UNIVERSITÄT SALZBURG

Real-Time Scheduling for Workload-oriented Programming

Silviu S. Craciunas Christoph M. Kirsch Harald Röck
Ana Sokolova

Technical Report 2008-02 September 2008

Department of Computer Sciences

Jakob-Haringer-Straße 2
5020 Salzburg
Austria
www.cosy.sbg.ac.at

Technical Report Series

Real-Time Scheduling for Workload-oriented Programming*

Silviu S. Craciunas Christoph M. Kirsch Harald Röck Ana Sokolova

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

Abstract. Workload-oriented programming is a design methodology for specifying throughput and latency of real-time software processes on the level of individual process actions. The key programming abstraction is that the workload involved in executing a process action such as a system or procedure call fully determines the action's response time, independently of any previous or concurrent actions. The model thus enables sequential and concurrent real-time process composition while maintaining each action's workload-determined real-time behavior. We introduce a process model and an EDF-based scheduler as foundation for workload-oriented programming along with a prototypical implementation and experimental results. We show that the scheduler can effectively manage in constant time any number of processes up to available memory while maintaining throughput and latency of individual process actions within a bounded range.

1 Introduction

Real-time software processes typically process quantifiable amounts of data under known temporal application requirements and resource constraints. Application requirements are, for example, the usually throughput-oriented rates at which video frames in an MPEG encoder must be processed, or the mostly latency-oriented rates at which sensor data in a control system must be handled. Resource constraints might be the maximum rate at which memory can be allocated or at which data can be written to a harddisk. Both application requirements and resource constraints in turn can often directly be related to the workload, in particular, the amount of involved data. For example, a real-time process that compresses video frames usually needs to process a given number of frames within some finite response time. Similarly, resource performance may be characterized by the execution time needed to process a given number of frames. If the execution time does not depend on any other parameters than the workload, because of the nature of the involved resources or the limited range of workloads, we speak of a compositional action of the process. The resulting resource utilization is then fully characterized by the ratio between execution and

* Supported by a 2007 IBM Faculty Award, the EU ArtistDesign Network of Excellence on Embedded Systems Design, and the Austrian Science Fund No. P18913-N15.

response times, and leaves easy-to-check room for compositional actions of other, concurrent processes. Workload-oriented programming is a design methodology for composing real-time processes while maintaining their actions' individual workload-determined response times provided the resource utilization stays below the maximum resource capacity.

We introduce a process model, a utilization-based schedulability test, and an EDF-based scheduling algorithm as foundation for workload-oriented programming along with a prototypical implementation and experimental results. The test and algorithm have been designed with a focus on reducing runtime overhead and improving predictability by trading off schedulability precision (test) as well as system utilization (algorithm) for less time complexity of managing process admission and more predictable administrative overhead of making scheduling decisions, respectively. In the experiments, we show that our prototype can effectively admit and schedule in constant time any number of real-time processes up to available memory while guaranteeing workload-oriented throughput and latency of individual process actions within a bounded range. This work is the second step in an ongoing effort of building a compositional real-time operating system called Tiptoe [8]. As first step, we have already developed a real-time memory management algorithm called compact-fit [9], which will be used by user processes running on Tiptoe but also by the Tiptoe kernel itself. Tiptoe processes are meant to be implemented using workload-oriented programming and will be managed by the scheduling scheme proposed here.

The structure of the rest of the paper is as follows. We start by introducing workload-oriented programming in Section 2 followed by a discussion of related work in Section 3. We then describe the process model and a theoretical schedulability result in Section 4, which shows correctness of our scheduling algorithm presented in Section 5. In Section 6, we analyze the algorithm's complexity under different choices of data structures. The results of our experiments are shown in Section 7. Section 8 gathers the conclusions.

2 Programming Model

For an example of workload-oriented programming consider Listing 1.1, which shows the pseudo-code implementation of a real-time software process P. The process reads a video stream from a network connection, compresses it, and finally stores it on disk, all in real time. More specifically, P periodically adapts the frame rate, allocates memory to store new frames prior to receiving them from the network connection, then compresses the frames, writes the result to disk, and finally deallocates the previously allocated memory to prepare for the next iteration. As a side note, allocating and deallocating memory periodically may appear wasteful to real-time programmers but raises the level of programming abstraction because it avoids the need for managing otherwise statically allocated memory. Our compact-fit memory management algorithm bounds fragmentation and guarantees execution times for malloc and free that are at most linear in the size of the involved objects, independently of the memory state [9].

```
1 loop {
2   int number_of_frames = determine_rate();
3
4   allocate_memory(number_of_frames);
5   read_from_network(number_of_frames);
6
7   compress_data(number_of_frames);
8
9   write_to_disk(number_of_frames);
10  deallocate_memory(number_of_frames);
11 } until (done);
```

Listing 1.1. A real-time software process P

Note that, with an adequate real-time garbage collector, deallocating memory explicitly in P could be avoided, thus raising the level of abstraction even more.

The `determine_rate` and `compress_data` procedures implement process functionality and are therefore referred to as process code. The other procedures are system code since they provide system functionality and may involve other resources than the CPU such as network and disk devices. In our process model, we call an invocation of process or system code an action of the invoking process. An action has an optional workload parameter, which describes the workload involved in executing the action. In the example, P has five actions with the same workload parameter, which specifies the number of frames to be handled. The parameter may be omitted in actions that only involve process code if the action's time complexity is constant or unknown. The `determine_rate` action does not have a workload parameter because the action always executes in constant time. For unknown complexity, an omitted workload parameter implicitly represents CPU time, see below for more details.

For each action, there are two discrete functions, f_R and f_E, which characterize the action's performance in terms of its workload parameter. Figure 1 shows example functions for the `allocate_memory` action.

Response Time The response-time (RT) function $f_R : \mathbb{N} \to \mathbb{Q}^+$ characterizes the action's response time bound for a given workload, independently of any previous or concurrent actions. Here, f_R is a linear function, which states that allocating memory, e.g., for 24 frames, may take up to 100ms, even when interrupted by other concurrent processes. RT-functions do not have to be linear or even monotonically increasing, but might be at least the latter in most cases. RT-functions map any workload to a non-zero, positive bound of which the smallest is the action's intrinsic response delay d_R. In the example, $d_R = 4$ms since $f_R(w) \geq 4$ms for all $w \in \mathbb{N}$, which means that allocating memory may take at least 4ms on any workload.

RT-functions help trading off throughput and latency. For example, if memory is allocated latency-oriented, say, just for a single frame, the response time

is only at most 8ms but the resulting allocation rate merely provides enough memory for 125 frames per second (fps) if the action were invoked repeatedly as fast as possible. If memory is allocated throughput-oriented, say, for 24 frames at once, there could be enough memory for at least 240fps because d_R plays a smaller role, but the action's response time would also increase to 100ms in the worst case.

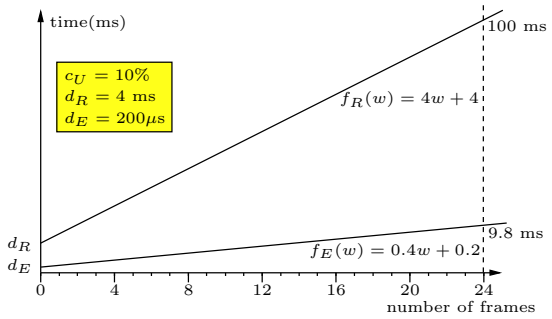


Fig. 1. Timing of the `allocate_memory` action

Because of the non-branching control flow of P, we can derive, from the individual actions' RT-functions, an exact RT-function f_P for P, which is simply the sum of its actions' RT-functions. For example, P can process 24fps if 24 frames are handled in one iteration and all actions other than the `allocate_memory` action take together no more than 900ms. Notice, however, that RT-functions of processes with branching control flow may in general just be approximated, or described more accurately, but only in languages richer than plain arithmetics, e.g., as in [7].

An RT-function provides an upper bound on an action's response time, i.e., the action may take less time but not more. However, in order to trade off responsiveness for determinacy, we can also see the bound as both upper and lower bound, similar to the notion of logical execution time (LET) [11]. In this case, we speak of a logical-response-time (LRT) function. The LET model can be seen as a special case of LRT-functions that map all workloads to a constant logical response time. Our scheduler implementation so far supports RT semantics only. However, an LRT extension is simple since it only involves delaying actions that would otherwise finish before their logical response times expired.

Execution Time The execution-time (ET) function $f_E : E_D \to \mathbb{Q}^+$ characterizes the action's execution time bound for workloads in the action's execution domain $E_D \subseteq \mathbb{N}$, in the absence of any concurrent actions. In the example, f_E is also a linear function with $E_D = \mathbb{N}$, which states that allocating memory, say, again for 24 frames, may take up to 9.8ms if not interrupted by any other process. Similar to RT-functions, ET-functions do not have to be linear or even monotonically

increasing but, at least, map any workload to a non-zero, positive bound of which the smallest is the action's intrinsic execution delay d_E. In the example, $d_E = 200\mu$s since $f_E(w) \geq 200\mu$s for all $w \in E_D$, which means that allocating memory may take at least 200μs on any workload, if not interrupted.

The notion of worst-case execution time (WCET) can be seen as a special case of ET-functions that map to a constant execution time bound for all workloads. In turn, this means that determining ET-functions requires parametric forms of WCET analysis, e.g., as in [3]. Moreover, not all actions may be ET-characterized, i.e., compositional, on large execution domains. For example, the temporal behavior of write accesses to harddisks is known to be unpredictable just in terms of the workload. However, even such actions may be properly ET-characterized by limiting workloads to smaller execution domains.

Utilization The ratio between f_E and f_R induces a discrete (partial) utilization function $f_U : E_D \to \mathbb{Q}_0^+$ with:

$$f_U(w) = \frac{f_E(w) - d_E}{f_R(w) - d_R}$$

assuming there is zero administrative overhead for handling concurrency. In the example, f_U is a function that maps any workloads $w \in \mathbb{N}^+$ to a constant $c_U = 0.1$ or 10% CPU utilization when allocating memory. In general, only workloads $w \in E_D$ with $0 \leq f_U(w) \leq 1$ (and ratios $0 \leq d_E/d_R \leq 1$) may be handled properly. We call the set of such workloads the action's utilization domain $U_D \subseteq E_D$. Even if $f_U(w)$ is not constant for all $w \in U_D$, there is still a minimal upper bound c_U such that $f_U(w) \leq c_U$ for all $w \in U_D$. We discuss a conservative but fast c_U-based schedulability test below.

Recall that workload parameters are omitted in actions that only involve process code but have unknown time complexity. In this case, RT-functions directly determine the resulting CPU utilization. For example, consider a process that invokes process code with an unknown execution time. Then, the RT-function translates CPU time into real time by stating that, e.g., 10ms CPU time may take up to 100ms real time. The result is 10% CPU utilization since the ET-function is simply the identity function from CPU time to real time.

Programming Styles Response-time and execution-time functions characterize application requirements and resource constraints, respectively, while utilization functions determine resource utilization and thus process relevance, or inversely, remaining resource capacity and processing capabilities. Depending on which two functions are given the third follows and requires appropriate validation. For example, "system-driven programming" of f_R and f_E creates utilization bounded by f_U, "platform-driven programming" of f_E and f_U determines relevant real-time behavior f_R, and "application-driven" programming of f_R and f_U requires sufficient resource capacities f_E.

Scheduler We propose a schedulability test (Section 4) and a scheduling algorithm (Section 5) for workload-oriented programming with a focus on reducing

runtime overhead and improving predictability by trading off schedulability precision (test) as well as system utilization (algorithm) for less time complexity of managing process admission and more predictable administrative overhead of making scheduling decisions, respectively. In other words, the schedulability test may not admit a schedulable process but can be performed fast (in constant time with a fixed number of resources) and the scheduling algorithm may interrupt the system more frequently but does so predictably often and makes scheduling decisions fast (in constant time with a fixed timeline but for any number of processes).

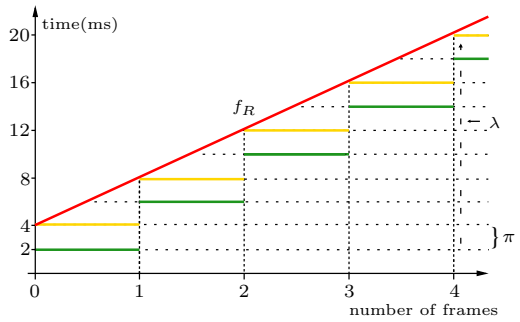


Fig. 2. The execution of `allocate_memory`

Figure 2 shows, for a workload of up to 4 frames, the scheduled execution of the `allocate_memory` action. Consider the invocation for 4 frames where the response time is 20ms. The scheduler could in principle immediately release the action with a 20ms-deadline and apply earliest-deadline-first (EDF) scheduling [15]. This strategy would, however, involve schedulability tests that depend on action invocation times and thus require analyzing process implementations and interactions. Moreover, the strategy only works with actions for which workloads are known prior to invoking the actions. We therefore propose to use the concept of virtual periodic resources [19] for action scheduling.

A virtual periodic resource R has a period π and a limit λ, which control, in our case, the execution of process actions. A process declares a finite set of such resources. Each process action uses exactly one resource declared by the process. In the example, the resource used by the `allocate_memory` action has a period of 2ms and a limit of 200μs. Upon arrival, the action is released according to one of two possible release strategies, called early and late release strategy. In the late release strategy, the release time of the action is delayed until the beginning of the next period, unless the arrival is already at the beginning. In the example the late release strategy is shown, the arrival is late, and thus the first period is already lost. In the early release strategy, the action is released immediately when it arrives, but its resource limit is decreased proportional to

the available time in the current period. All released actions are then EDF-scheduled using their resources' periods as deadlines but are prevented from executing for more than their resources' limits. When the action has exhausted the limit, it is again delayed until the beginning of the next period and so on, until the action completes. In the example, the action executes for 9 periods after being delayed in the first period and completes in the tenth period. Finally, for the invoking process to proceed to the next action, the completed action must be terminated (or finished) by the scheduler, which does that at the end of the period in which the action completed, i.e., the tenth period in the example. With the given limit, the action may have executed for up to 1.8ms, which is exactly its execution time bound for 4 frames.

The duration from the action's arrival until its termination is denoted by the scheduled-response-time (SRT) function f_S. For all $w \in E_D$, we have that:

$$f_S(w) \leq \pi - 1 + \pi \cdot \left\lceil \frac{f_E(w)}{\lambda} \right\rceil$$

if the system utilization through the most-utilized resources R of each process P is less than or equal to 100%, i.e., if

$$\sum_P \max_R \frac{\lambda}{\pi} \leq 1 \tag{1}$$

as shown in Section 4. The upper bound (yellow step function in Figure 2) occurs if the action arrives right after a new period begins, and can be reached in both release strategies. If the schedulability test holds, then the scheduled-response-time function also has a lower bound, that varies in both strategies. We have

$$f_S(w) \geq \begin{cases} \pi \cdot \left\lceil \frac{f_E(w)}{\lambda} \right\rceil & \text{, for late release} \\ \\ \pi \cdot \left\lceil \frac{f_E(w)}{\lambda} \right\rceil - \pi + 1 & \text{, for early release} \end{cases}$$

In the late release strategy, the lower bound for f_S (green step function in Figure 2) occurs if the involved action arrives exactly at the beginning of a new period. In the early release strategy, the lower bound is achieved for particular action loads, namely such that the early release actually saves one period of execution. More precisely, the lower bound is achieved if the arrival time is one time unit before a period instance, and

$$\frac{1}{\pi} \cdot \lambda \geq f_E(w) - \left\lfloor \frac{f_E(w)}{\lambda} \right\rfloor \cdot \lambda > 0.$$

The schedulability test (1) 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.

Finally, for the link back to workload-oriented programming, it is necessary to identify constraints on π and λ such that the scheduled response time $f_S(w)$

of an action's invocation on a workload $w \in U_D$ is less than or equal to its specified response time $f_R(w)$ (red function in Figure 2). With $\lambda = \pi \cdot c_U$, we only have that $f_S(w) \leq f_R(w) + \pi$ if π divides d_R evenly, i.e., $\pi \mid d_R$, and

$$0 < \pi \leq d_R - \frac{d_E}{c_U}$$

This is true even if π does not divide d_R evenly, but only for π less than or equal to half of the upper bound. The constraint is a sufficient condition, which ensures that at least the d_E portion of an action's invocation will be completed within d_R time even when the first period of the invocation is not used. In our example, the upper bound is 2ms, which would come down to, say, 1ms if d_E were increased to 300μs. Note that, with d_E approaching 400μs, i.e., 10% utilization, π would have to become zero because of the potentially unused first period. In order to have that $f_S(w) \leq f_R(w)$ actually holds, π also needs to divide the remaining response time $f_R(w) - d_R$ evenly, i.e., $\pi \mid (f_R(w) - d_R)$ or equivalently $\lambda \mid (f_E(w) - d_E)$, which is true in the example. In general, checking this constraint may be difficult, in particular, on unbounded utilization domains, but is easy in case f_R or f_E are linear functions. For example, if $f_R(w) = a_R \cdot w + d_R$, then $\pi \mid (f_R(w) - d_R)$ for all $w \in \mathbb{N}$ if and only if $\pi \mid a_R$. Other less restrictive constraints than the above might also exist but their investigation is left for future work.

Implementation We have implemented our scheduling algorithm and benchmarked it on synthetic workloads. The processes that generate the workload were run in a simulation against the algorithm for lack of a stable process management in the Tiptoe kernel. The scheduling algorithm essentially manages a deadline-ordered set of released processes (ready queue) and a release-time-ordered set of delayed processes (blocked queue), as described in Section 5.

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

Table 1. Time and space complexity per plugin

We are interested in the algorithm's time complexity, i.e., the amount of time the scheduler needs to make a scheduling decision, in terms of the number of time instants (t) the scheduler can distinguish, i.e., look into the future, and the number of processes (n) in the system. In our implementation, we have separated the scheduling algorithm from the data structures implementing the ready and blocked queues, and then developed plugins based on lists, arrays, and matrices. The algorithm's time complexity is dominated by the queue operations' time complexities since the algorithm itself is loop-free and therefore runs in constant time. Table 1 shows the algorithm's time and space complexities distinguished

by plugin. With the array and matrix plugins, the number of time instants t is fixed. The matrix-based scheduler is therefore an $O(1)$ scheduler. Section 6 provides more details and also discusses a space-optimized version of the matrix plugin based on trees.

3 Related Work

We have mentioned work related to workload-oriented programming in the previous section. In this section, we relate our process model as well as the scheduler concept and implementation to other work. Virtual periodic resources [19], which we use in our scheduler, are related to resource reservations, which were introduced in [16] as CPU capacity reserves. Follow-up work [5] within the real-time operating system Eclipse employs resource reservations (reserves) for additional resources. The process 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 [14] 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 actions in the Rialto system. Another scheduler using reservation support via fair queuing is SMART [17].

The work on the Constant Bandwidth Server [1] is highly related to ours. The differences are that only a single resource is considered and, although resource reserves are described in terms of limit and period, the resource reserve is constant per process, whereas in our model different actions within one process may have different reserves. Another point of similarity is the use of an EDF-based algorithm for scheduling. A scheduling scheme that uses the concept of a constant bandwidth server has been developed in [10] for the purpose of scheduling multi-threaded real-time and non-real-time applications running concurrently in an open system. However, there is no notion of sequentiality within a thread/process there, i.e. no counterpart of our actions. A slightly different process model is used by RBED [4], which also employs an EDF-based scheduler. Similar to our choice, each action is assigned a resource reserve via a limit and a period. Unlike in our model, the ratio of limit over period is constant per process.

A different approach to real-time scheduling with resource reserves is via the use of a resource kernel [18], providing timely, guaranteed, and enforced access to physical resources for the process. A process requests a certain amount of a resource, whose availability is then guaranteed by the kernel, and not, like in all the above cases, by the scheduler.

Finally, we compare our scheduler implementation to other work in the context of the scheduler complexity. By n we denote the number of processes. The SMART [17] scheduler's 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 [13] 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 algorithm, makes use of bitmaps to improve performance.

4 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 process model are virtual periodic resources and processes composed of actions.

4.1 Resources and Processes

Each process declares a finite set of virtual periodic resources that it uses. A virtual periodic resource is an abstract notion, although one can think of the usual resources as CPU, memory, or I/O devices, that handle workloads of a certain type, with additional periodic capacity bounds. If no confusion arises, we will say resource for virtual periodic resource. More precisely, a virtual periodic resource is a triple

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

where N is the resource name, and the pair (λ, π) is the (virtual periodic) resource capacity. Here, λ stands for limit and π for period. The limit λ specifies the maximum amount of time the resource R can be used (by a process) within the period π. We assume that in a resource capacity (λ, π), $\lambda \leq \pi$. In case the name is not important, we may just consider that a resource is a pair representing a resource capacity. We allow for an arbitrary finite set of resources denoted by $\mathcal{R}$.

A process P is a finite or infinite sequence of action invocations,

$$P = \alpha_0 \alpha_1 \alpha_2 \ldots$$

for $\alpha_i \in \text{Act}$, where $\text{Act} = \mathbb{N} \times \mathcal{R}$. An action invocation $\alpha \in \text{Act}$ 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 virtual resource R. The load of an action is computable from the ET-function f_E of the action.

Note that the notion of load simplifies the model definition, although in the implementation it is in general not known a-priori: it is known for actions using resources such as memory and I/O, but not for the CPU. By $\mathcal{P}$ we denote a finite set of processes under consideration. If no confusion arises, we will just use the term action instead of action invocation.

Resource demands of the processes are expressed via the virtual periodic resources of their actions. If for a process action invocation we have $\alpha = (l, R)$ for $R = (N, \lambda, \pi)$, then the limit λ specifies the maximum amount of time the process P can perform on the resource R within the period π, while performing the action α.

Example 1. Assume we have the following resources,

$$\mathcal{R} = \{(C, 1, 2), (M, 1, 4), (I, 1, 3)\}$$

where C stands for computation i.e. CPU, M for memory, and I for interaction or an I/O channel, and all the time units are seconds. We consider a finite process P that first does a computation for 3 seconds, then works on allocating/deallocating memory objects of size 200KB, which takes 2 seconds, then it produces output of size 100KB on an I/O device in 1 second, then again it computes, now for 2 seconds. If our timeline has unit equal to 1 second, then we can represent P in our model as a finite sequence

$$P = \alpha_0\alpha_1\alpha_2\alpha_3 = (3, (C, 1, 2))(2, (M, 1, 4))(1, (I, 1, 3))(2, (C, 1, 2)).$$

Via the resource capacity, the process P requests up to, and in the same time it promises not to use more than, one second of computation resource each two seconds, only perform up to one second memory work each four seconds, and perform every three seconds up to one second of I/O related work. Note that, although not used in this example, a process could use the same resource name with different capacity, resulting in two virtual periodic resources, i.e. one action could use the CPU with capacity $(1, 2)$, and another one with a different capacity, for example $(1, 3)$.

An example of an infinite process Q is a loop that executes the same actions as P forever. It is given by the infinite (periodic) sequence of actions $Q = \beta_0\beta_1\beta_2\ldots$, for $\beta_{4i} = (3, (C, 1, 2)), \beta_{4i+1} = (2, (M, 1, 4)), \beta_{4i+2} = (1, (I, 1, 3))$ and $\beta_{4i+3} = (2, (C, 1, 2))$. $\Diamond$

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

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

- Termination or finishing 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.
- Release time r_i is the earliest time when this action 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. $s_i = f_i - a_i$. Similarly one can define response time for any finite sequence of actions, in particular for any finite process.

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 and resource capacity if and only if there exists a schedule $\sigma : \mathbb{N} \hookrightarrow \mathcal{P}$ that respects the resource capacity, 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.

Example 2. Consider the processes P and Q from Example 1. Given the bounds:

$$\begin{array}{l} b(\alpha_0) = b(\beta_{4i}) \quad = 7, b(\alpha_1) = b(\beta_{4i+1}) = 11 \\ b(\alpha_2) = b(\beta_{4i+2}) = 5, b(\alpha_3) = b(\beta_{4i+3}) = \ 5 \end{array}$$

we have that the set $\{P, Q\}$ is schedulable with respect to these bounds and the resource capacity from Example 1. A schedule for P and an initial segment of Q, $\beta_0\beta_1\beta_2\beta_3$, that meets the bounds is presented in Figure 3. The interested reader might want to check that this schedule indeed respects the resource capacity.

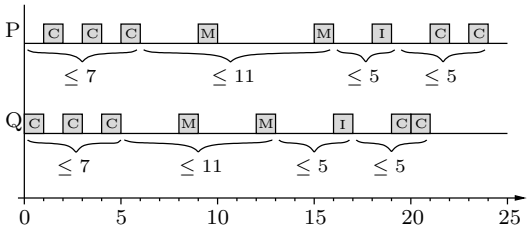


Fig. 3. Example schedule for P and Q

However, if we take $b(\alpha_0) = 3$, then regardless of the other bounds, the process P i.e. the singleton set $\{P\}$ is no longer schedulable. The reason is that the capacity of the resource used by P allows for 1 unit of computation each period of 2 units, so it is not possible to perform the workload 3 for α_0 within 3 units of time. $\Diamond$

4.3 Schedulability Result

Given a finite set $\mathcal{P} = \{P_i \mid 1 \leq i \leq n\}$ of processes with corresponding actions $\alpha_{i,j} = (l_{i,j}, R_{i,j})$ for $j \geq 0$, such that $P_i = \alpha_{i,0}\alpha_{i,1}\ldots$ we define response bounds

$$b_{i,j} = \pi_{i,j} - 1 + \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \cdot \pi_{i,j} \tag{2}$$

where $\alpha_{i,j} = (l_{i,j}, (R_{i,j}, \lambda_{i,j}, \pi_{i,j}))$ with $l_{i,j}, R_{i,j}, \lambda_{i,j}$ and $\pi_{i,j}$ being the load, the resource name, the limit and the period for the action $\alpha_{i,j}$.

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

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

$$\sum_{i \in I} \max_{j \geq 0} \frac{\lambda_{i,j}}{\pi_{i,j}} \leq 1, \tag{3}$$

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

The proof of Proposition 1 can be found in Appendix A. Here, we briefly describe the proof idea. To meet the bounds, each action of each process splits into a sequence of typed tasks, as in Lemma 1 (Appendix A), where the tasks have either (a) release time equal to the next period instance after the arrival time of the task, and duration equal to the limit (late strategy), or (b) release time equal to the arrival time, and adjusted duration so that the task does not exceed its limit nor utilization in the current period (early strategy). Hence we consider two release strategies. Moreover, the termination strategy described

above determines the arrival time of the first task of any action, namely it is the end of the period in which the previous action completed. This set of tasks is schedulable according to Lemma 1 if the utilization test (3) is satisfied.

Hence, we test whether the sum of the utilization each process achieves when running its "most expensive" action is less than 1. The test is finite even though the processes may be infinite because each process uses a finite set of resources. In addition, the test is computable even if the actual loads of the actions are unknown, as it is often the case in practice.

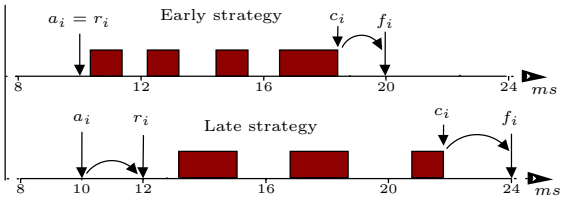


Fig. 4. Scheduling an action $\alpha = (5s, (2s, 4s))$

Example 3. Figure 4 presents the scheduling of an action 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 this situation the scheduled response time in the early release strategy is one period shorter than in the late release strategy. Moreover, using this graphical representation, it is easy to understand the correctness of the bounds on the scheduled-response-time functions from Section 2.$\Diamond$

5 Scheduling Algorithm

In this section we describe the scheduling algorithm. At any relevant time t, our system state is determined by the state of each process. A process may be running, blocked or ready. By Running, Ready, and Blocked we denote the current sets of running, ready, and blocked processes. 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,

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

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

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

where i is the process identifier, j stores the identifier of its current action $\alpha_{i,j}$, d_i is the current deadline, r_i is the next release time, l_i is the current load, and λ_i is the current limit. The scheduler also uses a time value t_s which stores the previous time instant at which it was invoked.

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

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

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 supply.

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

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, $\lambda_i \leftarrow \lambda_i - (t - t_s)$. The current load l_i becomes $l_{i,j+1}$.

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,

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

and P_i is then moved to Blocked.

The second strategy, called early release strategy, calculates the new resource limit for P_i, as

$$\lambda_i \leftarrow \left\lfloor (k\pi_{i,j+1} - t) \cdot \frac{\lambda_{i,j+1}}{\pi_{i,j+1}} \right\rfloor .$$

Given the new limit, we set the deadline to the end of the current period

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

and move P_i to Ready. The early release strategy is an optimization of the late release strategy as it results in higher utilization of the system depending on the given workload.

Limit P_i uses all of the resource limit for the resource $R_{i,j}$ given by λ_i. In this case we have $l_i \leftarrow l_i - (t - t_s)$, and

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

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 process load and limit,

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

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_l : the time at which the new running process P_i completes its entire work needed for its current action without preemption, i.e. $t_l = t + l_i$.
- t_λ : the time at which P_i consumes its limit for the current period of the resource R_i, i.e. $t_\lambda = t + \lambda_i$.
- 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 $t = \min(t_l, t_\lambda, t_r)$ at which point it gives control back to the scheduling algorithm.

The complexity of the scheduling algorithm amounts to the complexity of the plugins that manage the Ready and Blocked sets, the rest of the algorithm is constant time.

6 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 [8], supports doubly-linked lists (Section 6.1), time-slot arrays of FIFO queues (Section 6.2), and a time-slot matrix of FIFO queues (Section 6.3).

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.

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

	list	array	matrix
ordered-insert	$O(n)$	$\Theta(\log(t))$	$\Theta(\log(t))$
select-first	$\Theta(1)$	$O(\log(t))$	$O(\log(t))$
release	$O(n^2)$	$O(\log(t) + n \cdot \log(t))$	$\Theta(t)$

Table 2. Time complexity of the queue operations

6.1 Process List

The list plugin uses ordered doubly-linked lists for Ready, which is sorted by deadline, and Blocked, which is sorted by release time. 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 the first k processes in Blocked by moving them to Ready, which contains m processes, takes $k \cdot m$ steps. The upper bound of k and m is n, and therefore the complexity is $O(n^2)$. Advantages of this data structure are low memory usage (only two pointers per process) and no limitation on the resolution of the timeline.

6.2 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 by 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[1], 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. 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 $k \cdot \log_s(t)$. Additionally, the bit which indicates that there are processes at time instant t_i in Blocked is cleared in $\log_s(t)$ steps. As a result 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.

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

6.3 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 is 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, for example, we find the first set 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.

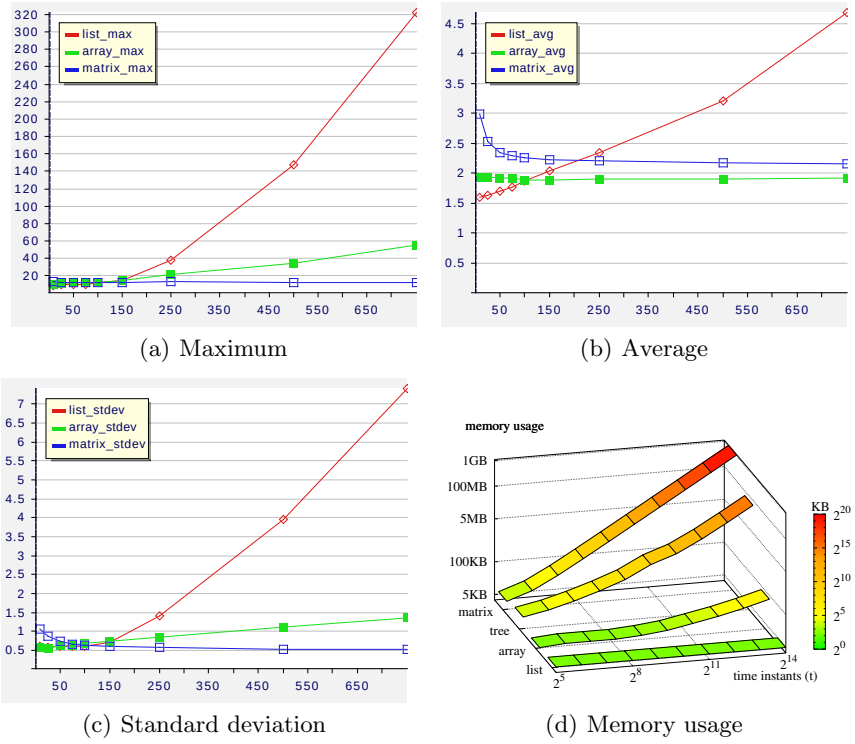
In addition to the static limitation for the number of time instants, the major disadvantage of the matrix plugin is the high memory usage. To distinguish t time instants the FIFO matrix uses t^2 pointers. Additionally, the meta-data consists of $(s/(s-1)) \cdot t \cdot (t-1)$ bits for the bit matrix 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 [2]. 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, for example 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.

7 Experiments and Results

In this section we present results of different experiments with our prototype implementation, running on a 2GHz AMD64 machine with 4GB of memory.



(a) Maximum (b) Average

(c) Standard deviation (d) Memory usage

Fig. 5. Scheduler time and space overhead

7.1 Scheduler Overhead

In order to measure scheduler execution times, we schedule 9 different sets of 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 [12]. From a sample of one million invocations we calculate the maximum (Figure 5(a)), the average (Figure 5(b)), and the standard deviation (Figure 5(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. Additionally, Figure 6 depicts histograms of the scheduler execution times for 750 processes. The B+ tree plugin performs the same as the matrix plugin up to 140ns, and is therefore not shown.

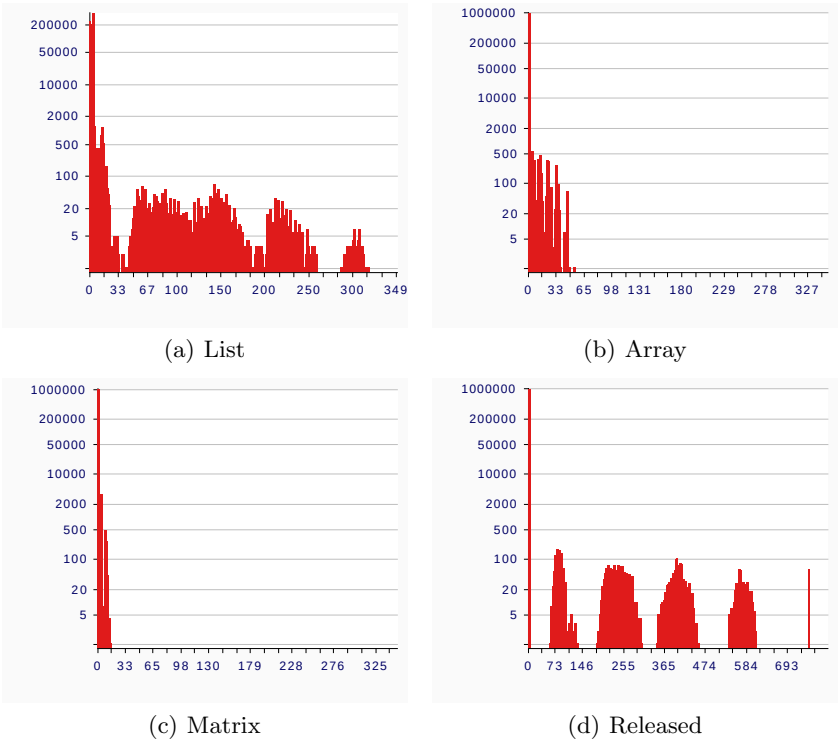
The execution time measurements conform to the complexity bounds from Section 6. For a low number of processes (less than 150), all plugins perform similarly and the scheduler needs at most 20 microseconds. On average (Figure 5(b)), for a low number of processes (up to 100) the list plugin is the fastest. Interestingly, on average the array plugin is always faster than the matrix plugin, even for a high number of processes.

The variability (jitter) of the scheduler execution can be expressed in terms of its standard deviation, depicted in Figure 5(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 5(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 5(d). The memory usage of just the B+ tree is 370KB, in comparison 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 5(d) are calculated from theoretical bounds. However, our experiments confirm the results.

Figures 6(a), 6(b), and 6(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 6(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 6(a), are closely related to the histogram of the number of processes released during the experiment (Figure 6(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 6(a) and



(a) List (b) Array

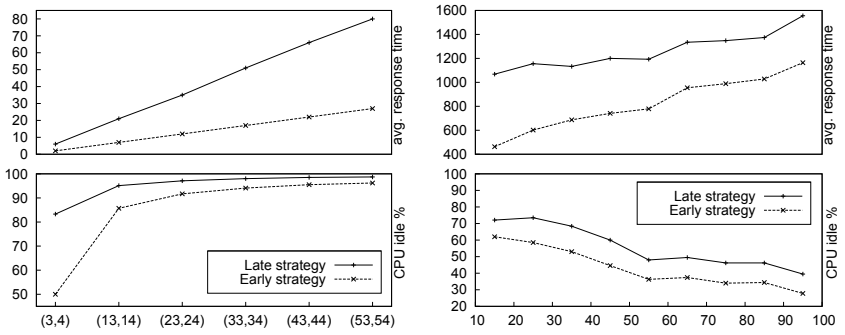
(c) Matrix (d) Released

Fig. 6. Execution times histograms

Figure 6(d) indicates that the release operation dominates the execution of the scheduler for the experiment with 750 processes.

7.2 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 (Figure 7(a), top), and improves average response times for an increasing number of processes with a random distribution of loads, limits, and periods (Figure 7(b), top). In both experiments, response times are in ms, 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 Figure 7(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



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

Fig. 7. Release strategies comparisons

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.

8 Conclusions

We have presented the workload-oriented programming model and, as foundation, an adequate process model with a fast EDF-based scheduler and an inexpensive sufficient schedulability test. Furthermore, we have designed and implemented a constant-time scheduling algorithm, analyzed its time and space complexity, and confirmed the theoretical results with a series of experiments.

References

1. L. Abeni and G. Buttazzo. Resource reservation in dynamic real-time systems. *Journal of Real-Time Systems*, 27(2):123–167, 2004.
2. R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.
3. G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *Proc. 25th Workshop on Real-Time Programming*, pages 15–19, 2000.
4. 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'03*, pages 396–408. IEEE Computer Society.
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'97*, pages 63–73. ACM.
6. G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

7. S. Chakraborty and L. Thiele. A new task model for streaming applications and its schedulability analysis. In *Proc. DATE'05*, pages 486–491. IEEE Computer Society.
8. S. S. Craciunas, C. M. Kirsch, H. Payer, H. Röck, A. Sokolova, H. Stadler, and R. Staudinger. The Tiptoe system. http://tiptoe.cs.uni-salzburg.at, 2007.
9. 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'08. To appear.*
10. 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*, 16(2-3):155–185, 1999.
11. T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proc of the IEEE*, 91(1):84–99, 2003.
12. IBM Corp. TuningFork Visualization Tool for Real-Time Systems. http://www.alphaworks.ibm.com/tech/tuningfork.
13. M. B. Jones, D. Roşu, and C. Roşu. CPU reservations and time constraints: efficient, predictable scheduling of independent activities. In *Proc. SOSP'97*, pages 198–211. ACM.
14. M.B. Jones, P.J. Leach, R.P. Draves, and J.S. Barrera. Modular real-time resource management in the Rialto operating system. In *Proc. HOTOS'95*, pages 12–18. IEEE Computer Society.
15. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.
16. C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *International Conference on Multimedia Computing and Systems'94*, pages 90–99.
17. J. Nieh and M. S. Lam. The design, implementation and evaluation of SMART: a scheduler for multimedia applications. In *Proc. SOSP'97*, pages 184–197. ACM.
18. S. Oikawa and R. Rajkumar. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Proc. RTAS'99*, pages 111–120. IEEE Computer Society.
19. I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proc. RTSS'03*, pages 2–14. IEEE Computer Society.

A Schedulability Proof

In order to prove Proposition 1, we first isolate a more essential schedulability property in the following section.

A.1 Typed EDF

We describe a schedulability test for a particular dynamic EDF 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. 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 a 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. On the other hand, 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. [6]. 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, c.f. Figure 8. Note that $t < d$ and t is a release time of some task.

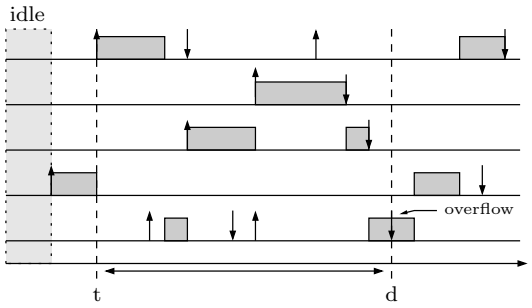


Fig. 8. Deadline miss and utilization

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.

A.2 Proof of Proposition 1

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 to 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 $\{\tau_{i,k} \mid i \in I, k \geq 0\}$ 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

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

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. Actually in average we may that way achieve better response times than the bounds provide and 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

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

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,

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

which completes the proof. □