process P_i for $i \in I$ provides a sequence of tasks $\tau_{i,k}$ by refining each action to a corresponding sequence of tasks. Consider the action $\alpha_{i,j} = (l_{i,j}, R_{i,j})$ with capacity of $R_{i,j}$ given by $(\lambda_{i,j}, \pi_{i,j})$. Let n_j be a natural number such that

$$a_{i,j} \in ((n_j - 1)\pi_{i,j}, n_j \pi_{i,j}]$$

if $j > 0$, and let $n_0 = 0$. We distinguish two cases, one for each release strategy.

Case 1: Late release strategy Let

$$k_j = \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil.$$

The action $\alpha_{i,j}$ produces tasks $\tau_{i,k}$ for $k_0 + \cdots + k_{j-1} \leq k \leq k_0 + \cdots + k_{j-1} + k_j - 1$ given by:

$$\tau_{i,k} = ((n_j + m)\pi_{i,j}, e_{i,k}, (n_j + m + 1)\pi_{i,j})$$

where $m = k - (k_0 + \cdots + k_{j-1})$ and $e_{i,k} = \lambda_{i,j}$ if $k < k_0 + \cdots + k_j - 1$ or if $k = k_0 + \cdots + k_j - 1$ and $\lambda_{i,j}$ divides $l_{i,j}$, otherwise $e_{i,k} = l_{i,j} - \left\lfloor \frac{l_{i,j}}{\lambda_{i,j}} \right\rfloor \cdot \lambda_{i,j}$.

Hence, the workload of the action $\alpha_{i,j}$ is split into several tasks that all have type $(\lambda_{i,j}, \pi_{i,j})$. Moreover, the tasks in $\{\tau_{i,k} \mid k \geq 0\}$ are released in a sequence, such that (because of the termination strategy and the resource capacity) the release time of the next task is always equal or grater than the deadline of a given task. Therefore, Lemma 1 is applicable, and from the utilization test we get that the set of tasks $\{\tau_{i,k} \mid i \in I, k \geq 0\}$

[1] The proof for the lower response-time bound has also appeared in [6].

is schedulable so that all tasks meet their deadlines. Let σ be a schedule for this set of tasks. It corresponds to a schedule $\hat{\sigma}$ for the set of processes $\mathcal{P}$ by: $\hat{\sigma}(t) = P_i$ if and only if $\sigma(t) = \tau_{i,k}$ for some $k \geq 0$.

By construction, $\hat{\sigma}$ respects the resource capacity: Consider $P_i \in \mathcal{P}$ and $R \in \mathcal{R}$ with capacity (λ, π). In any interval of time $[n\pi, (n+1)\pi)$ there is at most one task $\tau_{i,k}$ of type (λ, π) produced by an action with resource R which is available and running, and its duration is limited by λ.

For the bounds, for each action $\alpha_{i,j}$, according to the termination strategy and the late release, we have

$$f_{i,j} = r_{i,j} + \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j}$$

where the release times are given by $r_{i,j} = n_j \pi_{i,j}$. The arrival times are $a_{i,j} = f_{i,j-1} \in ((n_j - 1)\pi_{i,j}, n_j \pi_{i,j}]$. Therefore

$$\begin{aligned} s_{i,j} &= f_{i,j} - a_{i,j} \\ &= \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j} + r_{i,j} - a_{i,j} \\ &\leq \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j} + \pi_{i,j} - 1 \\ &= b^u_{i,j} \end{aligned}$$

which completes the proof in the case of the late strategy. Hence, if the release time of each action is delayed to the next period instance, then we safely meet the response bounds. However, such a delay is not necessary. We may keep the release time equal to the arrival time and still meet the bounds. On average, we may achieve better response times than indicated by the bounds and also higher utilization.

Case 2: Early release strategy If the action $\alpha_{i,j}$ arrives on a period instance $\alpha_{i,j} = n_j \pi_{i,j}$ then there is nothing we can do better than in the late strategy. If not, then let

$$e_{i,j} = \min \left\{ l_{i,j}, \left\lfloor (n_j \pi_{i,j} - a_{i,j}) \frac{\lambda_{i,j}}{\pi_{i,j}} \right\rfloor \right\}$$

and

$$k_j = \left\lceil \frac{l_{i,j} - e_{i,j}}{\lambda_{i,j}} \right\rceil + 1.$$

Then $\alpha_{i,j}$ produces k_j tasks $\tau_{i,k}$ for $k_0 + \cdots + k_{j-1} \leq k \leq k_0 + \cdots + k_{j-1} + k_j - 1$ given by: for $k = k_0 + \cdots + k_{j-1}$

$$\tau_{i,k} = (a_{i,j}, e_{i,j}, n_j \pi_{i,j}),$$

and if $k_j > 1$, then for $k_0 + \cdots + k_{j-1} < k < k_0 + \cdots + k_{j-1} + k_j - 1$

$$\tau_{i,k} = ((n_j + m)\pi_{i,j}, \lambda_{i,j}, (n_j + m + 1)\pi_{i,j}),$$

where $m = k - (k_0 + \cdots + k_{j-1} + 1)$. Now if $\lambda_{i,j}$ divides $l_{i,j} - e_{i,j}$, then also for $k = k_0 + \cdots + k_{j-1} + k_j - 1$ we have

$$\tau_{i,k} = ((n_j + m)\pi_{i,j}, \lambda_{i,j}, (n_j + m + 1)\pi_{i,j}),$$

with $m = k - (k_0 + \cdots + k_{j-1} + 1)$. If, on the other hand, $\lambda_{i,j}$ does not divide $l_{i,j} - e_{i,j}$, then for $k = k_0 + \cdots + k_{j-1} + k_j - 1$ we have

$$\begin{aligned} \tau_{i,k} = ((n_j + m)\pi_{i,j}, l_{i,j} - e_{i,j} \\ - \left\lfloor \frac{l_{i,j} - e_{i,j}}{\lambda_{i,j}} \right\rfloor \cdot \lambda_{i,j}, (n_j + m + 1)\pi_{i,j}), \end{aligned}$$

where again $m = k - (k_0 + \cdots + k_{j-1} + 1)$.

Hence, we let a task of $\alpha_{i,j}$ start as soon as $\alpha_{i,j}$ has arrived, taking care not to exceed the limit in the current period as well as to keep the utilization below the specified bound (via the duration of the task $e_{i,j}$). The rest of the action is divided in tasks as before. Note that all tasks produced by $\alpha_{i,j}$ are still of type $(\lambda_{i,j}, \pi_{i,j})$. Also in this case the termination strategy makes sure that each release time is larger than or equal to the deadline of the previous task. Hence, the set of tasks is schedulable via Lemma 1, and the induced process schedule respects the resource capacity. For the bounds, we now have

$$f_{i,j} \leq r_{i,j} + \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j} + \pi_{i,j} - 1$$

and $a_{i,j} = r_{i,j}$. Hence,

$$\begin{aligned} s_{i,j} &= f_{i,j} - a_{i,j} \\ &\leq \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j} + \pi_{i,j} - 1 \\ &= b^u_{i,j} \end{aligned}$$

which completes the proof.

2.6 Lower response-time bound

For each action $\alpha_{i,j}$, according to the termination strategy and the late release strategy, we have

$$f_{i,j} = r_{i,j} + \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j} \tag{5}$$

where the release times are given by $r_{i,j} = n_j \pi_{i,j}$ for some natural number n_j such that the arrival times are $a_{i,j} = f_{i,j-1} \in ((n_j - 1)\pi_{i,j}, n_j \pi_{i,j}]$. Therefore, for the late strategy we have

$$\begin{aligned} s_{i,j} = f_{i,j} - a_{i,j} \overset{(5)}{=} \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j} + r_{i,j} - a_{i,j} \\ \geq \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j} = b^l_{i,j}, \text{ for late release.} \end{aligned}$$

For the early release strategy, we distinguish two cases depending on whether the following inequality holds

$$\left\lfloor m \frac{\lambda_{i,j}}{\pi_{i,j}} \right\rfloor \geq l_{i,j} - \left\lfloor \frac{l_{i,j}}{\lambda_{i,j}} \right\rfloor \lambda_{i,j} \tag{6}$$

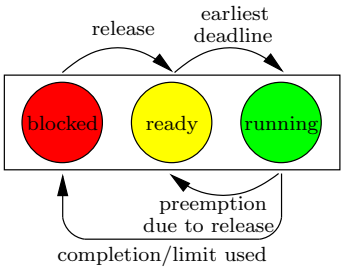


Fig. 2. Process states [1]

where $m = n_j\pi_{i,j} - r_{i,j}$ and $r_{i,j} = a_{i,j} = f_{i,j-1} \in ((n_j-1)\pi_{i,j}, n_j\pi_{i,j}]$. The finishing time for the action $\alpha_{i,j}$ is

$$f_{i,j} = \begin{cases} a_{i,j} + \left\lfloor \frac{l_{i,j}}{\lambda_{i,j}} \right\rfloor \pi_{i,j} + m \text{ , if (6) holds} \\ a_{i,j} + \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j} + m \text{ , otherwise} \end{cases} \quad (7)$$

In both cases $f_{i,j} \geq \left\lfloor \frac{l_{i,j}}{\lambda_{i,j}} \right\rfloor \pi_{i,j} + a_{i,j}$, so

$$\begin{aligned} s_{i,j} &= f_{i,j} - a_{i,j} \\ &\geq \left\lfloor \frac{l_{i,j}}{\lambda_{i,j}} \right\rfloor \pi_{i,j} = b^l_{i,j}, \text{for early release.} \end{aligned}$$

3 Scheduling Algorithm

In this section, we describe the scheduling algorithm which follows the proof of Proposition 1. At any relevant time t, our system state is determined by the state of each process. A process may be blocked, ready, or running as depicted in Fig. 2. By Blocked, Ready, and Running we denote the current sets of blocked, ready, and running processes, respectively. These sets are ordered: Blocked is ordered by the release times, Ready is ordered by deadlines, and Running is either empty (for an idle system) or contains the currently running process of the system. Thus,

$$\mathcal{P} = \text{Blocked} \cup \text{Ready} \cup \text{Running}$$

and the sets are pairwise disjoint. Additionally, each process is represented by a tuple in which we keep track of the process evolution. For the process P_i we have a tuple

$$(i, j, d_i, r_i, l^c_i, \lambda^c_i)$$

where i is the process identifier, j stores the identifier of its current action $\alpha_{i,j}$, d_i is the current deadline (which is not the deadline for the entire action, but rather an instance of the action period $\pi_{i,j}$), r_i is the next release time, l^c_i is the current load, and λ^c_i is the current limit. The scheduler also uses a global time value t_s which stores the previous time instant at which the scheduler was invoked.

Given n processes $P_1, \ldots, P_n$, as defined in the previous section, initially we have

$$\text{Blocked} = \{P_1, \ldots, P_n\}, \text{ Ready} = \text{Running} = \emptyset.$$

At specific moments in time, including the initial time instant, we perform the following steps:

1. Update process state for the process in Running.
2. Move processes from Blocked to Ready.
3. Update the set Running.

We discuss each step in more detail below.

1. If Running $= \emptyset$, i.e., the system was idle, we skip this step. Otherwise, let P_i be the process in Running at time t. We differentiate three reasons for which P_i is preempted at time t: completion, limit, and release.

Completion. P_i completes the entire work related to its current action $\alpha_{i,j} = (l_{i,j}, R_{i,j})$. If we have reached process termination, i.e., there is no next action, we have a zombie process and remove it from the system. Otherwise, $j \leftarrow j+1$ and the current action becomes $\alpha_{i,j+1} = (l_{i,j+1}, R_{i,j+1})$ with the resource capacity $(\lambda_{i,j+1}, \pi_{i,j+1})$. The current load l^c_i becomes $l_{i,j+1}$.

If $R_{i,j+1} = R_{i,j}$, i.e., the resource of the next action remains the same, then there is no need to reschedule: P_i is moved to Ready, its deadline d_i, and release time r_i remain unchanged, and we subtract the work done from λ^c_i, $\lambda^c_i \leftarrow \lambda^c_i - (t - t_s)$.

If $R_{i,j+1} \neq R_{i,j}$, we have currently implemented two release strategies as described above. First we take care of the termination strategy. Let $m \in \mathbb{N}$ be a natural number such that $t \in ((m-1)\pi_{i,j}, m\pi_{i,j}]$. According to our termination strategy, the action $\alpha_{i,j}$ is terminated at time $m\pi_{i,j}$ which is the end of the period in which the action has completed. Now let $k \in \mathbb{N}$ be a natural number such that

$$m\pi_{i,j} \in ((k-1)\pi_{i,j+1}, k\pi_{i,j+1}].$$

The first strategy, the late release strategy, calculates r_i, the next release time of P_i, as the start of the next period of $R_{i,j+1}$ and its deadline as the start of the second next period,

$$r_i \leftarrow k\pi_{i,j+1},\ d_i \leftarrow (k+1)\pi_{i,j+1}.$$

The new current limit becomes $\lambda_{i,j+1}$ and P_i is moved to Blocked.

The second strategy, the early release strategy, sets the release time to the termination time and the deadline to the end of the release-time period

$$r_i \leftarrow m\pi_{i,j},\ d_i \leftarrow k\pi_{i,j+1}$$

and calculates the new current limit for P_i, as

$$\lambda^c_i \leftarrow \left\lfloor (d_i - r_i)\frac{\lambda_{i,j+1}}{\pi_{i,j+1}} \right\rfloor.$$

The process P_i is moved to Blocked.

Limit. P_i uses all of the current limit λ_i^c for the resource $R_{i,j}$. In this case, we update the current load, $l_i^c \leftarrow l_i^c - (t - t_s)$, and

$$\lambda_i^c \leftarrow \lambda_{i,j},\ r_i \leftarrow k\pi_{i,j},\ d_i \leftarrow (k+1)\pi_{i,j},$$

with $k \in \mathbb{N}$ such that $t \in ((k-1)\pi_{i,j}, k\pi_{i,j}]$. With these new values P_i is moved to Blocked.

Release. If a process is released at time t, i.e., P_m is a process, $P_m \neq P_i$, with the release time $r_m = t$, then the priorities have to be established anew. We update the current load and limit,

$$l_i^c \leftarrow l_i^c - (t - t_s),\ \lambda_i^c \leftarrow \lambda_i^c - (t - t_s).$$

The deadline for P_i is set to the end of the current period, $d_i \leftarrow k\pi_{i,j}$, with $k \in \mathbb{N}$ such that $t \in ((k-1)\pi_{i,j}, k\pi_{i,j}]$. P_i is then moved to Ready.

2. In the second step, the scheduler chooses the processes from Blocked which are to be released at the current time t, i.e., $\{P_i \mid r_i = t\}$, and moves them to the set Ready.

3. In the third step if the Ready set is empty, the scheduler leaves the Running set empty, thus the system becomes idle. Otherwise, the scheduler chooses a process P_i with the earliest deadline from Ready and moves it to Running.

We calculate:

- t_c : the time at which the new running process P_i would complete its entire work needed for its current action without preemption, i.e., $t_c = t + l_i^c$.
- t_l : the time at which P_i consumes its current limit for the current period of the resource R_i, i.e., $t_l = t + \lambda_i^c$.
- t_r : the next release time of any process in Blocked. If Blocked is empty, $t_r = \infty$.

The scheduler stores the value of the current time in t_s, $t_s \leftarrow t$, and the system lets P_i run until the time $t = \min(t_c, t_l, t_r)$ at which point control is given back to the scheduling algorithm.

As stated, the algorithm uses knowledge of the load of an action. However, in the implementation there is a way around it (by marking a change of action that forces a scheduler invocation) which makes the algorithm applicable to actions with unknown load as well, in which case no explicit response-time guarantees are given. The complexity of the scheduling algorithm amounts to the complexity of the plugins that manage the ordered Blocked and Ready sets, the rest of the algorithm has constant-time complexity.

4 Implementation

The scheduler uses a well-defined interface to manage the processes in the system. This interface is implemented by three alternative plugins, each with different attributes regarding time complexity and space overhead. Currently, our implementation, available via the Tiptoe homepage [14], supports doubly-linked lists (Sect. 4.1), time-slot arrays of FIFO queues (Sect. 4.2), a time-slot matrix of FIFO queues (Sect. 4.3), and a tree-based optimization of the matrix. The implementation details can also be found in a technical report version [5] of [1].

	list	array	matrix/tree
time	$O(n)$	$O(\log(t) + n\log(t))$	$\Theta(t)$
space	$\Theta(n)$	$\Theta(t+n)$	$O(t^2+n)$

Table 1. Time and space complexity per plugin

The array and matrix implementation impose a bound on the number of time instants. For this reason, we introduce a finite coarse-grained timeline with t time instants and a distance between any two instants equal to a fixed natural number d. Deadlines and release times are then always in the coarse-grained timeline, which restricts the number of different periods in the system. The scheduler may be invoked at any time instant of the original (fine-grained) timeline. However, the second step of the algorithm (releasing processes) is only executed during scheduler invocations at time instants of the coarse-grained timeline.

The matrix- and tree-based implementations are $O(1)$-schedulers since the period resolution is fixed. However, not surprisingly, temporal performance comes at the expense of space complexity, which grows quadratically in period resolution for both plugins. Space consumption by the tree plugin is significantly smaller than with the matrix plugin, if the period resolution is higher than the number of servers. The array-based implementation runs in linear time in the number of servers and requires linear space in period resolution. The list-based implementation also runs in linear time in the number of servers, but only requires space independent of period resolution (although insertion is more expensive than for the array plugin).

Table 1 shows the system's time and space complexities distinguished by plugin in terms of the number of processes in the system (n), and in the period resolution, that is, the number of time instants the system can distinguish (t). For efficiency, we use a time representation similar to the circular time representation of [15].

Table 2 summarizes the queue operations' time complexity in terms of the number of processes (n) and the number of time instants (t). The first operation is called ordered-insert by which processes are inserted according to a key, and processes with the same key are kept in FIFO order to maintain fairness. The select-first operation selects the first element in the respective queue. The release operation finds all processes with a certain key, reorders them according to a new key, and merges the result into another given queue. Note that t is actu-

	list	array	matrix
ordered-insert	$O(n)$	$\Theta(\log(t))$	$\Theta(\log(t))$
select-first	$\Theta(1)$	$O(\log(t))$	$O(\log(t))$
release	$O(n)$	$O(\log(t) + n \cdot \log(t))$	$\Theta(t)$

Table 2. Time complexity of the queue operations.

ally a constant, so the matrix implementation achieves constant time for all three operations.

We now describe each plugin in more detail.

4.1 Process List

The list plugin uses ordered doubly-linked lists for Ready, which is sorted by deadline, and Blocked, which is sorted first by release time and then by deadline. Therefore, inserting a single element has linear complexity with respect to the number of processes in the queue, while selecting the first element in the queue is done in constant time. Releasing processes in Blocked which contains k processes by moving them to Ready, which contains m processes, takes $k + m$ steps. The upper bound of k and m is n, and therefore the complexity is $O(n)$. Advantages of this data structure are low memory usage (only two pointers per process) and no limitation on the resolution of the timeline.

4.2 Time-Slot Array

The array plugin uses an array of pointers to represent the timeline. Each element in the array points to a FIFO queue of processes. A pointer at position t_i in Blocked, for instance, points to a FIFO queue of processes that are to be released at time t_i. In Ready, a pointer at position t_i is a reference to a FIFO queue of processes with a deadline t_i. Note that whenever we speak of time instants in the array or matrix plugins, we mean time instants modulo the size of the array or matrix, respectively.

In a naive implementation, inserting a process would be achieved in constant time using the key (release time or deadline) as index into the array. Finding the first process of this array would then be linear in the number of time instants (t). A more balanced version uses an additional bitmap to represent whether there are any processes at a certain time instant or not. The bitmap is split into words with an additional header bitmap that indicates which word of the bitmap has at least one bit set. Furthermore, if the header bitmap has more than s bits, where s denotes the word size,[2] it recursively maintains a header again. The bitmap implementation, therefore, can be seen as a tree with depth $\log_s(t)$ where the nodes are words and each node has s children. In this way the select-first operation improves from linear complexity to $\log_s(t)$, but the ordered-insert operation degrades from constant time to $\log_s(t)$ complexity, due to necessary updates in the bitmap.

During the release operation at time instant t_i, all k processes in the FIFO queue at position t_i in the Blocked array are inserted at the correct position in the Ready array. The complexity of this operation is $k \cdot \log_s(t)$. The bit which indicates that there are processes at time instant t_i in Blocked is cleared in $\log_s(t)$ steps. Thus, the time complexity of the release operation is at most $n \cdot \log_s(t) + \log_s(t)$, since n is the upper bound of k.

The disadvantage of this plugin is the static limit on the timeline, i.e., the number of time-slots in an array. This imposes a limitation of how far into the future a process can be released, and on the maximum deadline of a process. Therefore, the possible range of resource periods in the system is bounded with this plugin. Furthermore, the array introduces a constant memory overhead. Each array with t time-slots, contains t pointers and $(s/(s-1)) \cdot (t-1)$ bits for the bitmap. For example, with $t = 1024$ this results in 4KB for the pointers and 132 bytes for the bitmap.

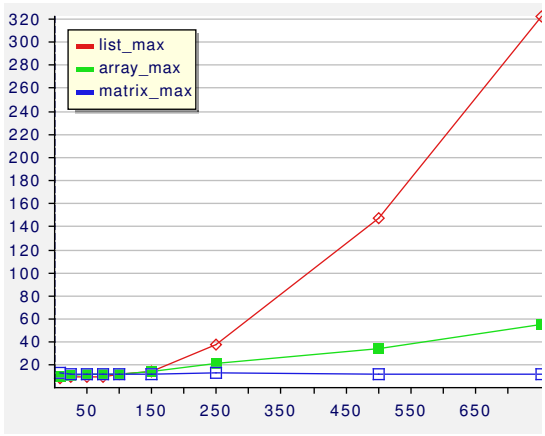
4.3 Time-Slot Matrix

In order to achieve constant execution time in the number of processes for all operations on the queues we have designed a matrix of FIFO queues, also referred to as FIFO matrix. The matrix contains all processes in the system, and the position in the matrix indicates the processes deadline (column entry) and the processes release time (row entry). The matrix implicitly contains both Ready and Blocked, which can be computed by providing the current time. In a naive implementation, select-first has complexity $O(t^2)$, whereas insert-ordered and release are constant time. To balance this, additional meta-data are introduced which reduces the complexity of select-first to $O(\log(t))$ and degrades the complexity of the other operations (cf. Table 2).

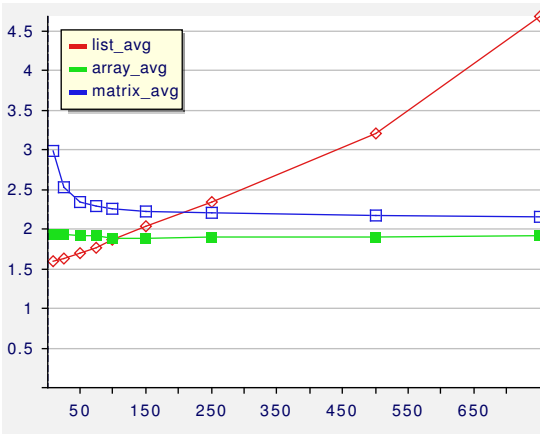
We introduce a two-dimensional matrix of bits, having value 1 at the positions at which there are processes in the original matrix. In addition, we use two more bitmaps, called release bitmap and ready bitmap. The release bitmap indicates at which row in the matrix of bits at least one bit is set. The ready bitmap contains the information in which columns released processes can be found. Note that the release bitmap merely reflects the content of the matrix. The ready bitmap provides additional information, it indicates where the currently released processes are located.

A process is put into the FIFO matrix in constant time. However, the corresponding updates of the bit matrix and the release bitmap take $\log_s(t)$ operations each. Therefore, inserting a process has a time complexity of $2 \cdot \log_s(t)$. Finding and returning the first process in Released or Blocked has also a complexity of $2 \cdot \log_s(t)$. To find the first process in Ready, e.g., we find the first set

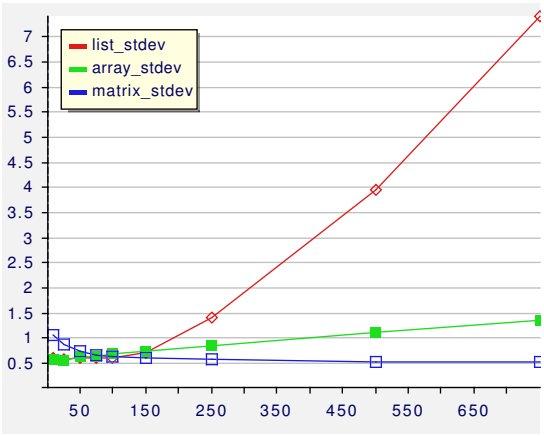
[2] Our implementation supports 32-bit and 64-bit word size, on corresponding CPU architectures. The measurements and example calculations were done for $s = 32$.



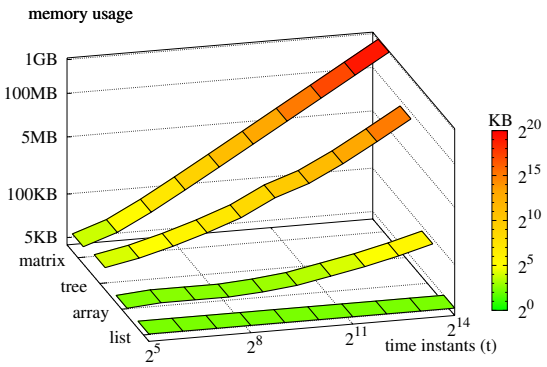
(a) Maximum



(b) Average



(c) Standard deviation



(d) Memory usage

Fig. 3. Scheduler time (a),(b),(c) [x-axis: number of processes, y-axis: execution time in microseconds] and space overhead (d) [1]

bit in the ready bitmap in $\log_s(t)$ operations. If the bit is at position i, then the ith column in the bit matrix is examined, in order to find the set bit j corresponding to the first process in Ready, also in at most $\log_s(t)$ operations. The two indexes, i and j, are then used to access the process in the FIFO matrix. As a result, the operation of selecting the first process has a total complexity of $2 \cdot \log_s(t)$.

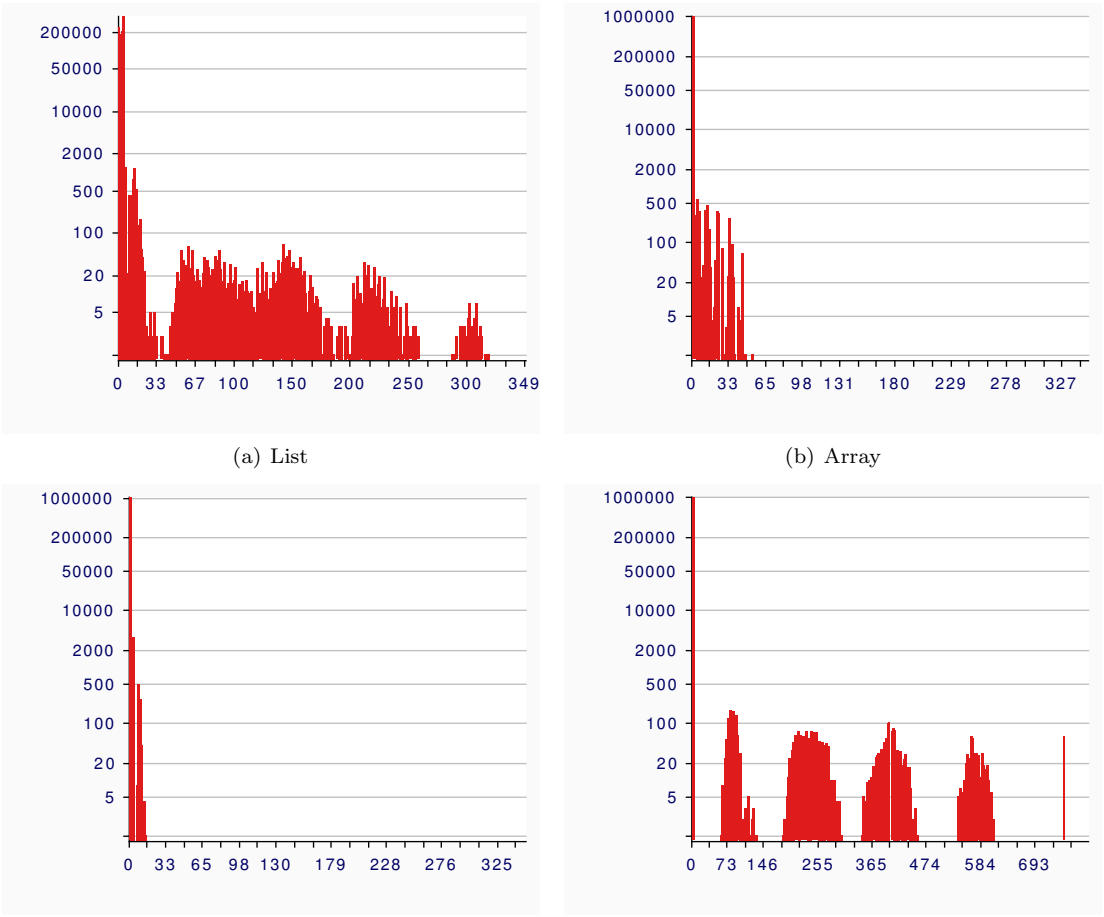
The release operation does not involve moving processes. Releasing processes is done by updating the ready bitmap. More precisely, the new ready bitmap for time instant t_i is the result of a logical OR between row t_i in the bit matrix and the old ready bitmap. The OR operation is word-wise and, therefore, linear in the size of the bitmap, which is linear in the number of time instants.

In addition to the static limitation for the number of time instants, a disadvantage of the matrix plugin is the high memory usage. To distinguish t time instants the FIFO matrix uses t^2 pointers. Additionally, the metadata consists of $(s/(s-1)) \cdot t \cdot (t-1)$ bits for the bit matrix and $(s/(s-1)) \cdot (t-1)$ bits for each bitmap. In order to fully exploit the available hardware instruction for searching and modifying bitmaps, the transpose of the bit matrix is also kept in memory, which adds additional $(s/(s-1)) \cdot t \cdot (t-1)$ bits.

As an alternative to the FIFO matrix representation, we also implemented the FIFO matrix as a B+ tree [16]. Using a B+ tree for the FIFO matrix adds $2 \cdot \log_s(t)$ operations to the complexity of the ordered-insert and select-first operations, because the depth of the tree is $2 \cdot \log_s(t)$. The memory usage of the B+ tree might in the worst case exceed the memory usage of the FIFO matrix. A worst-case scenario occurs when each position of the FIFO matrix contains at least one process. However, if the FIFO matrix is sparse, e.g., because the number of processes in the system is much smaller than the number of distinguishable time instants, then the memory overhead reduces drastically. See the next section for details.

5 Experiments and Results

We present results of different experiments with the scheduler implementation, running on a 2GHz AMD64 machine with 4GB of memory.



(a) List (b) Array

(c) Matrix (d) Process releases

Fig. 4. Execution time histograms (a),(b),(c) [x-axis: execution time in microseconds, y-axis: (log-scale) number of scheduler calls] and process release histogram (d) [x-axis: number of released processes, y-axis: (log-scale) number of scheduler calls] [1]

5.1 Scheduler Overhead

In order to measure scheduler execution times, we schedule 9 different sets of simulated processes with 10, 25, 50, 75, 100, 150, 250, 500, and 750 processes each, with the number of distinguishable time instants t in the scheduler fixed to $2^{14} = 16384$. Each process has a variable number of actions with random periods and loads. The limit of every action is 1. During these experiments, the execution time of every single scheduler invocation is measured using the software oscilloscope tool TuningFork [17]. From a sample of one million invocations, we calculate the maximum (Fig. 3(a)), the average (Fig. 3(b)), and the standard deviation (Fig. 3(c)) in execution times. The x-axis of each of the three figures represents the number of processes in the set and the y-axis the execution time in microseconds. The B+ tree plugin performs the same as the matrix plugin up to 140ns, and is therefore not shown. Note that the experimental results with the list plugin were obtained using an implementation described in [1,5] which had quadratic time complexity.

The execution time measurements conform to the complexity bounds from Sect. 4. For a low number of processes (less than 150), all plugins perform similarly and the scheduler needs at most 20 microseconds. On average (Fig. 3(b)), for a low number of processes (up to 100) the list plugin is the fastest. Interestingly, on average the array plugin is always faster than the matrix plugin, even for a high number of processes. The reason is that the constant overhead of the matrix operations is higher, which can be seen in the average but not in the maximal execution times.

The variability (jitter) of the scheduler execution can be expressed in terms of its standard deviation, depicted in Fig. 3(c). The variability of the list and array plugins increases similarly to their maximum execution times when more than 150 processes are scheduled. The matrix plugin, however, has a lower standard deviation for a high number of processes and a higher standard deviation for a low number of processes. This is related to the better average execution time (Fig. 3(b)) for higher number of processes, as a result of cache effects. By instrumenting the scheduler we discovered that bitmap func-

tions, e.g., setting a bit, are on average up to four times faster with 750 processes than with 10 processes, which suggests CPU cache effects.

The memory usage of all plugins, including the tree plugin, for 750 processes with an increasing number of distinguishable time instants is shown in Fig. 3(d). The memory usage of just the B+ tree is 370KB, compared to the 1GB for the matrix plugin. In both cases up to 66MB additional memory is used for meta-data, which dominates the memory usage of the tree plugin. The graphs in Fig. 3(d) are calculated from theoretical bounds. However, our experiments confirm the results.

Figures 4(a), 4(b), and 4(c) show the different behavior of the presented plugins when scheduling 750 processes. These figures are histograms of the scheduler execution time and are used to highlight the distribution of it. The x-axis represents the execution time in microseconds and the y-axis (log-scale) represents the number of scheduler calls. For example, in Fig. 4(a) there are about 50 scheduler calls that executed for 100 microseconds during the experiment.

The list plugin varies between 0 and 350 microseconds, the array plugin between 0 and 55 microseconds, and the matrix plugin does not need more than 20 microseconds for any scheduler execution. The execution time histograms, especially histogram 4(a), are closely related to the histogram of the number of processes released during the experiment (Fig. 4(d)). The x-axis represents the number of released processes and the y-axis (log-scale) represents how many times a certain number of processes is released. The similarity of Fig. 4(a) and Figure 4(d) indicates that the release of processes dominates the execution of the scheduler for the experiment with 750 processes.

5.2 Release Strategies

In this section we compare the two implemented release strategies of the scheduler in two experiments and show that the early strategy achieves optimal average response times (always better by one period than the late strategy) for a single process with increasingly non-harmonic periods (Fig. 5(a), top), and improves average response times for an increasing number of processes with a random distribution of loads, limits, and periods (Fig. 5(b), top). In both experiments, response times are in milliseconds, and limits and periods are chosen such that the theoretically possible CPU utilization (Proposition 1) is close to one. The early strategy achieves at least as high actual CPU utilization as the late strategy, and thus less CPU idle time (bottom part of both figures). For Fig. 5(a), the single process alternates between two actions that have their periods (and limits) equal to some natural number n and $n+1$, respectively, shown as pairs $(n, n+1)$ on the x-axis. Hence, the actions are increasingly non-harmonic and the corresponding periods are relatively prime. The process always invokes the actions on the lowest possible load to maximize switching between actions resulting in increasingly lower CPU utilization.

6 Scheduler Overhead Accounting

The response-time bounds (1) and (2) presented in Sect. 2.3 are the result of several assumptions on the process and system model. One important assumption that has been implicitly made in the previous sections and which is prevalent in literature is that the scheduler overhead is zero. However, in a real system the effect of the scheduler overhead on the response-time bounds of processes (or actions) can be substantial. We (a subset of the authors) have extended the VBS schedulability and response-time bound analysis to include scheduler overhead in [6]. Here, we present a condensed version of that analysis for completeness of the VBS result.

The first step towards including the scheduler overhead in the VBS analysis is to determine an upper bound on the number of scheduler invocations that can occur during a time interval. In particular, we want to determine the worst-case number of scheduler invocations that an action of a VBS process experiences during one period.

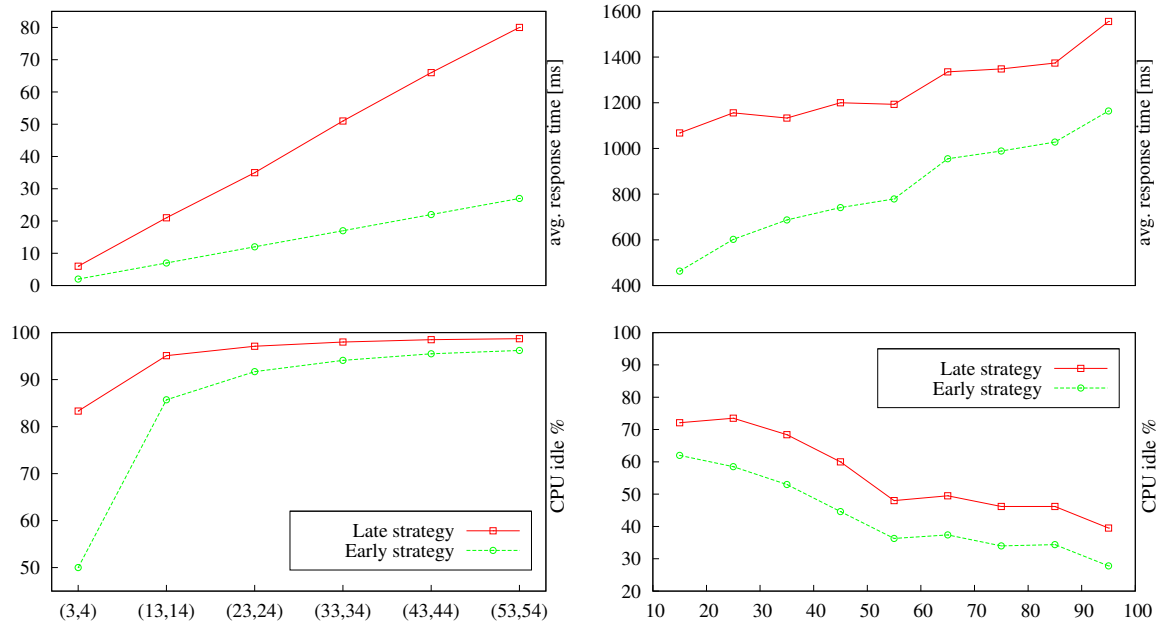
The duration of a scheduler invocation is typically several orders of magnitude lower than a unit execution of an action. Therefore, we make the assumption that all periods belong to the set of discrete time instants $M = \{c \cdot n \mid n \geq 0\} \subset \mathbb{N}$, for a constant value $c \in \mathbb{N}, c > 1$. Hence, for any action $\alpha_{i,j}$ with its associated virtual periodic resource $R_{i,j} = (\lambda_{i,j}, \pi_{i,j})$ we have that $\pi_{i,j} = c \cdot \pi'_{i,j}$ with $\pi'_{i,j} \in \mathbb{N}$. We call c the scale of the system. Intuitively, we can say that there are two different timelines, the "fine-grained timeline" given by the set of natural numbers and the "coarse-grained timeline" given by the set M. Resource periods are defined on the "coarse-grained timeline", while the execution time of the scheduler is defined on the "fine-grained timeline".

In VBS scheduling, a process P_i is preempted at a time instant t if and only if one of the following situations occurs:

1. Completion. P_i has completed the entire work related to its current action $\alpha_{i,j} = (l_{i,j}, R_{i,j})$.
2. Limit. P_i uses up all resource limit $\lambda_{i,j}$ of the current resource $R_{i,j}$.
3. Release. A task of an action is released at time t, i.e., an action of another process is activated. Note that all preemptions due to release occur at time instants on the "coarse-grained timeline", the set M.

The following result holds.

Lemma 2 ([6]). *Let $\mathcal{P} = \{P_i(u_i) \mid 1 \leq i \leq n\}$ be a set of VBS processes with actions $\alpha_{i,j}$ and corresponding virtual periodic resources $(\lambda_{i,j}, \pi_{i,j})$. There are at most*



(a) Increasing non-harmonicity (b) Increasing concurrency

Fig. 5. Release strategies comparisons [5]

$N_{i,j} = N^R_{i,j} + 1$ *scheduler invocations every period* $\pi_{i,j}$ *for the action* $\alpha_{i,j}$*, where*

$$N^R_{i,j} = \left\lceil \frac{\pi_{i,j}}{gcd(\{\pi_{m,n} \mid m \in I, n \geq 0, m \neq i\})} \right\rceil \quad (8)$$

for $I = \{i \mid 1 \leq i \leq n\}$.

If we denote the duration of one scheduler invocation by ξ, the total scheduler overhead for one period of an action $\alpha_{i,j}$ is $\delta_{i,j} = N_{i,j} \cdot \xi$. The total overhead is therefore made up of $N_{i,j}$ pieces of non-preemptable workload ξ. An important consequence of this demarcation is that the scheduler overhead only depends on the finitely many periods in the system and not on the load of the action.

We have determined two complementary methods to account for scheduler overhead, either by decreasing the speed at which processes run to maintain CPU utilization, or by increasing CPU utilization to maintain the speed at which processes run.

We call the first method response accounting and the second utilization accounting. Scheduler overhead accounting can therefore be done in two ways. One way is to allow an action to execute for less time than its actual limit within one period and use the remaining time to account for the scheduler overhead. The other way is to increase the limit so that the action can execute both its original limit and the time spent on scheduler invocations within one period.

We write that the overhead is

$$\delta_{i,j} = \delta^b_{i,j} + \delta^u_{i,j},$$

where $\delta^b_{i,j}$ is the overhead that extends the response-time bounds of the respective action and $\delta^u_{i,j}$ increases the utilization. Since ξ is not divisible, both $\delta^b_{i,j}$ and $\delta^u_{i,j}$ are multiples of ξ.

We differentiate three cases summarized in Table 3:

- Response accounting (RA), $\delta_{i,j} = \delta^b_{i,j}$, when the entire overhead is executing within the limit of the action, keeping both the limit and period (and thus the utilization) of the actions constant but increasing the response-time bounds.
- Utilization accounting (UA), $\delta_{i,j} = \delta^u_{i,j}$, when the entire overhead increases the limit of the action, and thus the utilization, but the response-time bounds remain the same.
- Combined accounting (RUA), with $\delta_{i,j} = \delta^b_{i,j} + \delta^u_{i,j}$, $\delta^b_{i,j} > 0$, and $\delta^u_{i,j} > 0$, which offers the possibility to trade-off utilization for response time, for each action, in the presence of scheduler overhead.

For an action $\alpha_{i,j}$, in the presence of overhead, we denote the new load by $l^*_{i,j}$, the new limit by $\lambda^*_{i,j}$, and the new utilization by $u^*_{i,j}$. Using these new parameters and the old response-time bounds, defined in Equation (1) and (2), we determine the new upper and lower response-time bounds which we denote with $b^{u*}_{i,j}$ and $b^{l*}_{i,j}$, respectively. The upper response-time bound $b^{u*}_{i,j}$ for action $\alpha_{i,j}$ is

$$b^{u*}_{i,j} = \left\lceil \frac{l^*_{i,j}}{\lambda^*_{i,j}} \right\rceil \pi_{i,j} + \pi_{i,j} - 1. \quad (9)$$

Case	Overhead distribution	Load	Utilization	Schedulability test
RA	$\delta^b_{i,j} = \delta_{i,j},\ \delta^u_{i,j} = 0$	$l^*_{i,j} = l_{i,j} + \left\lceil \frac{l_{i,j}}{\lambda_{i,j} - \delta_{i,j}} \right\rceil \delta_{i,j}$	$u^*_{i,j} = \frac{\lambda_{i,j}}{\pi_{i,j}}$	$\sum_{i \in I} \max_{j \geq 0} \frac{\lambda_{i,j}}{\pi_{i,j}} \leq 1$
UA	$\delta^b_{i,j} = 0,\ \delta^u_{i,j} = \delta_{i,j}$	$l^*_{i,j} = l_{i,j} + \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \delta_{i,j}$	$u^*_{i,j} = \frac{\lambda_{i,j} + \delta_{i,j}}{\pi_{i,j}}$	$\sum_{i \in I} \max_{j \geq 0} \frac{\lambda_{i,j} + \delta_{i,j}}{\pi_{i,j}} \leq 1$
RUA	$\delta^b_{i,j}, \delta^u_{i,j} > 0$	$l'_{i,j} = l_{i,j} + \left\lceil \frac{l_{i,j}}{\lambda_{i,j} - \delta^b_{i,j}} \right\rceil \delta^b_{i,j},\ l^*_{i,j} = l'_{i,j} + \left\lceil \frac{l'_{i,j}}{\lambda_{i,j}} \right\rceil \delta^u_{i,j}$	$u^*_{i,j} = \frac{\lambda_{i,j} + \delta^u_{i,j}}{\pi_{i,j}}$	$\sum_{i \in I} \max_{j \geq 0} \frac{\lambda_{i,j} + \delta^u_{i,j}}{\pi_{i,j}} \leq 1$

Table 3. Scheduler overhead accounting [6]

The lower response-time bound $b^{l*}_{i,j}$ for $\alpha_{i,j}$ using the late release strategy is

$$b^{l*}_{i,j} = \left\lceil \frac{l^*_{i,j}}{\lambda^*_{i,j}} \right\rceil \pi_{i,j}, \tag{10}$$

whereas using the early release strategy is

$$b^{l*}_{i,j} = \left\lfloor \frac{l^*_{i,j}}{\lambda^*_{i,j}} \right\rfloor \pi_{i,j}. \tag{11}$$

We now give a condensed version of the three cases. The proofs of the results and a more detailed study can be found in [6].

6.1 Response accounting

In the response accounting case an action will have the same utilization but its response-time bounds increase. We have that $\delta_{i,j} = \delta^b_{i,j}$.

We compute the new load of the action $\alpha_{i,j}$ as

$$l^*_{i,j} = l_{i,j} + \left\lceil \frac{l_{i,j}}{\lambda_{i,j} - \delta_{i,j}} \right\rceil \delta_{i,j}.$$

The scheduler overhead that an action experiences during one period has to be smaller than its limit, otherwise the action will not execute any real workload. The new limit and utilization of the action are the same as without overhead, i.e., $\lambda^*_{i,j} = \lambda_{i,j}$ and $u^*_{i,j} = u_{i,j} = \frac{\lambda_{i,j}}{\pi_{i,j}}$. Since $l^*_{i,j} > l_{i,j}$ and $\lambda^*_{i,j} = \lambda_{i,j}$, we get that both the upper and the lower response-time bounds increase in case of response accounting.

Proposition 2 ([6]). *Let $\mathcal{P} = \{P_i(u_i) \mid 1 \leq i \leq n\}$ be a set of VBS processes each with bandwidth cap u_i. If $\sum_{i=1}^n u_i \leq 1$ and $\delta_{i,j} < \lambda_{i,j}$, with $\delta_{i,j}, \lambda_{i,j}$ as defined above, then the set of processes are schedulable with respect to the new response-time bounds $b^{u*}_{i,j}$ and $b^{l*}_{i,j}$, in the presence of worst-case scheduler overhead.*

The jitter for any action $\alpha_{i,j}$ in the response accounting case is at most $b^{u*}_{i,j} - b^l_{i,j}$.

For further reference, we write the new load in the response accounting case as a function

$$RA(l, \lambda, \delta) = l + \left\lceil \frac{l}{\lambda - \delta} \right\rceil \delta.$$

6.2 Utilization accounting

In the utilization accounting case an action is allowed to execute for more time than its original limit within a period and hence its utilization will increase (since the period duration remains the same). We have that $\delta_{i,j} = \delta^u_{i,j}$. The new load of action $\alpha_{i,j}$ becomes

$$l^*_{i,j} = l_{i,j} + \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \delta_{i,j}.$$

The new limit is $\lambda^*_{i,j} = \lambda_{i,j} + \delta_{i,j}$, and the new utilization is

$$u^*_{i,j} = \frac{\lambda_{i,j} + \delta_{i,j}}{\pi_{i,j}}.$$

Proposition 3 ([6]). *Given a set of processes $\mathcal{P} = \{P_i(u_i) \mid 1 \leq i \leq n\}$, let*

$$u^*_i = \max_{j \geq 0} \frac{\lambda_{i,j} + \delta_{i,j}}{\pi_{i,j}}.$$

If $\sum_{i=1}^n u^*_i \leq 1$, then the set of processes $\mathcal{P}$ is schedulable with respect to the original response-time bounds $b^u_{i,j}$ and $b^l_{i,j}$ defined in Section 2.3, in the presence of worst-case scheduler overhead.

Since the response-time bounds do not change in the utilization accounting case, the jitter for any action is the same as in the analysis without overhead.

We write the new load again as a function

$$UA(l, \lambda, \delta) = l + \left\lceil \frac{l}{\lambda} \right\rceil \delta.$$

6.3 Combined Accounting

The combined accounting case allows the total scheduler overhead to be accounted for both in the response-time bounds and in the utilization of an action. This allows a system designer for example to increase the utilization of the actions up to available CPU bandwidth and account the rest of the scheduler overhead in the response-time bounds, thus only delaying the finishing of action by the smallest possible amount.

We have that $\delta_{i,j} = \delta^b_{i,j} + \delta^u_{i,j}$, $\delta^b_{i,j} > 0$, and $\delta^u_{i,j} > 0$. Given an action $\alpha_{i,j}$ with its associated virtual periodic resource $R_{i,j} = (\lambda_{i,j}, \pi_{i,j})$, and load $l_{i,j}$, the new load

$l^*_{i,j}$ is computed in two steps. First we account for the overhead that increases the response time

$$l'_{i,j} = l_{i,j} + \left\lceil \frac{l_{i,j}}{\lambda_{i,j} - \delta^b_{i,j}} \right\rceil \delta^b_{i,j}$$

and then we add the overhead that increases the utilization

$$l^*_{i,j} = l'_{i,j} + \left\lceil \frac{l'_{i,j}}{\lambda_{i,j}} \right\rceil \delta^u_{i,j}.$$

The load function for the combined case is therefore

$$RUA(l, \lambda, \delta^b, \delta^u) = UA(RA(l, \lambda, \delta^b), \lambda, \delta^u).$$

The new limit for action $\alpha_{i,j}$ is $\lambda^*_{i,j} = \lambda_{i,j} + \delta^u_{i,j}$, and the utilization becomes

$$u^*_{i,j} = \frac{\lambda_{i,j} + \delta^u_{i,j}}{\pi_{i,j}}.$$

The upper response-time bound $b^{u*}_{i,j}$ for action $\alpha_{i,j}$ is now

$$b^{u*}_{i,j} = \left\lceil \frac{RUA(l_{i,j}, \lambda_{i,j}, \delta^b_{i,j}, \delta^u_{i,j})}{\lambda_{i,j} + \delta^u_{i,j}} \right\rceil \pi_{i,j} + \pi_{i,j} - 1.$$

The lower response-time bound $b^{l*}_{i,j}$ for the same action using the late release strategy is

$$b^{l*}_{i,j} = \left\lceil \frac{RUA(l_{i,j}, \lambda_{i,j}, \delta^b_{i,j}, \delta^u_{i,j})}{\lambda_{i,j} + \delta^u_{i,j}} \right\rceil \pi_{i,j},$$

and using the early release strategy is

$$b^{l*}_{i,j} = \left\lfloor \frac{RUA(l_{i,j}, \lambda_{i,j}, \delta^b_{i,j}, \delta^u_{i,j})}{\lambda_{i,j} + \delta^u_{i,j}} \right\rfloor \pi_{i,j}.$$

Proposition 4 ([6]). *Given a set of processes* $\mathcal{P} = \{P_i(u_i) \mid 1 \le i \le n\}$, *let*

$$u^*_i = \max_{j \ge 0} \frac{\lambda_{i,j} + \delta^u_{i,j}}{\pi_{i,j}}.$$

If $\sum_{i=1}^{n} u^*_i \le 1$, *then the set of processes* $\mathcal{P}$ *is schedulable with respect to the response-time bounds* $b^{u*}_{i,j}$ *and* $b^{l*}_{i,j}$, *in the presence of worst-case scheduler overhead.*

6.4 Optimizations

In [6] we also discuss optimizations to the combined and utilization accounting cases presented before. The optimization possibility comes from the over-approximation of the estimate of the number of scheduler invocations during an action period. During one period of an action any other process can have a release triggering a scheduler invocation. In the above cases, these invocations are accounted for in the response-time bounds or utilization of that action. However, the invocations due to the same release are incorporated as overhead for more than one action, meaning that scheduler overhead is sometimes accounted for more than necessary.

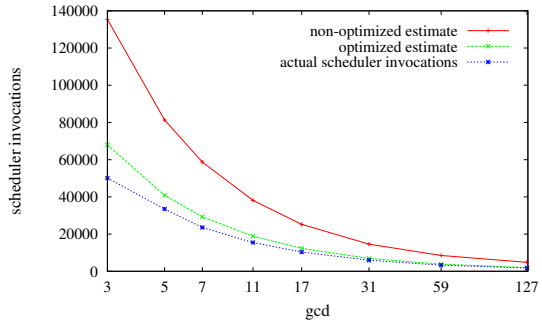


Fig. 6. Accuracy of the estimate on scheduler invocations [6]

In order to separate the reasons for scheduler invocations, we observe a natural division of the VBS scheduler overhead into overhead due to releasing and due to suspending processes. Scheduler invocations that occur due to releasing of processes can be improved by accounting for them in a separate, virtual VBS process instead of the given VBS processes. We call this process the scheduler process. The scheduler process is then accounted for in increased overall CPU utilization. The remaining overhead due to suspending processes may then be accounted for using the methods described above.

A sufficient condition for improvement of the estimate is that there exists a process P_m that accounts, in the original estimate, for as many scheduler invocations as the scheduler process, i.e., $gcd(\{\pi_{i,j} \mid i \in I, j \ge 0\})$ equals $gcd(\{\pi_{i,j} \mid i \in I, j \ge 0, i \ne m\})$.

We denote the scheduler process by P_S with all actions equal $\alpha_{S,j} = (\xi, R_S)$ where $R_S = (\lambda_S, \pi_S) = (\xi, gcd(\{\pi_{i,j} \mid i \in I, j \ge 0\})$. Thus, the utilization of the scheduler process is

$$u_S = \frac{\xi}{gcd(\{\pi_{i,j} \mid i \in I, j \ge 0\})}.$$

In the combined accounting case an optimization is possible only in the case $\delta^u_{i,j} = N^R_{i,j} \cdot \xi$ and $\delta^b_{i,j} = \xi$. In the utilization accounting case the limit of the action $\alpha_{i,j}$ becomes $\lambda^*_{i,j} = \lambda_{i,j} + \xi$ and therefore the utilization is

$$u^*_{i,j} = \frac{\lambda_{i,j} + \xi}{\pi_{i,j}}.$$

An important property of the scheduler process is that the period of its action is smaller than or equal to any other period in the system meaning that it can always execute at the appropriate time and therefore does not change the schedulability result.

In order to see the result of the optimization on the scheduler invocation estimates we conducted a simulation experiment (Figure 6) in [6] showing a comparison of the global optimized and non-optimized estimates, and the total number of actual scheduler invocations measured for three concurrently running actions over 100000 time units. The periods of the actions are chosen so that

they result in an increasing gcd (logarithmic x-axis). The periods are actually multiples of the gcd by $2, 3$, and 5. For small values of the gcd the accuracy of the optimized estimate is considerably better than the accuracy of the non-optimized estimate. For large values of the gcd, the estimates converge.

7 Power-aware VBS

An important non-functional aspect of real-time systems is power consumption. Many modern processors contain mechanisms that enable dynamic voltage and frequency scaling (DVS) [18,19]. In the area of real-time systems, power-aware scheduling mechanisms make use of DVS in order to allow processes to maintain their real-time properties while reducing the overall CPU power consumption. Power-aware real-time scheduling, e.g. for EDF and rate-monotonic, has been extensively studied [19–22]. Similarly, there has also been research concerning power-aware server mechanisms [23,24]. We have introduced and discussed a power-aware version of VBS in [7]. Here, we briefly present some of our findings in order to give a complete picture of VBS scheduling.

In a simplified CPU power model [19], the consumed CPU power P is proportional to the operating frequency f and the square of the voltage level V, i.e., $P \propto f \cdot V^2$ [18]. Since a reduced voltage imposes a maximal available frequency level and reducing the operating frequency extends the execution time of a workload [19], the scheduling mechanism has to find the minimal possible frequency at which the processes still meet their deadlines.

A reduction in the operating frequency is only possible when there is so-called slack in the system, i.e., when the scheduled processes do not use 100% of the CPU bandwidth.

Since the VBS process model is different from the EDF or CBS process model, in [7], we distinguish two types of slack specific to VBS systems, namely static and dynamic VBS slack. The static slack results from the predefined bandwidth caps of the VBS processes whereas the dynamic slack results from the individual actions of each VBS process. Furthermore, we divide the dynamic slack in action and termination slack. We next briefly describe the implications on power consumption of each type of slack.

Static slack. In the VBS process model, each process P_i has a bandwidth cap u_i which represents the maximum utilization that any of its actions may have. It has been shown in [19,7] that if the total utilization of a set of EDF processes is $U \in [0, 1]$, the frequency f can be computed as $f = U \cdot f_{max}$, where f_{max} is the maximal available frequency, such that no process will violate its deadline. As a consequence, if the sum of all VBS bandwidth caps is less than 1, we can safely scale down the processor frequency to f without any consequence on the response-time bounds of actions. The computed frequency f is set once, before the system runs, and is not modified during runtime.

Dynamic slack. Due to the action model of VBS, sometimes slack arises during runtime such that at certain points in time the frequency can be scaled lower than the aforementioned computed value. The important aspect of these dynamic frequency scalings is that we have to make sure that we adapt the frequency to the currently available slack so that no action will miss deadlines and the response-time bounds of actions remain unchanged. We distinguish two types of dynamic slack.

- *Termination slack*, resulting from the VBS termination strategy.
- *Action slack*, generated by an action having utilization less than the bandwidth cap of the process.

The termination strategy, explained in Section 2.2 postpones the logical termination of an action to the end of the last period of that action. We can generate slack at runtime by computing at every arrival of a new action the lowest possible limit such that the action still finishes its load within the original number of periods, i.e, such that the original response-time bound is maintained.

For example, an action $\alpha = (55, (30, 100))$ could run for 28 time units every period and still meet its response time bound 200. Therefore the virtual periodic resource for this action can be changed from $(30, 100)$ to $(28, 100)$ and the resulting slack of 2 time units per period can be used to scale down the processor.

More formally, at every arrival time t of an action, a new limit is computed as follows:

$$\lambda^*_{i,j} = \left\lceil \frac{l_{i,j}}{n_{i,j}} \right\rceil,$$

where $n_{i,j} = \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil$ is the number of periods needed for the action $\alpha_{i,j}$ to finish its load. Note that the new limit never exceeds the old limit ($\lambda^*_{i,j} \leq \lambda_{i,j}$), so this change does not influence schedulability. The action $\alpha_{i,j}$ will thus be transformed into an action $\alpha^*_{i,j} = (l_{i,j}, (\lambda^*_{i,j}, \pi_{i,j}))$.

The second type of dynamic slack is the action slack which is a result of newly arriving actions for a VBS process. Dynamic slack is generated when the utilization of the new action is lower than the bandwidth cap for the process. In [7] we show that it is possible to scale the frequency at runtime by a new scaling factor computed as the sum of remaining utilizations of the active actions (the current actions of each VBS process). The computation and scaling of the frequency happens at every time instant when an action has a release.

Proposition 5 ([7]). *Let $\mathcal{P} = \{P_i(u_i) \mid 1 \leq i \leq n\}$ be a schedulable set of VBS processes, with a total utilization*

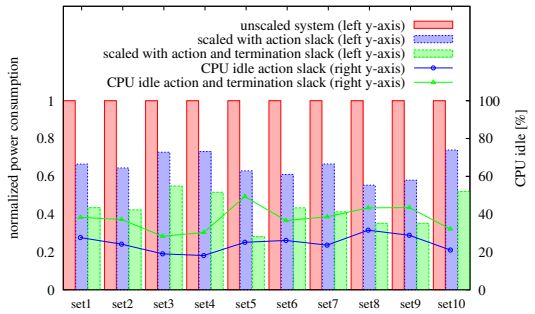


Fig. 7. Normalized power consumption for 10 sets of random processes [7].

cap $U = \sum_{i=1}^{n} u_i \leq 1$ *and corresponding process actions* $\alpha_{i,j}$*, with virtual periodic resources* $(\lambda_{i,j}, \pi_{i,j})$*, for* $j \geq 0$*, of process* P_i*. This set of processes is schedulable within the response-time bounds if in between two action releases the processor frequency is at least* $f_{new} = U_c \cdot f_{max}$ *where* $U_c = \sum_{i=1}^{n} \frac{\lambda_{i,j_i}}{\pi_{i,j_i}}$ *is the total utilization of all released actions* α_{i,j_i} *in the considered interval of time between two action releases.*

Since the two types of dynamic slack address separate aspects of VBS process execution, they can be exploited separately or together. Using only the termination slack may reduce the actual response time jitter of the action, while using only the action slack does not modify the original limit of the action. If both types of dynamic slack are exploited, the minimum possible operating frequency is achieved and CPU utilization is maximized.

We present an experiment (Figure 7) where we show the power savings using action slack alone, and action slack combined with termination slack for 10 sets of randomly-generate simulated VBS-processes. The left y-axis shows the normalized power consumption while the right y-axis shows the CPU idle percentage. As expected, we see that higher CPU idle time results in lower power consumption and that the combined method results in the lowest power consumption as opposed to the unscaled system and the system scaled using the action slack only.

The presented methods for reducing power consumption based on static and dynamic slack adhere strictly to the VBS process model described in Section 2. Diverging from the strict model and allowing the scheduler to redistribute computation time of process actions among the server periods during which the actions execute, results in even lower power consumption without affecting the actions' original response-time bounds. In order to be able to redistribute computation time between periods the power-aware VBS scheduler must be aware of the future, i.e., it must know the sequence of action changes of every VBS process in advance.

Reducing power consumption using future knowledge depends on the specific power model of the CPU (represented through a power consumption function) as well as the frequency switching overhead. In [7] we (a subset of the authors) present an optimal offline algorithm that computes the best possible configuration of server bandwidths for every period of an action during the whole lifetime of a system thus minimizing the given power-consumption function. We show that it is also possible to incorporate frequency switching overhead into the computation. Since the offline method may not be feasible for a real system, we also give an approximate to the optimal offline method using a feasible online algorithm. Given a simplified power consumption model, we show (c.f. [7]) that it is possible to approximate the optimal offline results by decreasing CPU utilization jitter, i.e., the actual CPU utilization is steered towards a computed average.

8 Related Work

There is a significant body of research concerning temporal isolation. We first present several mechanisms that enable some form of temporal isolation for resources like CPU, disk, and I/O.

One of the the first models proposed is the fair queueing model in the context of communication networks [25, 26]. In a simplified version, each communication source is serviced in a dedicated queue of infinite storage size and the queues are handled in round-robin fashion. Isolation is provided in the sense that a source will not affect other sources if it receives more requests than specified.

The Generalized Processor Sharing (GPS) approach [27] also allows communication sources to be isolated in their temporal behavior under the assumption that the network traffic is infinitely divisible. Both [27] and [26] describe approximations of the GPS algorithm that are viable for real systems. The methods described above are similar to proportional share allocation (PSA) introduced in [28,29] which allows weighted sharing of computing resources in isolation. Each shared resource is discretized in quanta of size q and each process is assigned a fraction of the resources through a predefined weight. The deviation of this model from the ideal one is bounded by q [29].

In the context of CPU scheduling for real-time systems, the concept of CPU capacity reserves was introduced in [30,31] for both real-time and non-real-time processes. Virtual periodic resources [8] are similar to CPU capacity reserves but are used to describe periodic resource allocation used in compositional timing analysis. We describe a more general form of resource reservations that are not restricted to the CPU resource. The resource reservation concept is extended in [32] to include other system resources with a focus on QoS guarantees. The model in [32] is similar to ours but uses resource fractions instead of limits and periods. As a consequence, there are no process actions and hence no

deadlines that would enable EDF [2] scheduling. A similar model is found in the Rialto [33,34] system but lacks the concept of sequential process actions from the VBS model. SMART [35] is another related scheduling model intended for adaptive multimedia and soft real-time processes. Similar to our approach, SMART allows processes to vary their alloted reservation but is not designed to support hard deadlines.

Server mechanisms [3] are another form of resource reservations. A server is usually defined by a server capacity or limit (C_S) and a server period (T_S) [13]. A process that is encapsulated within a server will execute at most C_S time units in a time window of T_S time units. A large variety of server mechanisms have been introduced, e.g., [36,11,12,37,3,38].

The constant-utilization server (CUS) [37,39] and the total-bandwidth server (TBS) [38] are very similar to VBS but do not have the ability to change the server limit and period at runtime. There is no notion of sequentiality within a process, i.e., there is no counterpart of our action model.

The work on constant-bandwidth servers (CBS) [3] is highly related to ours, as already elaborated in the previous sections. Similar to CBS, VBS also uses an EDF-based algorithm for scheduling. The drawback of a CBS, like with CUS and TBS, is that its resource's limit and period cannot be changed. As elaborated before, a process may sometimes need to execute a small portion of its code with lower latency than the rest of its code and therefore temporarily require a shorter period [4,40]. We next present several related mechanisms that allow such a change in the rate of execution within the process.

RBED [41,42] is a rate-based scheduler that involves a modified EDF process model which allows the bandwidth (limit) and rate (period) of processes to be changed at runtime. RBED is most closely related to VBS as it also incorporates dynamic adjustments in process parameters. RBED and VBS differ on the level of abstraction: in VBS, processes are modeled as sequences of actions to quantify the response times of portions of process code where each transition from one action to the next offers the possibility of parameter adjustment. RBED provides ranges of feasible reconfiguration but does not specify any higher-level mechanism for changing the period or limit of processes. Moreover, the reconfiguration is done within the limits of the currently available system utilization making an offline analysis difficult.

Elastic scheduling [43,44] introduces a new process model in conjunction with EDF, which views process utilization as a spring with a certain elasticity and maximum length configuration. Given the configuration, processes can change both limit and period within the resulting constraints. In [45], CBS are dynamically reconfigured using a benefit function through genetic algorithms. The goal of this and similar approaches, such as [46], is mainly to handle reconfiguration of processes and the potentially ensuing overloads in a more robust manner by performing adaptations based on the current system utilization. The VBS model offers flexibility by defining processes whose throughput and latency change at runtime through individual actions, and therefore differs in implementation and goal from the mentioned approaches.

In [47], the authors discuss dynamic reconfiguration of servers based on TDMA that enables a change in both limit and period of the TDMA partitioning allocation. The authors present schedulability analyses and algorithms for all possible reconfiguration cases.

In the context of virtualization research, XEN [48] employs three schedulers which were compared and discussed in [49]: borrowed virtual time [50], a weighted proportional-share scheduler; SEDF [51], a modified version of EDF; and a credit scheduler that allows automatic load balancing. Another scheduling method for XEN, which gives more attention to I/O-intensive domains, was presented in [52].

Other areas of related work that enable either temporal isolation or dynamic reconfiguration of systems are strongly partitioned systems (e.g. [39,53–55]) and mode switches in real-time systems (cf. [56]). Our system can be seen as scheduling partitions, namely processes correspond to partitions and inside a partition actions are sequentially released processes. Similarly, on an abstract level, actions of VBS processes can be interpreted as different modes of the same process and the VBS mechanism as providing safe mode changes. The scheduling goal, process model, and methods, however, are in both cases different from the mentioned areas of research.

Finally, we compare the complexity of our queue management plugins and scheduler to other work. By n we denote the number of processes. The SMART [35] scheduler's time complexity is determined by the complexity of special list operations and the complexity of schedule operations. The list operations complexity is $O(\log(n))$ if tree data structures are used and $O(n)$ otherwise. The schedule complexity is $O(n_R^2)$, where n_R is the number of active real-time processes with higher priority than non-real-time processes. The authors in [35] point out that, in special cases, the complexity can be reduced to $O(n)$ and even $O(1)$. The Move-to-Rear List scheduling algorithm [32] has $O(log(n))$ complexity. A recent study [57] on EDF-scheduled systems describes an implementation mechanism for queue operations using deadline wheels where the resulting time and space complexities are similar to those achieved by our four queue mechanisms.

9 Conclusions

We have presented a comprehensive study of theoretical and practical aspects of variable-bandwidth servers (VBS) [1]. In particular, we focused on the VBS process

model and schedulability results with detailed proofs. Furthermore, we have presented implementation details and experimental results involving four queue management plugins that allow trading-off time and space complexity of the VBS scheduler. We have also presented a survey of results on scheduler overhead accounting [6] and power-aware scheduling with VBS [7].

We now briefly discuss possible future work. On the practical side, it is certainly interesting to see VBS in action, i.e., scheduling realistic case studies (beyond synthetic benchmarks). Another practical issue would be to exploit even further the possible ways to efficiently implement the queues, e.g. using (binary) heap(s) [58].

On the conceptual side, temporal isolation is just one type of process isolation. Another well-known type of process isolation that is relevant in virtually all computer systems including non-real-time systems is spatial isolation. Processes that are spatially isolated do not tamper with the memory regions of one another, except when explicitly allowed. A third, more recently studied type of process isolation is power isolation where the amount of power consumed by a process may be approximated independently of the power consumed by other processes [59], or individually controlled and capped by a power manager [60,61]. We envision a system that provides full temporal, spatial, and power isolation of software processes simultaneously [62]. We have taken the first step towards this goal by studying process isolation with respect to each property individually, through the VBS approach for temporal isolation, the compact-fit memory management system [63] for spatial isolation in real time, and the work on power isolation in EDF-scheduled systems [59]. The key insight is that there appears to be a fundamental trade-off between quality and cost of time, space, and power isolation. Designing an integrated system that supports temporal, spatial, and power isolation remains the key challenge in this context.

References

1. S. S. Craciunas, C. M. Kirsch, H. Payer, H. Röck, and A. Sokolova, "Programmable temporal isolation through variable-bandwidth servers," in *Proc. SIES*. IEEE, 2009.
2. C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.
3. L. Abeni and G. Buttazzo, "Resource reservation in dynamic real-time systems," *Real-Time Syst.*, vol. 27, no. 2, pp. 123–167, 2004.
4. A. Cervin, "Improved scheduling of control tasks," in *Proc. ECRTS*. IEEE, 1999.
5. S. S. Craciunas, C. M. Kirsch, H. Röck, and A. Sokolova, "Real-time scheduling for workload-oriented programming," Department of Computer Sciences, University of Salzburg, Tech. Rep. 2008-02, September 2008.
6. S. S. Craciunas, C. M. Kirsch, and A. Sokolova, "Response time versus utilization in scheduler overhead accounting," in *Proc. RTAS*. IEEE, 2010.
7. S. S. Craciunas, C. M. Kirsch, and A. Sokolova, "Power-aware temporal isolation with variable-bandwidth servers," in *Proc. EMSOFT*. ACM, 2010.
8. I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Proc. RTSS*. IEEE, 2003.
9. I. Shin and I. Lee, "Compositional real-time scheduling framework," in *Proc. RTSS*. IEEE, 2004.
10. M. G. Harbour, M. H. Klein, and J. P. Lehoczky, "Timing analysis for fixed-priority scheduling of hard real-time systems," *IEEE Transactions on Software Engineering*, vol. 20, no. 1, pp. 13–28, 1994.
11. B. Sprunt, L. Sha, and J. P. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *Real-Time Syst.*, vol. 1, 1989.
12. 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*, vol. 44, no. 1, pp. 73–91, 1995.
13. G. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Norwell, MA, USA: Kluwer Academic Publishers, 1997.
14. S. S. Craciunas, C. M. Kirsch, H. Payer, H. Röck, A. Sokolova, H. Stadler, and R. Staudinger, "The Tiptoe system," 2007, http://tiptoe.cs.uni-salzburg.at.
15. G. Buttazzo and P. Gai, "Efficient implementation of an EDF scheduler for small embedded systems," in *Proc. OSPERT*, 2006.
16. R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indices," *Acta Informatica*, vol. 1, pp. 173–189, 1972.
17. D. F. Bacon, P. Cheng, and D. Grove, "Tuningfork: a platform for visualization and analysis of complex real-time systems," in *OOPSLA Companion*. ACM, 2007.
18. T. D. Burd and R. W. Brodersen, "Energy efficient CMOS microprocessor design," in *Proc. HICSS*. IEEE, 1995.
19. P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *Proc. SOSP*. ACM, 2001.
20. H. Aydin, P. Mejía-Alvarez, D. Mossé, and R. Melhem, "Dynamic and aggressive scheduling techniques for power-aware real-time systems," in *Proc. RTSS*. IEEE, 2001.
21. A. Qadi, S. Goddard, and S. Farritor, "A dynamic voltage scaling algorithm for sporadic tasks," in *Proc. RTSS*. IEEE, 2003.
22. D. Shin and J. Kim, "Dynamic voltage scaling of periodic and aperiodic tasks in priority-driven systems," in *Proc. ASP-DAC*. IEEE Press, 2004, pp. 27–60.
23. M. P. Lawitzky, D. C. Snowdon, and S. M. Petters, "Integrating real time and power management in a real system," in *Proc. OSPERT*, 2008.
24. C. Scordino and G. Lipari, "Using resource reservation techniques for power-aware scheduling," in *Proc. EMSOFT*. ACM, 2004.
25. J. Nagle, "On packet switches with infinite storage," *IEEE Transactions on Communications*, vol. 35, no. 4, pp. 435–438, 1987.
26. A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," *SIGCOMM Comput. Commun. Rev.*, vol. 19, pp. 1–12, 1989.

27. A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case," *IEEE/ACM Trans. Netw.*, vol. 1, pp. 344–357, 1993.
28. K. Jeffay, F. D. Smith, A. Moorthy, and J. Anderson, "Proportional share scheduling of operating system services for real-time applications," in *Proc. RTSS*. IEEE Computer Society, 1998.
29. I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton, "A proportional share resource allocation algorithm for real-ti time-shared systems," in *Proc. RTSS*. IEEE Computer Society, 1996.
30. C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves for multimedia operating systems," Carnegie Mellon University, Tech. Rep., 1993.
31. C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: Operating system support for multimedia applications," in *Proc. ICMCS*, 1994.
32. J. Bruno, E. Gabber, B. Özden, and A. Silberschatz, "Move-to-rear list scheduling: a new scheduling algorithm for providing QoS guarantees," in *Proc. MULTIMEDIA*. ACM, 1997.
33. M. Jones, P. Leach, R. Draves, and J. Barrera, "Modular real-time resource management in the Rialto operating system," in *Proc. HOTOS*. IEEE, 1995.
34. M. B. Jones, D. Roşu, and C. Roşu, "CPU reservations and time constraints: efficient, predictable scheduling of independent activities," in *Proc. SOSP*. ACM, 1997.
35. J. Nieh and M. S. Lam, "The design, implementation and evaluation of SMART: a scheduler for multimedia applications," in *Proc. SOSP*. ACM, 1997.
36. J. P. Lehoczky, L. Sha, and J. K. Strosnider, "Enhanced aperiodic responsiveness in hard real-time environments," in *Proc. RTSS*. IEEE, 1987.
37. Z. Deng, J. W.-S. Liu, and S. Sun, "Dynamic scheduling of hard real-time applications in open system environment," University of Illinois at Urbana-Champaign, Tech. Rep., 1996.
38. M. Spuri and G. C. Buttazzo, "Scheduling aperiodic tasks in dynamic priority systems," *Journal of Real-Time Systems*, vol. 10, no. 2, pp. 179–210, 1996.
39. Z. Deng, J. W.-S. Liu, L. Zhang, S. Mouna, and A. Frei, "An open environment for real-time applications," *Real-Time Syst.*, vol. 16, no. 2-3, pp. 155–185, 1999.
40. A. Cervin and J. Eker, "The Control Server: A computational model for real-time control tasks," in *Proc. ECRTS*. IEEE, 2003.
41. S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson, "Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes," in *Proc. RTSS*. IEEE, 2003.
42. S. Goddard and X. Liu, "Scheduling aperiodic requests under the rate-based execution model," in *Proc. RTSS*. IEEE, 2002.
43. G. Buttazzo and L. Abeni, "Adaptive workload management through elastic scheduling," *Real-Time Syst.*, vol. 23, no. 1-2, pp. 7–24, 2002.
44. G. C. Buttazzo, G. Lipari, and L. Abeni, "Elastic task model for adaptive rate control," in *Proc. RTSS*. IEEE, 1998.
45. M. A. C. Simoes, G. Lima, and E. Camponogara, "A GA-based approach to dynamic reconfiguration of real-time systems," in *Proc. APRES*, 2008.
46. G. Beccari, M. Reggiani, and F. Zanichelli, "Rate modulation of soft real-time tasks in autonomous robot control systems," in *Proc. ECRTS*, 1999.
47. N. Stoimenov, L. Thiele, L. Santinelli, and G. Buttazzo, "Resource adaptations with servers for hard real-time systems," in *Proc. EMSOFT*. ACM, 2010.
48. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proc. SOSP*. ACM, 2003.
49. L. Cherkasova, D. Gupta, and A. Vahdat, "Comparison of the three CPU schedulers in Xen," *SIGMETRICS Perform. Eval. Rev.*, vol. 35, no. 2, pp. 42–51, 2007.
50. K. J. Duda and D. R. Cheriton, "Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler," *SIGOPS Oper. Syst. Rev.*, vol. 33, no. 5, pp. 261–276, 1999.
51. I. M. Leslie, D. Mcauley, R. Black, T. Roscoe, P. T. Barham, D. Evers, R. Fairbairns, and E. Hyden, "The design and implementation of an operating system to support distributed multimedia applications," *IEEE JSAC*, vol. 14, no. 7, pp. 1280–1297, 1996.
52. S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam, "Xen and co.: communication-aware cpu scheduling for consolidated Xen-based hosting platforms," in *Proc. VEE*. ACM, 2007.
53. D. Kim, Y.-H. Lee, and M. Younis, "SPIRIT-μKernel for strongly partitioned real-time systems," in *Proc. RTCSA*. IEEE, 2000.
54. D. Kim and Y.-H. Lee, "Periodic and aperiodic task scheduling in strongly partitioned integrated real-time systems," *Comput. J.*, vol. 45, no. 4, pp. 395–409, 2002.
55. G. Lipari and E. Bini, "A methodology for designing hierarchical scheduling systems," *J. Embedded Comput.*, vol. 1, no. 2, pp. 257–269, 2005.
56. J. Real and A. Crespo, "Mode change protocols for real-time systems: A survey and a new proposal," *Real-Time Syst.*, vol. 26, pp. 161–197, 2004.
57. M. Short, "Improved task management techniques for enforcing EDF scheduling on recurring tasks," in *Proc. RTAS*. IEEE, 2010.
58. C. Stein, T. Cormen, R. Rivest, and C. Leiserson, *Introduction To Algorithms*. MIT Press, 2001.
59. S. S. Craciunas, C. M. Kirsch, and A. Sokolova, "The power of isolation," Department of Computer Sciences, University of Salzburg, Tech. Rep. 2011-02, July 2011.
60. X. Liu, P. Shenoy, and M. Corner, "Chameleon: application level power management with performance isolation," in *Proc. MULTIMEDIA*. ACM, 2005.
61. Q. Cao, D. Fesehaye, N. Pham, Y. Sarwar, and T. Abdelzaher, "Virtual battery: An energy reserve abstraction for embedded sensor networks," in *Proc. RTSS*. IEEE Computer Society, 2008, pp. 123–133.
62. S. S. Craciunas, A. Haas, C. M. Kirsch, H. Payer, H. Röck, A. Rottmann, A. Sokolova, R. Trummer, J. Love, and R. Sengupta, "Information-acquisition-as-a-service for cyber-physical cloud computing," in *Proc. HotCloud*. USENIX, 2010.
63. S. S. Craciunas, C. M. Kirsch, H. Payer, A. Sokolova, H. Stadler, and R. Staudinger, "A compacting real-time memory management system," in *Proc. USENIX ATC*. USENIX, 2008.