Programmable Temporal Isolation
through Variable-Bandwidth Servers[*]

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

Department of Computer Sciences
University of Salzburg, Austria
Email: firstname.lastname@cs.uni-salzburg.at

Abstract—We introduce variable-bandwidth servers (VBS) for scheduling and executing processes under programmable temporal isolation. A VBS is an extension of a constant-bandwidth server where throughput and latency of process execution can not only be controlled to remain constant across different competing workloads but also to vary in time as long as the resulting bandwidth stays below a given bandwidth cap. We have designed and implemented a VBS-based EDF-style constant-time scheduling algorithm, a constant-time admission test, and four alternative queue management plugins which influence the scheduling algorithm's overall temporal and spatial complexity. Experiments confirm the theoretical bounds in a number of microbenchmarks and demonstrate that the scheduler can effectively manage in constant time any number of processes up to available memory while maintaining response times of individual processes within a bounded range. We have also developed a small-footprint, bare-metal virtual machine that uses VBS for temporal isolation of multiple, concurrently running processes executing real code.

I. INTRODUCTION

Virtualization has always been a fascinating topic in systems research and more recently lead to impressive success stories in industry. The key benefit of virtualization is isolation. Software processes and even whole systems running on virtualized hardware may be effectively isolated from the specifics of real hardware but also from each other when sharing resources such as CPUs, memory, and I/O devices. However, virtualization typically comes at the expense of increasing the already complex temporal dependencies when sharing resources even further.

In this paper, we show that virtualization not only enables the well-known benefits of traditional CPU, memory, and I/O isolation but may also have the potential for temporally isolating access to shared resources. This is particularly relevant when using virtualization

in time-sensitive application areas such as control and automation but also mobile computing. Intuitively, the execution of a piece of sequential program code of a process (called action) is temporally isolated if the response times of the code as well as the variance of the response times (jitter) are solely determined by the code itself and its inputs, independently of any other, concurrently executing actions and the system on which the actions execute. The response time of an action is 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). In this model, process execution corresponds to a possibly infinite sequence of actions. We say that temporal isolation is programmable if response times and jitter can be modified by processes, at least within some platform-dependent range.

We introduce the notion of variable-bandwidth servers (VBS), which enable programmable temporal isolation of processes. VBS are a generalized form of constant-bandwidth servers (CBS) [1]. Given a pair (λ, π) called virtual periodic resource [2], where λ is the limit and π is the period of the resource, a CBS executes a single process for λ units of time every π units of time. In other words, a CBS discretizes or "samples" the progress of time at a "sampling frequency" of one unit of time over π. The virtual periodic resource of a CBS is fixed and therefore determines a constant server bandwidth (and sampling frequency). Multiple CBS are scheduled using earliest-deadline-first (EDF) scheduling with deadlines equal to the servers' periods. New servers and thus processes can simply be admitted to the system as long as the sum of the bandwidth of all servers is less than the system's capacity.

The drawback of a CBS is that its resource's period and limit cannot be changed. For example, 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. This is why we need VBS. A VBS merely has a fixed bandwidth cap but can otherwise switch to any virtual periodic resource

with a capacity less or equal to its bandwidth cap. In particular, a VBS can switch to any resource periods and limits as long as the resulting bandwidth does not exceed the bandwidth cap. Switching virtual periodic resources needs to follow a particular sequence of steps to avoid ever exceeding the bandwidth cap so that the admission of new servers can be handled in a similar fashion as in a CBS system, simply by checking that the sum of the bandwidth caps of all servers remains less or equal to the system's capacity. A process running on a VBS can initiate a switch from one action to the next.

Despite adapting to varying throughput and latency requirements, switching to different periods allows to trade off scheduling overhead and temporal isolation at runtime. Smaller periods and thus higher sampling frequencies better isolate servers because their response times for executing a given piece of code are better maintained across larger sets of server workloads, at the expense of higher administrative overhead through more scheduler invocations.

The key contribution of this paper is the design and implementation of a VBS-based system consisting of a constant-time scheduling algorithm, a constant-time admission test, and four alternative queue management plugins based on lists, arrays, matrices, and trees. The plugins trade off time and space complexity dominating the overall complexity of the implementation.

Our experiments with a high-performance, uniprocessor implementation of the scheduling algorithm and the plugins confirm the theoretical time and space bounds in a number of microbenchmarks. VBS workloads were simulated and thus not created by executing real code. In order to obtain real world benchmarks our implementation may readily be integrated into an existing kernel or virtual machine. However, we chose to develop from scratch a small-footprint, bare-metal virtual machine called Tiptoe [3] using VBS for scheduling in order to keep system complexity manageable and have full control over all relevant aspects including memory and I/O management. Our prototype implementation is meant to support mobile computing platforms. So far, it runs on an XScale 400MHz-processor with 64MB of RAM and virtualizes an AVR microcontroller. With the prototype, we have performed a bare-metal microbenchmark when executing AVR code. The experiment shows that VBS in Tiptoe effectively provides temporal isolation of multiple, concurrently running AVR instances.

The structure of the rest of the paper is as follows. We start by a discussion of related work in Section II. We then describe VBS conceptually in Section III and present the scheduling algorithm in Section IV. In Section V, we briefly present the implementation complexity

under the four different choices of queue management plugins. The results of our experiments are shown in Section VI. In Section VII we describe the integration of VBS into Tiptoe and report on the bare-metal experiment with VBS and Tiptoe. Section VIII gathers the conclusions.

II. RELATED WORK

Virtual periodic resources [2] are related to resource reservations, which were introduced in [4] as CPU capacity reserves. Follow-up work [5] within the real-time operating system Eclipse employs resource reservations (reserves) for additional resources. The scheduling model in [5] is very similar to ours, except that the resource reserve is a rate or a percentage of the resource that a process might use, and not a pair of a limit and a period. As a consequence, there is no notion of a deadline of a task that could be scheduled with classical algorithms. The Rialto [6] system also considers the possibility of multiple resources and uses an even stronger notion of resource reserves for resource management. However, there is no model of sequential process actions in the Rialto system. Another scheduler using reservation support via fair queuing is SMART [7].

The work on CBS [1] is highly related to ours, as already elaborated in the introduction. Similar to CBS, VBS also uses an EDF-based algorithm for scheduling. Another scheduling scheme for CBS has been developed in [8] for the purpose of scheduling multi-threaded, real-time and non-real-time applications running concurrently in an open system. There is no notion of sequentiality within a process there, i.e., no counterpart of our actions. RBED is a rate-based scheduler extending resource reservations [9] most closely related to VBS. It also uses EDF scheduling and allows dynamic bandwidth and rate adjustments. 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 marks an adjustment in bandwidth and rate.

The idea of decomposing a task into subtasks that run sequentially has also appeared before, in the context of fixed-priority scheduling [10], and was extended in [11] for solving control-related issues.

Several flexible scheduling solutions have been proposed that deal with the dynamic reconfiguration of task rates. Elastic scheduling [12], [13] proposes a new task model in conjunction with EDF, which is able to adjust task utilization parameters by treating tasks as springs with given elastic coefficients and constraints. The goal of this and similar approaches, such as [14], [15], is

to handle variable execution rates and overload scenarios in a flexible way by dynamically checking system utilization and adapting task parameters. In [16], CBS are dynamically reconfigured by redistributing processor time using a benefit function. Flexibility in our approach amounts to defining processes whose throughput and latency varies in time, and therefore differs in implementation and goal from the mentioned flexible scheduling approaches.

In the context of virtual machine monitors, XEN [17] employs three proportional share schedulers. Borrowed virtual time [18] is a fair share scheduler based on the concept of virtual time, which provides low-latency support for real-time and interactive applications. SEDF [19] is a modified version of EDF that distributes slack time fairly where the fairness depends on the period. Credit scheduler allows automatic load balancing of virtual CPUs across physical CPUs. Modifications to the SEDF scheduler to enable preferential scheduling of I/O-intensive domains by taking into consideration the amount of communication performed by each domain have also been proposed [20].

Another solution for temporal isolation in real-time systems is the strongly partitioned system concept. In such systems tasks are grouped into partitions and scheduled using a two-level scheduling structure [8], [21], [22], [23]. At partition level, the tasks inside a partition are scheduled using fixed-priority-based algorithms; at system level, partitions are assigned bandwidth and processor time in a cyclic fashion. Our system can be seen as scheduling partitions, namely processes correspond to partitions and inside a partition there are sequentially released tasks. However, the scheduling goal and methods in both cases are different.

Finally, we compare our scheduler implementation to other work regarding scheduling complexity. By n we denote the number of processes. The SMART [7] scheduler's time complexity is given by the complexity of managing a special list and the cost of managing the working schedule. The list requires $O(n)$ work, which can be reduced to $O(\log(n))$ if tree data structures are used. The worst-case complexity of managing the schedule is $O(n_R^2)$, where n_R is the number of some particular active real-time tasks. In special cases this complexity can be reduced to $O(n)$ and $O(1)$. The Move-to-Rear List scheduling of the Eclipse [5] operating system implies several operations that are constant time while in total it takes $O(n)$, which can also be optimized to $O(log(n))$ time. In the EDF-based scheduler of Rialto [24] the scheduling decision takes $O(1)$ time, but the scheduling algorithm is not compositional and requires a pre-computation of a so-called scheduling graph. The latest Linux 2.6 scheduler runs in $O(log(n))$ time. There is also an earlier $O(1)$ version, which, like our implementation, makes use of bitmaps to improve performance.

III. PROCESS SCHEDULING

We work with a discrete time domain, i.e., the set of natural numbers $\mathbb{N}$ is the timeline. The main ingredients of the scheduling model are variable-bandwidth servers (VBS) defined by virtual periodic resources and VBS-processes composed of sequential actions.

A. VBS and Processes

A virtual periodic resource (capacity) is a pair

$$R = (\lambda, \pi)$$

where λ stands for limit and π 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) [1] is uniquely determined by a virtual periodic resource $R = (\lambda, \pi)$. A constant-bandwidth server 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,

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

for $\alpha_i \in \text{Act}$, where $\text{Act} = \mathbb{N} \times \mathcal{R}$. An action $\alpha \in \text{Act}$ 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.

Given a set $\mathcal{R} = \{(1s, 2s), (1s, 4s), (1s, 3s)\}$ of resources, we consider a finite process $P(0.5)$ that first does some computation for $3s$ with a virtual periodic resource $(1s, 2s)$, then it works on allocating/deallocating memory objects of size 200KB, which takes $2s$ with the resource $(1s, 4s)$, then it produces output of size 100KB on an I/O device in $1s$ with $(1s, 3s)$, then again it computes, now for $2s$, with $(1s, 2s)$ 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 a $1s$-timeline. This process corresponds to (can be served by) a VBS with utilization $u = 0.5$ (or more).

B. 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 zero arrival time.

- 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.

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 whenever 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 response bounds b_i 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 given response bounds, i.e., $s_i \leq b_i$ for all involved actions α_i.

C. 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 $P_i(u_i) = \alpha_{i,0} \alpha_{i,1} \dots$ corresponds to a VBS with utilization u_i, we define response bounds

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

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 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

$$\sum_{i \in I} u_i \leq 1, \tag{2}$$

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

Hence, it is enough to test whether the sum of the utilization (bandwidth) cap of all processes is less than one. The test is finite even though the processes may be infinite because each process is a VBS-process. 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. The proof of Proposition 1 can be found in the full version technical report [25].

Let us still mention the two release strategies and elaborate the scheduling method via an example. 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 deferrable server [26] from classical scheduling theory, and the early strategy is similar in goal to the polling server [27]: it improves the average response times by servicing tasks that arrive during the current period.

Figure 1 presents the scheduling of an action $\alpha = (5s, (2s, 4s))$ with load of $5s$, arriving at time $10s$, in both strategies. The resource used by the action has a period of $4s$ and a limit of $2s$. In the late strategy, the action is only released at time $12s$, which is the next period instance after the actual arrival time. Then it takes three more periods for the action to finish. In
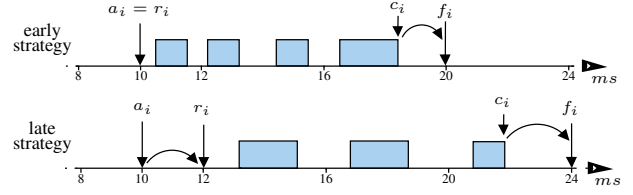

Fig. 1. Scheduling an action $\alpha = (5s, (2s, 4s))$

the early strategy, the action is released at once, but in the remaining time of the current period ($2s$) the limit is adjusted to $1s$, 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 $10s$, $12s$, and $16s$; have deadlines $12s$, $16s$, and $20s$; and durations $1s$, $2s$, and $2s$, respectively. In the late strategy the tasks are released at times $12s$, $16s$, and $20s$; have deadlines $16s$, $20s$, and $24s$; and durations of $2s$, $2s$, and $1s$, respectively. Our scheduling result shows schedulability of such sets of (sequences of) tasks using EDF, cf. [25].

Recall that $s_{i,j}$ denotes the scheduled response time of the action $\alpha_{i,j}$. The upper bound on $s_{i,j}$, i.e., $s_{i,j} = b_{i,j}$, if the schedulability test holds, occurs if the action arrives right after a new period begins, and can be reached in both release strategies. The scheduled response times also have a lower bound, that varies in both strategies. For the late release strategy we have

$$s_{i,j} \geq \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j}$$

and it is achieved if the action arrives at a period instance. Therefore the response-time jitter for the late release strategy is at most $\pi_{i,j} - 1$. For the early strategy, a more careful analysis provides lower and more accurate upper bounds. Let k be the smallest natural number such that

$$\left\lfloor k \frac{\lambda_{i,j}}{\pi_{i,j}} \right\rfloor \geq l_{i,j} - \left\lfloor \frac{l_{i,j}}{\lambda_{i,j}} \right\rfloor \lambda_{i,j}.$$

Then $k \in [0, \pi_{i,j}]$. The definition of k guarantees that the load of the action can be performed in $\left\lfloor \frac{l_{i,j}}{\lambda_{i,j}} \right\rfloor \pi_{i,j} + k$ time units. Now we give more precise lower and upper bounds for the early release strategy. If the action arrives early enough so that it can save one period of execution, i.e., $a_{i,j} \leq n_j \pi_{i,j} - k$ where n_j is such that $a_{i,j} \in ((n_j - 1)\pi_{i,j}, n_j \pi_{i,j}]$, then

$$s_{i,j} \geq \left\lfloor \frac{l_{i,j}}{\lambda_{i,j}} \right\rfloor \pi_{i,j} + k \geq \left\lfloor \frac{l_{i,j}}{\lambda_{i,j}} \right\rfloor \pi_{i,j}$$

and

$$s_{i,j} \leq \left\lfloor \frac{l_{i,j}}{\lambda_{i,j}} \right\rfloor \pi_{i,j} + \pi_{i,j} - 1.$$

Otherwise, if $a_{i,j} > n_j \pi_{i,j} - k$, in which case $k > 0$, then no time can be saved

$$s_{i,j} \geq \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j} + (k-1) \geq \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j}$$

and

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

In both cases the jitter is bounded by $\pi_{i,j} - 1$. Note that, if response time is defined as the time from arrival to completion, then the (non-logical) jitter is bounded by $2(\pi_{i,j} - 1)$ with both strategies.

The schedulability/admission test is a sufficient condition for schedulability. A more precise or even necessary condition is an interesting target for future work but may require incorporating details of process implementations and interactions.

IV. 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 Figure 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

$$P_i = (i, j, d_i, r_i, l_i^c, \lambda_i^c)$$

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_i^c is the current load, and λ_i^c 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

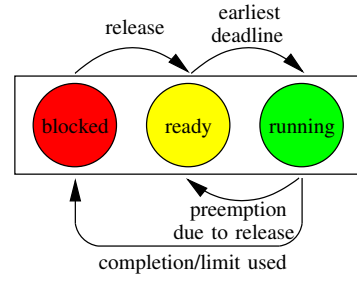Blocked $= \{P_1, \ldots, P_n\}$, Ready $=$ Running $= \emptyset$.

Fig. 2. Process states

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_i^c becomes $l_{i,j+1}$.

If $R_{i,j+1} = R_{i,j}$, 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_i^c, $\lambda_i^c \leftarrow \lambda_i^c - (t - t_s)$.

If $R_{i,j+1} \neq R_{i,j}$, we have currently implemented two release strategies handling the process, following the proof of Proposition 1. But 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, called 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,

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

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

The second strategy, called 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

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

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 (in a fair fashion) 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.

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

TABLE I
TIME AND SPACE COMPLEXITY PER PLUGIN

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.

V. IMPLEMENTATION

The scheduler implementation uses a well-defined interface to manage a queue of ready processes and a queue of blocked processes. The interface is implemented by four alternative plugins, each with different attributes regarding time complexity and space overhead. Currently, the implementation, available via the Tiptoe homepage [28], supports doubly-linked lists, time-slot arrays of FIFO queues, as well as a time-slot matrix of FIFO queues and a tree-based optimization of the matrix.

Table I 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 [29].

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 list-based implementation runs in quadratic time in the number of servers but only requires constant space. The array-based implementation runs in linear time in the number of servers but requires linear space in period resolution. For the implementation and complexity details on each of the plugins, we refer the reader to the full version technical report [25].

VI. EXPERIMENTS AND RESULTS

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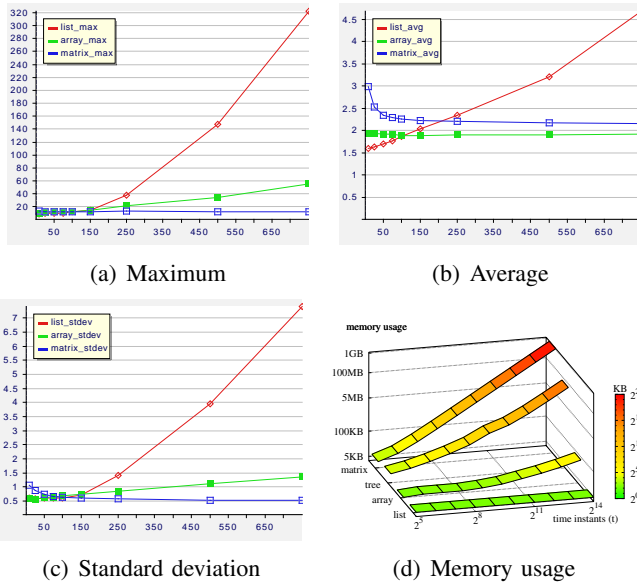
(a) Maximum   (b) Average

(c) Standard deviation   (d) Memory usage

Fig. 3.   Scheduler time and space overhead

(a) List   (b) Array

(c) Matrix   (d) Process releases

Fig. 4.   Execution times histograms

A. 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 $2^{14} = 16384$. During these experiments the execution time of every single scheduler invocation is measured using the software oscilloscope tool TuningFork [30]. From a sample of one million invocations we calculate the maximum (Figure 3(a)), the average (Figure 3(b)), and the standard deviation (Figure 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.

The execution time measurements conform to the complexity bounds from Section V. For a low number of processes (less than 150), all plugins perform similarly and the scheduler needs at most 20 microseconds. On average (Figure 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 Figure 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
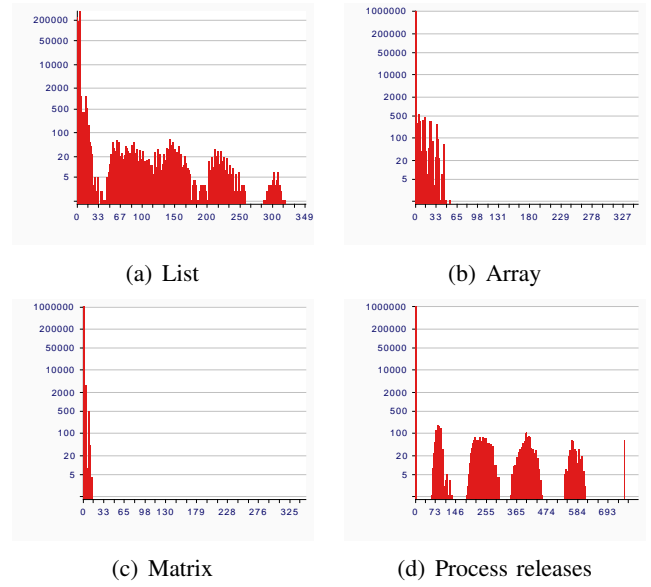
high number of processes and a higher standard deviation for a low number of processes. This is related to the better average execution time (Figure 3(b)) for higher number of processes, as a result of cache effects. By instrumenting the scheduler we discovered that bitmap functions, e.g. setting a bit, 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 Figure 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 Figure 3(d) are calculated from theoretical bounds. However, our experiments confirm the results.

Figures 4(a), 4(b), and 4(c) highlight 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 and the y-axis (log-scale) represents the number of scheduler calls. For example, in Figure 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 (Figure 4(d)). The x-axis

represents the number of processes and the y-axis (log-scale) represents how many times a certain number of processes is released. The similarity of Figure 4(a) and Figure 4(d) indicates that the release of processes dominates the execution of the scheduler for the experiment with 750 processes.

B. Release Strategies

We have compared the two implemented release strategies of the scheduler in experiments showing that the early strategy improves average response times over the late strategy both for a single process with increasingly non-harmonic periods and for an increasing number of processes with a random distribution of loads, limits, and periods. More details and figures presenting the results of the experiments can be found in the full version technical report [25].

VII. VBS INTEGRATION INTO A REAL VM

Tiptoe [3] is a small-footprint, bare-metal virtual machine, which currently runs on an XScale 400MHz-processor with 64MB RAM and uses VBS for scheduling. For development purposes, we can also run Tiptoe on Linux as a single user process. The bare-metal version comes with its own C library, device drivers for setting I/O pins, and a serial driver. The MMU is set up to provide a single linear static address space. The CPU exception code is mapped to the first physical page. There is also a microsecond timer framework and a 1KHz timer interrupt to keep the system synchronized with real time.

The current Tiptoe implementation interprets arbitrary AVR code virtualizing an Atmega128 processor with 4KB RAM and 128KB Flash storage. The Tiptoe VM schedules multiple interpreter instances using a unique VBS for each instance. A VBS is currently configured manually by setting its bandwidth cap in percentage of total CPU time. The interpreter instance assigned to the VBS may then configure any number of virtual periodic resources with application-dependent utilization levels below the bandwidth cap. Determining the appropriate limits and periods, also known as the server design problem, cf. [23], is left for future work. Each resource is associated with a unique action, i.e., a fixed piece of AVR code running on the interpreter instance. The AVR code marks the switch from one action to the next by writing a special I/O port of the virtualized Atmega128.

Context switching between interpreter instances is implemented cooperatively as follows. The AVR interpreter is invoked with a timeout determined by the VBS scheduler. At any time instant, the scheduler not only knows
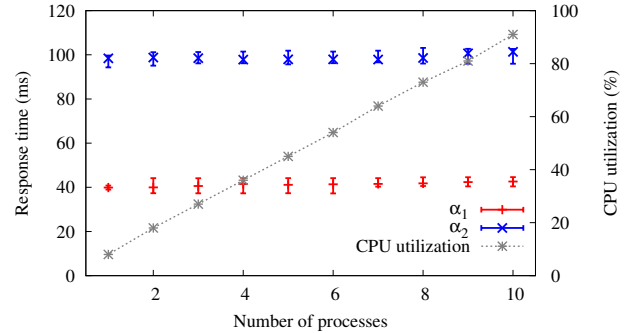

Fig. 5. VM experiment [3]

which instance needs to be executed next but also, by nature of its scheduling algorithm, for how much time the instance may execute before another scheduling decision must be made. Interpreter preemption can therefore be planned entirely. The AVR interpreter regularly returns cooperatively to the scheduler after its time has elapsed. Our bare-metal experiments have been conducted with this version of Tiptoe [3].

Ongoing work on Tiptoe focuses on further integrating VBS into the system as part of three research thrusts: a hypervisor version (for efficiency and legacy support), a byte code interpreter (for studying language-enabled memory protection using our compact-fit memory management [31]), and I/O management (for enabling real world experiments with our model helicopter [32]).

A. VM Experiment

We demonstrate the temporal isolation capabilities of our scheduler as well as its support for adapting the execution speed of portions of code to different latency requirements. Consider a process implementing a simple feedback controller that consists of two actions. Action α_1 is associated with the virtual periodic resource $R_1 = (320\mu s, 3550\mu s)$ while action α_2 uses the resource $R_2 = (500\mu s, 5340\mu s)$. Latency and jitter are critical to control systems, thus splitting a control process in two actions improves controller performance [11]. The process utilizes the CPU at around 9%. In order to show temporal isolation, we increase system utilization by starting additional processes, each utilizing the CPU at around 10%. Note that all processes execute actual AVR code (without performing any I/O).

Figure 5 shows the minimum, maximum, and average response times of action α_1 and α_2, respectively (left y-axis). The response time jitter of each action varies within two periods of the virtual periodic resource used by the respective action independently of the overall system utilization (right y-axis). CPU utilization increases from 9% when the measured process is the only process

in the system up to 92% when 9 additional processes run concurrently with the measured process (x-axis).

The theoretical bound for jitter is one period assuming zero scheduler overhead. In a real system, however, there is an additional administrative overhead. Nevertheless, the variance is still bounded and not influenced by the system utilization. Giving guaranteed bounds with non-zero scheduler overhead is a topic that we plan to pursue as future work.

VIII. Conclusions

We have introduced variable-bandwidth servers (VBS), and designed and implemented a VBS-based EDF-style constant-time scheduling algorithm, a constant-time admission test, and four alternative queue management plugins based on lists, arrays, matrices, and trees. Experiments confirm the theoretical bounds in a number of microbenchmarks. We have also developed a small-footprint, bare-metal virtual machine that uses VBS for temporal isolation of multiple processes executing real code. An interesting direction for future work is to study whether VBS can also control throughput and latency of other activities of the system such as memory and I/O management.

References

[1] L. Abeni and G. Buttazzo, "Resource reservation in dynamic real-time systems," *Journal of Real-Time Systems*, vol. 27, no. 2, pp. 123–167, 2004.

[2] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Proc. RTSS*. IEEE, 2003.

[3] S. Craciunas, C. Kirsch, H. Payer, H. Röck, and A. Sokolova, "Programmable temporal isolation in real-time and embedded execution environments," in *Proc. IIES*. ACM, 2009.

[4] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: Operating system support for multimedia applications," in *Proc. ICMCS*, 1994.

[5] 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.

[6] M. Jones, P. Leach, R. Draves, and J. Barrera, "Modular real-time resource management in the Rialto operating system," in *Proc. HOTOS*. IEEE, 1995.

[7] J. Nieh and M. S. Lam, "The design, implementation and evaluation of SMART: a scheduler for multimedia applications," in *Proc. SOSP*. ACM, 1997.

[8] Z. Deng, J. W.-S. Liu, L. Zhang, S. Mouna, and A. Frei, "An open environment for real-time applications," *Journal of Real-Time Systems*, vol. 16, no. 2-3, pp. 155–185, 1999.

[9] 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.

[10] M. G. Harbour, M. H. Klein, and J. P. Lehoczky, "Timing analysis for fixed-priority scheduling of hard real-time systems," *IEEE Trans. Softw. Eng.*, vol. 20, no. 1, pp. 13–28, 1994.

[11] A. Cervin, "Improved scheduling of control tasks," in *Proc. ECRTS*. IEEE, 1999.

[12] G. Buttazzo and L. Abeni, "Adaptive workload management through elastic scheduling," *Real-Time Syst.*, vol. 23, no. 1-2, pp. 7–24, 2002.

[13] G. Buttazzo, G. Lipari, and L. Abeni, "Elastic task model for adaptive rate control," in *Proc. RTSS*. IEEE, 1998.

[14] G. Beccari, M. Reggiani, and F. Zanichelli, "Rate modulation of soft real-time tasks in autonomous robot control systems," in *Proc. ECRTS*, 1999.

[15] T. Nakajima, "Resource reservation for adaptive qos mapping in real-time mach," in *Proc. WPDRTS*, 1998.

[16] M. A. C. Simoes, G. Lima, and E. Camponogara, "A GA-based approach to dynamic reconfiguration of real-time systems," in *Proc. APRES*, 2008.

[17] 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.

[18] 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.

[19] 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 Journal of Selected Areas in Communications*, vol. 14, no. 7, pp. 1280–1297, 1996.

[20] 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.

[21] D. Kim, Y.-H. Lee, and M. Younis, "SPIRIT-μKernel for strongly partitioned real-time systems," in *Proc. RTCSA*. IEEE, 2000.

[22] 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.

[23] G. Lipari and E. Bini, "A methodology for designing hierarchical scheduling systems," *J. Embedded Comput.*, vol. 1, no. 2, pp. 257–269, 2005.

[24] M. B. Jones, D. Roşu, and C. Roşu, "CPU reservations and time constraints: efficient, predictable scheduling of independent activities," in *Proc. SOSP*. ACM, 1997.

[25] S. Craciunas, C. 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.

[26] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Trans. Comput.*, vol. 44, no. 1, pp. 73–91, 1995.

[27] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *Journal of Real-Time Systems*, vol. 1, 1989.

[28] 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.

[29] G. Buttazzo and P. Gai, "Efficient implementation of an EDF scheduler for small embedded systems," in *Proc. OSPERT*, 2006.

[30] IBM Corp., "TuningFork Visualization Tool for Real-Time Systems," http://www.alphaworks.ibm.com/tech/tuningfork.

[31] S. Craciunas, C. Kirsch, H. Payer, A. Sokolova, H. Stadler, and R. Staudinger, "A compacting real-time memory management system," in *Proc. ATC*. USENIX, 2008.

[32] S. Craciunas, C. Kirsch, H. Röck, and R. Trummer, "The JAviator: A high-payload quadrotor UAV with high-level programming capabilities," in *Proc. GNC*. AIAA, 2008.