I. PRELIMINARIES

Time utility function (TUF) scheduling goes beyond the starttime-deadline notion to express tasks’ temporal constraints. The goal of the scheduler is to maximize system utility under the assumption that tasks aggregate a given amount of utility to the system as a function of when they execute. Work available in the literature include the characterization of time utility functions to express applications’ requirements [5], [7], and several proposals of scheduling algorithms for increased utility accrual [8], [9], [3] which differ in their supported utility functions, overhead, etc.

Most work on TUF scheduling, including all previously mentioned scheduling algorithms, do not consider preemption. Many simple embedded systems do not support preemption, and some applications, like network packet scheduling, are inherently non-preemptive. However, if allowed, preemption opens up possibility for more efficient resource utilization, and being able to explore it is advantageous. We believe that the scarcity of work considering preemption reflects the lack of understanding of the impact of preemption on the utility accrual. Moreover, preemptive scheduling imposes extra challenges such as context switch overhead, which may lead to poor and inefficient resource utilization.

II. THE RELATION BETWEEN PREEMPTION AND UTILITY ACCRUAL

To the best of our knowledge, preemptive TUF scheduling has been addressed only in [6] and [2]. The work in [6] proposes a TUF preemptive scheduling algorithm assuming that the time of completion of a task defines the utility accrual to the system. A previous version of the work had focused on network packet scheduling, where this assumption holds, but the problem is inherently non-preemptive. However, as we will see later in this section, not all applications accrue utility to the system as a function of the completion time. Therefore, this model does not generalize the expression of the impact of preemption on the accrued utility.

The work in [2] proposes a preemptive TUF based scheduler which assumes that each instruction of a job’s execution accrues utility to the system as a function of the moment of execution. The utility of a job is, then, the integral of the utility function within the time intervals this job executes (see figure 1). This figure depicts the schedule of a job which executes in the time intervals \[ t_1, t_2 \] and \[ t_3, t_4 \], and the corresponding utility accrual. We believe that this task model has some flaws in expressing the utility accrual of applications as a function of time. For example, the authors of [2] use as motivating example a tracking system which verifies whether objects cross a certain boundary and intercepts them. This system reads the coordinates of the objects from registers that are periodically updated by a sensory system, processes some data, and generates as output the interception point based on trajectory pattern and speed. The utility of this application depends on the age of the sensed position when applying the interception (the older the data, the less accurate the position of the object), and the time of the output (early or late action will fail to intercept the object). Therefore, not all instructions of the code alter the utility that the application accrues to the system.

Let us analyze the timeliness requirements of other 2 applications for a better understanding of the impact of preemption on system utility: multimedia and control. Video decoding and playout require strictly periodic frame display for maximum...
The basic timing parameters of control tasks are shown in figure 2. Control tasks are released (e.g., inserted into the ready queue of the real-time operating system) periodically at times \( r_k \), and \( r_{k+1} - r_k = p \), where \( p \) is the period of the controller. Due to preemption from other tasks in the system, the actual start of the task may be delayed for some time \( L_s(k) \). This is called the sampling latency of the controller. A dynamic scheduling policy will introduce variations in this sampling latency across intervals. These variations are called sampling jitter. The maximum sampling jitter is quantified by the difference between the maximum and minimum sampling latencies in all task instances, thus \( J_{s \text{MAX}}^k = L_{s \text{MAX}}^k - L_{s \text{MIN}}^k \).

The sampling interval latency \( h_k \) is the interval of time between two consecutive samplings \( I_k \), thus \( h_k = I_{k+1} - I_k \). The nominal sampling interval is such that \( h_k = p \). Jitter in the sampling latency will of course also introduce jitter in the sampling interval latency, called sampling interval jitter. The maximum sampling interval jitter is \( J_{s \text{MAX}}^k = I_{s \text{MAX}}^k + L_{s \text{MIN}}^k \).

After some computation time and possibly further preemption from other tasks, the controller will actuate the control signal (or control output) at time \( O_k \). The delay from the sampling to the actuation is the input-output latency \( L_{IO(k)} = O_k - I_k \). Varying execution times or task scheduling preemptions will introduce variations in this interval. The maximum input-output jitter is quantified by the difference between the maximum and minimum input-output sampling latencies in all task instances, thus \( J_{IO \text{MAX}}^k = L_{IO \text{MAX}}^k - L_{IO \text{MIN}}^k \).

Basic control theory assumes all latencies to be zero, which also implies that all jitters are zero, for optimum control performance (maximum utility). Latencies and jitters are tolerable, albeit at lower utility. In this application, preemptions may impact on the utility accrual due to input-output latency.

III. HOW TO EXPRESS THE RELATION BETWEEN PREEMPTION AND UTILITY ACCRUAL IN TASK MODELS?

One of the open problems is which instructions of a task contribute to the utility accrual. The examples and observations of the previous section lead us to conclude that the utility that an application accrues to the system varies as a function of the moment of I/O operations. The internal computational state of an application is not visible to the system, and hence, should not alter the utility accrual.

Another problem is the task model abstraction which is necessary to express the utility accrual of applications. We believe that a potential solution is to extend task models with the information of which instructions of a task accrue utility to the system. Such an extension to real-time task models provides for the generalization of the task models proposed in [2] and [6]. A task may express that only one instant of the execution defines the utility accrual, like the moment completion as proposed in [6], and that every instruction accrues utility, as the model in [2]. We wonder which other application examples back up our proposal, and whether a designer can define all points of utility accrual for every application.

REFERENCES


