On the Expressiveness of Fixed-Priority Scheduling Contexts for Mixed-Criticality Scheduling

Marcus Völp*†
*School of Computer Science, Logical Systems Lab
Carnegie Mellon University
Pittsburgh, PA, USA
mvoelp@cs.cmu.edu

Adam Lackorzynski†, Hermann Härtig†
†Institute for Systems Architecture, Operating Systems Group
Technische Universität Dresden
Dresden, Germany
{voelp, adam, haertig}@os.inf.tu-dresden.de

***Abstract*—Scheduling contexts allow flattening hierarchical schedules in virtualized mixed-criticality setups. However, their expressiveness in terms of supported higher-level scheduling algorithms is not yet well understood. This paper makes a first step in this direction by investigating how recently proposed mixed-criticality algorithms can be mapped to fixed-priority scheduling contexts and how scheduling contexts can be extended to support these algorithms. We found that although the initial implementation of scheduling contexts was rather limited, a few practically feasible extensions broadened their applicability to all investigated algorithms.**

I. Introduction

In 2005, Steinberg, Wolter and Härtig [1] introduced scheduling contexts as an elegant and simple way of implementing priority inheritance in microkernel-based systems. In these systems the majority of resources are threads executing in application-level servers. The basic idea is to schedule time quanta (described through scheduling contexts) instead of the threads or jobs that correspond to them. This way, inheritance is simply a matter of activating the recipient thread whenever the quanta is selected.

The primary purpose of scheduling contexts was to improve imprecise [2] and quality-assuring scheduling [3]. However, we have seen examples that make use of scheduling contexts in a much more general way, even when not donating them to other threads. For example, Lackorzynski et al. [4] demonstrated the flattening of hierarchical mixed-criticality schedules in virtualized guest operating systems by exporting part of the internal task structure to the hypervisor and by assigning guests multiple scheduling contexts to choose from. But how general are scheduling contexts and how can they be extended to become more general?

As a first step in this direction, this paper investigates how mixed-criticality scheduling can be mapped to a scheduling-context-based fixed-priority scheduler in the hypervisor.

Mixed-criticality scheduling [5] seeks to consolidate tasks of different importance (or criticality) into the same system. Naturally, because tasks of higher criticality may cause more severe damage when late, their analysis is taken more seriously and results in more pessimistic worst-case execution time (WCET) estimates. Mixed-criticality scheduling is about granting all tasks (lower and higher criticality) their optimistic estimates while guaranteeing the completion of higher criticality tasks in the exceptional case where one of the more optimistic WCET estimates ceases to hold.

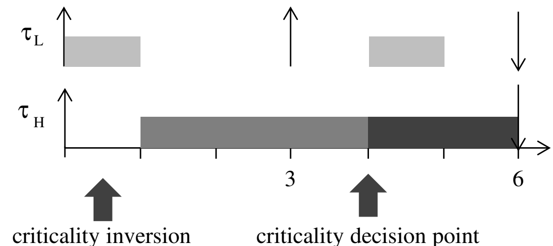

Fig. 1. Schedule of $\tau_L = (LO, 3, 3, (1, -)^T)$ and $\tau_H = (HI, 6, 6, (3, 2)^T)$. Shown is the criticality inversion of $\tau_{LO,1}$ and the criticality decision point after $\tau_{HI,1}$ received 3 time units.

As a side-effect of guaranteeing up to the optimistic WCET estimates for all tasks in case all higher criticality jobs complete within low bounds, a particularly puzzling situation called *criticality inversion* may occur. Figure 1 gives an example of such a situation for the two tasks τ_L and τ_H. Here, and in the following, we denote the release of a job with an upward arrow and its absolute deadline with a downward arrow. Darker colors are used to mark the *excess budget* of a task, that is the difference between the WCET estimate for the higher criticality level and for the respective next lower. If we had given τ_H priority over the first job of τ_L, the low criticality job could miss its deadline if τ_H executes longer than two time units. Latest at time 4, after τ_H received 3 time units, we know whether τ_H exceeds its low WCET estimate. If so, the scheduler drops the second job of τ_L and relocates its resources to τ_H in order to guarantee the completion of high criticality tasks in all situations. Otherwise, if τ_H stops within the low bounds, sufficient time remains to complete $\tau_{LO,2}$. We call the point in time after which τ_H received its low budget a *criticality decision point* of τ_H.

The contribution of this paper is an analysis of situations like the one in Figure 1 to identify whether and how mixed-criticality schedulers can be mapped to a configuration of scheduling contexts. More precisely, we shall look at such mappings for the mixed-criticality schedulers: criticality monotonic (CrMPO) [5], [6], own-criticality based priority (OCBP) [5], [6] and several static mixed criticality variants (SMC-*) [7], adaptive mixed criticality (AMC-*) [7], and earliest deadline first with virtual deadlines (EDF-VD) [8].

After formally introducing mixed-criticality tasksets, their feasibility criterion and scheduling contexts, we exemplify how such a mapping works by reciting some of the results from flattening. In Section III, we return to the question how to map the above schedulers to a fixed-priority scheduler with multiple scheduling contexts per task. Section IV reviews related attempts. Section V summarizes what we achieved and shows directions where to go from here.

II. Mixed Criticality, Scheduling Contexts and Flattening

A. Mixed Criticality

Although mixed-criticality scheduling is not limited to sporadic tasks, let us focus in this paper on this specific type of tasksets.

Let $l_i \in L$ be the criticality level of the sporadic task τ_i drawn from the totally ordered set of criticality levels L. We characterize τ_i by the tuple $\tau_i = (l_i, \delta_i, P_i, C_i)$ where δ_i is the deadline relative to the release $r_{i,j}$ of τ_i's current job $\tau_{i,j}$ and P_i is the minimal interrelease time. We assume constrained task sets, that is, $\delta_i \leq P_i$. The vector C_i denotes for each criticality level l the WCET estimate $C_i(l)$ at this level. We assume WCET estimates are monotonically increasing with increasing criticality level and constrain C_i by $C_i(l) \leq C_i(h)$ for every $l \leq h$. We generally assume that schedulers enforce the execution budgets they grant. That is, once a job exceeds $C_i(l_i)$, the scheduler will reclaim all resources assigned to it (in our case CPU-time). It is therefore safe to set $C_i(h) = C_i(l_i)$ for all criticality levels $h \geq l_i$. The feasibility criterion for mixed-criticality schedulers is:

Definition 1 (MC-Schedulability): A task set T is mixed-criticality schedulable if all jobs $\tau_{i,j}$ receive $C_i(l_i)$ time units in between $r_{i,j}$ and $r_{i,j} + \delta_i$ provided all jobs of higher criticality tasks τ_h complete before $C_h(l_i)$.

From Baruah and Vestal [9] we know that sporadic task sets are MC-schedulable if they are MC-schedulable at their synchronous arrival sequence. In this sequence, the first job of all tasks arrives at time 0 and subsequent jobs follow P_i apart. Also we know from Baruah et al. [10] that OCBP is optimal among the class of fixed job-priority algorithms.

In this paper, we shall use the terms *job* and *task* (i.e., sequence of jobs) to refer to the entities considered by mixed-criticality schedulers and scheduling contexts (*SC*) and *thread* when we talk about mapping tasks and jobs to a SC-based scheduler. We will introduce SCs in the next paragraph. We shall also write $\hat{x}$ to distinguish SC parameters from task parameters, which we denote by simple variables x. We assume that threads signal completion after they finish executing a job and that they then wait for a signal indicating the release of the next job.

B. Scheduling Contexts

In the following we introduce scheduling contexts (SC) and exemplify their use in previous work [4]. SCs are basic operating system primitives that are used for driving scheduling decisions. Traditionally, operating systems keep scheduling information (such as a thread's priority or budget) and all other thread-specific information (such as the thread's user-level register content) in the same data structure called thread control block (TCB). SC-based systems refactor TCBs into two data structures: the scheduling context (SC), which keeps all scheduling information plus the pointers to be linked into the ready queue, and the execution context (EC), which keeps all remaining state that is required to execute the thread.

Now by separating SCs and ECs, the first limitation of TCBs that can easily be dropped is that threads can have no more than one time quantum. For now we regard a time quantum as a guarantee of the scheduler to provide CPU time up to a given budget $\hat{C}_i$ (typically set to the task's WCET C_i) every $\hat{P}_k$ time units whenever there is no thread that is currently consuming higher prioritized time. Imprecise computations [2] and quality assurance scheduling [3] made use of this option to prioritize the mandatory work of all threads over optional parts such as filters and other video post-processing steps, which improve the result. The mechanism that is required to implement these scenarios is the ability to switch between SCs. That is, a thread executing on an SC must be able to select another SC, possibly discarding the remaining budget of the former.

A second application becomes possible by allowing also incoming signals to choose SCs. When scheduling the virtual CPUs (vCPUs) of multiple virtual machines (VM) one has to frequently activate each to preserve the impression of reactiveness although the VM may be off focus and running non-interactive background load. By assigning one SC with a small high priority time quantum and a second SC with a larger quantum but lower priority, VMs can react quickly to incoming signals activating the high priority SC and dropping down to the low priority SC for non-interactive tasks or in other situations determined by the VM-internal scheduler. Running applications of different importance in a VM turns this setup into a mixed-criticality system.

C. Flattening: Exemplifying SCs for Mixed Criticality

To obtain a deeper understanding how SCs can express mixed-criticality systems and in particular to clarify some of the properties we like to preserve when extending SC-based scheduling, let us repeat some of the results from flattening hierarchical mixed-criticality scheduling.

When considering VMs, time-slicing multiple vCPUs on one physical CPU is a typical approach to progress tasks scheduled by the VMs' internal schedulers. However, when VMs are mixed-criticality, time-slicing ceases to work. In [4], we presented an example similar to the one depicted in Figure 2(a). Assume two VMs (VM_A and VM_B) run the two low criticality tasks $\tau_L^A = (LO, 3, 3, (1, 1)^T)$ and $\tau_L'^A = (LO, 6, 6, (1, 1)^T)$ in VM_A and the high criticality task $\tau_H^B = (HI, 6, 6, (3, 4)^T)$ in VM_B on the same physical CPU. It is easy to see that there is no single time quanta such that VM_A's and VM_B's internal scheduler can guarantee that all tasks meet their deadlines. Because $C_H(LO) = 3$, we have to prioritize τ_L^A over τ_H^B and invert criticality. But then, if we assign a single budget of $\hat{C}_A = 1.5$ time units every $\hat{P}_A = 3$ (or even a budget of 1 for the first 3 time units and of 2 for the second 3), VM_B can no longer guarantee the completion of τ_H^B. However, if like in Figure 2(b) we assign two distinct time quanta to VM_A by

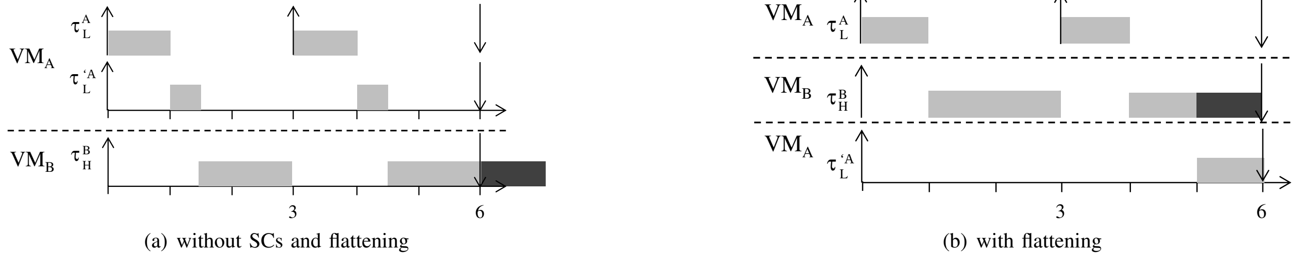

Fig. 2. Mixed-criticality schedule of the tasks $\tau_L^A = (LO, 3, 3, (1, 1)^T)$ and $\tau_L'^A = (LO, 6, 6, (1, 1)^T)$ in VM_A and $\tau_H^B = (HI, 6, 6, (3, 4)^T)$ in VM_B.

linking to it one scheduling context SC_A^1 with $\hat{C}_A^1 = 1$ every $\hat{P}_A^1 = 3$ and a second one SC_A^2 with $\hat{C}_A^2 = 1$ every $\hat{P}_A^2 = 6$, the taskset becomes MC schedulable. The priority ordering for this to work is that SC_A^1 is higher prioritized than SC_B and SC_A^2 lower than SC_B.

In Figure 2(b), the scheduler inside VM_A used the budget of SC_A^1 to run τ_L^A and SC_A^2 to run $\tau_L'^A$. However, it is also possible for VM_A to use SC_A^1 at time 3 to complete $\tau_L'^A$ and leave SC_A^2 for τ_L^A's second job. There are two points why this choice is important: First, although we will not further follow this direction in this paper, we would like to allow VMs to hide as much of their internal structure as possible. That is, VM internal schedulers should be able to give guarantees to the VM internal threads without having to expose all these threads to the hypervisor scheduler, which in fact may be different from the internal scheduler. However, unlike typical hierarchical schedulers (see e.g., Regehr and Stankovic [11] or Zhang and Burns [12]), SCs allow nested schedulers to work with more than just a single time quantum.

The second important point is that we seek to build our systems such that guarantees are robust against failures in tasks, nested schedulers or even the entire VM. As a consequence, any operation that a VM or thread may perform on its assigned SCs must not violate the timing guarantees offered to other threads. Notice, we do not extend this control to the thread as a whole because SCs provide us external control over the timing behavior of a thread without having to worry about the remainder of its state. An execution context without an SC simply will never get the CPU.

III. Mixed-Criticality Scheduling with Scheduling Contexts

We now turn our attention to the mapping of mixed criticality schedulers to SCs.

A. Criticality-Monotonic and Static Fixed Task-Priority Algorithms

The initial implementation described in Steinberg et al. [1] equipped SCs with just a budget $\hat{C}_i$, a period $\hat{P}_i$ and a priority π_i. However, they already allowed restricting their release to the point in time when both an inter-process communication message was received and $\hat{P}_i$ time units have passed since the last release. SCs were scheduled by a fixed-priority scheduler and only the single highest prioritized active SC was kept in the scheduler's ready queue.

It is easy to see that such a setting can support implicit deadline constrained sporadic tasksets (i.e., tasks with $\delta_i = P_i$) and static-priority mixed-criticality schedulers. A scheduler is static-priority if it assigns a fixed priority to each task and then relies on this priority (and the enforcement of deadlines and budgets) to ensure MC schedulability. Each task τ_i is assigned exactly[1] one SC with $\hat{C}_i = C_i(l_i)$ and $\hat{P}_i = P_i$.

Prominent examples of such a mixed-criticality scheduler are the fixed task-priority instances of the Criticality Monotonic Priority Ordering (CrMPO) [6] family. CrMPO assigns priorities such that higher criticality tasks are strictly higher prioritized than lower criticality tasks. Within the priority strips of equally critical tasks, priorities are assigned following a standard scheduling algorithm such as rate or deadline monotonic.

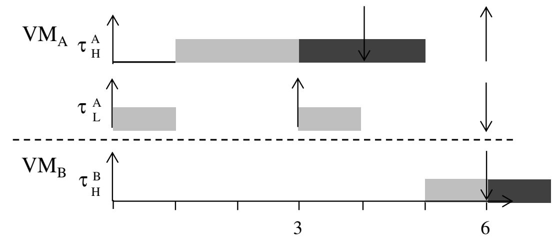

Fig. 3. Deadline overrun with side effect to task in other VM.

Standard SCs are not equipped for enforcing deadlines of constrained tasksets. For example, the taskset comprised of $\tau_H^A = (HI, 4, 6, (2, 4)^T)$, $\tau_L^A = (LO, 3, 3, (1, 1)^T)$ and $\tau_H^B = (HI, 6, 6, (1, 2))$ is MC-feasible (e.g., with $\pi_H^A > \pi_L^A > \pi_H^B$). However, a bug in τ_H^A causing a late start after its release may result in τ_H^A overrunning its deadline and in turn τ_H^B missing its deadline. Figure 3 illustrates this point.

To support constrained deadline sporadic tasks, our first extension to SCs is a deadline $\hat{\delta}_i$ up to which budgets must have been consumed to not interfere with other tasks in the way we have just seen. Because our scheduler enforces budgets by setting a timeout to the remaining budget $\hat{C}_{i,remaining}$, enforcing deadlines comes almost for free. Instead of setting this timeout to $t + \hat{C}_{i,remaining}$ when switching at time t to $\tau_{i,j}$, we set it to $\min(t + \hat{C}_{i,remaining}, r_{i,j} + \hat{\delta}_i)$ where $r_{i,j}$ is the release of $\tau_{i,j}$. The only situation where constrained

[1] In case of hierarchical scheduling, multiple tasks of the same VM with adjacent priorities and the same periods could be consolidated to the same SC. However, as already mentioned, this line of argumentation is out of the scope of this paper.

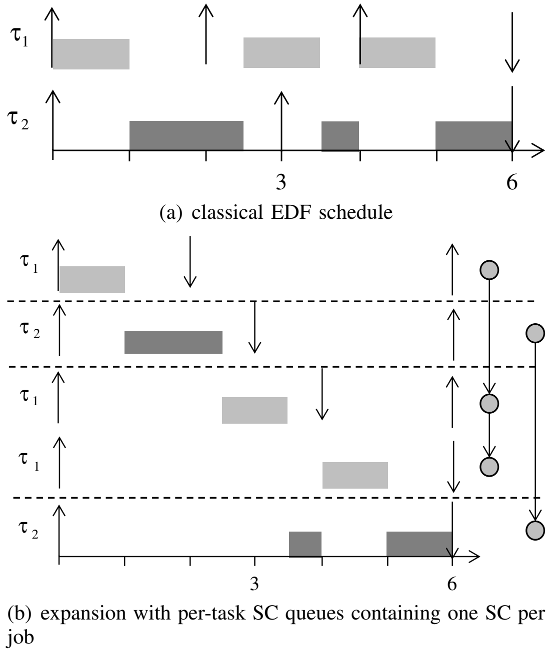

Fig. 4. Expansion of EDF schedule using SC queues and deadline-enforcing SCs

deadline enforcement causes a scheduling overhead not present in budget enforcing schedulers is when the activation signal of the next sporadic job $\tau_{i,j+1}$ arrives before $r_{i,j} + P_i$. In this case we have to set a second timer to $r_{i,j} + P_i$, which will always fire.

B. Own-Criticality Based Priority Ordering and Static Fixed Job-Priority Algorithms

The extension to deadline-enforcing SCs and a feature that was already present and required for imprecise computing [2] and quality-assuring scheduling (QAS) [3] brings us a big step closer to fixed job-priority algorithms such as own-criticality based priority ordering [6].

In QAS, the idea is to drop to a lower priority once the important part of work has completed. This gives other threads the chance of completing their important work before continuing to increase the quality of the result. The SC mapping therefore involves one SC for the mandatory part of the work, which must be completed, plus one additional SC for each optional, quality improving step. Once a thread completes its mandatory work, it steps ahead to the next SC, which in the course discards the budget of the previous SC and enables the new budget.

Although more efficient solutions exist for greedy fixed job-priority algorithms such as EDF, it is "almost" possible to use SCs and a fixed SC-priority scheduling algorithm to implement arbitrary static fixed job-priority algorithms. Let us first explain the idea and then fix the "almost" in the above statement.

Figure 4 gives an example for EDF with a classic (i.e., single-criticality) taskset. Rather than using SCs to describe a single recurring release of a task, we use it to describe the release of a single job in the hyperperiod HP of the entire taskset. A task τ_i executes $n = \frac{HP}{P_i}$ jobs in a hyperperiod. For each task, we instantiate n SCs and combine them into a list in the order of the release of these jobs. The parameters of these SCs are $\hat{P}_i = HP$ and $\hat{C}_i = C_i$ for all SCs and $\delta_{i,j} = P_i \cdot j$ for the SC representing the jth job $\tau_{i,j}$ ($1 \leq j \leq n$). We set the priority of the SCs according to the fixed job-priority algorithm. For EDF, these are $\pi_{o,p} > \pi_{q,r}$ whenever $r_{o,p} + \delta_o \leq r_{q,r} + \delta_q$ (breaking ties if necessary).

Because SCs are ordered in lists, a thread executing on the list of SCs will discard the remaining budget prior to executing the next job. Also, no thread can extend the budget received for executing a job $\tau_{i,j}$ beyond this job's deadline. However, and here comes the "almost" into play, there is so far no means of preventing a thread from immediately switching to the next SC and hence from consuming the budget of a not-yet released job. Although this is not a problem for EDF where SC priority are monotonically decreasing, it becomes an issue when allowing priorities to increase.

To enforce the use of budgets only after the release of the corresponding job, we propose to apply the same mechanism for switching to the next job that Jean Wolter introduced to cope with sporadic tasks in the first place. In addition to $\hat{P}_i$, we therefore introduce a second inter-release time: the refill time $\hat{R}_i$ plus configuration options to determine whether SCs are *queued* or *loose* (i.e., simultaneously available) and whether switching to the next SC corresponds to the release of the next sporadic job. To not confuse terminology with the task parameters, we will use $\hat{P}_i$ as before for the job-to-job minimal inter-release time and set $\hat{R}_i = HP$ to ensure the job's availability in the next hyperperiod.

Several variants of static mixed-criticality (SMC) algorithms have been proposed that can all be mapped in the above described way. For example, Vestal [5] suggested an algorithm based on Audsley's algorithm [13] to determine a MC-feasible per job priority assignment called own-criticality based priority assignment (OCBP). The basic idea is that if a job τ_i still receives sufficient time in between its release and deadline when it runs at the lowest priority and if we don't care about the order of or deadline misses of higher prioritized jobs, then we can fix this job at this priority and search for a next suitable job in the remaining set. Thereby, higher prioritized jobs τ_h are assumed to require at most $C_h(l_i)$. Lacking budget enforcement, Vestal first assumed a complete analysis of lower criticality tasks also at higher criticality levels leading to $C_i(h) > C_i(l_i)$ for $h > l_i$. Adding this additional constraint, Baruah and Burns [14] could improve on the schedulability of this algorithm. In [7], Baruah, Burns and Davis could finally show that presenting OCBP with a limited choice (deadline monotonic sorting of jobs within a criticality level and then for each level the job with the largest absolute deadline) suffices to obtain a feasible schedule.

Notice, although SC representations of all jobs in the hyperperiod are possible, other representations may be more appropriate for systems where it is feasible to support one specific class of MC-scheduling algorithms. In these systems, where the majority of all jobs follow a standard priority assignment such as EDF, a third extension of SCs will be helpful: to regard queued SCs as exceptions of one dedicated SC yielding the regular behavior. For example, if all jobs but

the third and fifth follow the EDF priority assignment, an EDF SC-scheduler could be used to schedule the dedicated SC with the exception of those jobs that have alternatives queued. That is, after finishing the first job, the scheduler awaits the second release of the dedicated SC at which time it inserts it to the ready queue based on the stored relative deadline. However, for the third release it makes this decision based on the adjusted deadline of the alternative SC.

C. Adaptive Mixed-Criticality Algorithms

So far we have only considered scheduling algorithms that rely on the relative priority ordering of SCs to guarantee MC-schedulability. However, with an appropriate monitor of thread execution times, which is trivially given in the form of the next scheduling-context signal, MC schedulers could also react to tasks exceeding their low criticality WCET estimates. This class of scheduling algorithms is called adaptive mixed-criticality algorithms [7]. Whenever a job $\tau_{i,j}$ exceeds the WCET estimate $C_i(l)$ of a criticality level l, the scheduler follows this transition from l to $l+1$ and discontinues the execution of all l-criticality tasks. To do so, with scheduling contexts, a fourth extension is required: to disable a group of scheduling contexts. Adding this extension requires an indirection to an *enabled token*, which the scheduler can toggle to disable at once all SCs that refer to this token.

Notice, to meet the demands of the MC-schedulability criterion, lower criticality jobs need never be continued once a higher criticality job has exceeded its low WCET estimate. However, it is of course desirable to return to a fully operational system as quickly as possible. The challenge is therefore in quickly re-enabling SCs once they have been disabled and it is safe to re-enable low SCs. As low WCET overruns are extremely rare events, we propose to simply deactivate but not dequeue inactive SCs from the ready queue. This way, re-enabling boils down to setting a bit in the enabled token. The additional scheduling overhead when skipping over disabled SCs is deterministic and can be considered for the high criticality WCET estimates.

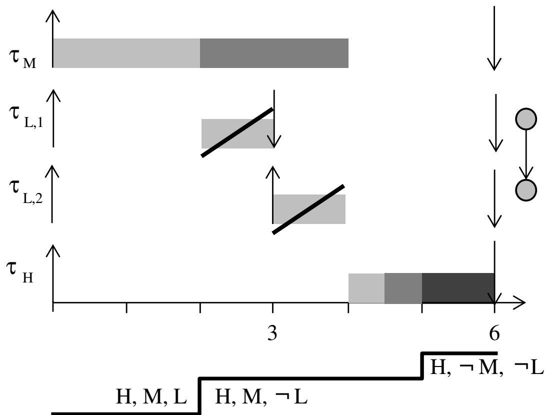

Fig. 5. Use of enabled token to disable groups of SCs.

Figure 5 shows the structure of enabled tokens in an example that is not schedulable with classical OCBP because of the consideration of $\tau_{L,1}$ for τ_H, although $\tau_{L,1}$'s execution time is hidden behind τ_M's medium-criticality excess budget. Because τ_M and τ_H require together all 6 time units, none of the low jobs $\tau_{L,1}$ and $\tau_{L,2}$ may be higher prioritized than both of the higher criticality tasks. On the other hand, because τ_M's low WCET estimate is 2 time units and τ_H's is 0.5, we cannot run both of them at a higher priority than the first low job $\tau_{L,1}$ because otherwise, if both the medium and the high criticality task stay within their low budgets, $\tau_{L,1}$'s completion could not be guaranteed. The only priority assignments that remain are therefore $\tau_{L,1}$ at an intermediate priority between τ_M and τ_H. However, OCBP considers $C_L(H) + C_M(H)$ if $\tau_{L,2}$ executes at a lower priority than τ_H or $2C_L(H) + C_M(H)$ otherwise, which both is not enough to complete τ_H at one of the two lowest priorities.

Shown at the bottom of Figure 5 is the gradual increase of the system's criticality level, which has lead to the dropping of τ_L's jobs. Also shown is the enabled token.

We suggest implementing the enabled token as a small bitfield complemented with a mask inside each SC. The mask is used to determine which bits are significant for this SC. This way, the first n criticality levels could be disabled by clearing the bit of the nth criticality level and making all tasks of criticality l significant on all bits of higher or equal criticality.

D. Earliest Deadline First — Virtual Deadlines

Realizing that fixed-job priority algorithms (such as OCBP) are limited in their schedulable utilization, Baruah, Bonifaci and D'Angelo [8] transitioned from single priority schemes to a dual priority scheme (or more generally an n priority scheme where $n = |L|$ is the number of criticality levels). For as long as no task exceeds its low WCET estimate, the schedule follows the EDF algorithm with virtual deadlines to make room for a potential criticality change. Once such a change happens, low tasks are disabled and the high tasks transition to a second priority scheme based on classical EDF.

Group enable and disable (as we have seen it in the previous section) allow us to keep both priority settings simultaneously (either explicit or implicit through deadlines interpreted by a host EDF scheduler). Once a criticality change happens, all low-criticality SCs are disabled by clearing their significant low flag and high criticality SCs are enabled by setting the formerly disabled high criticality flag in the enabled token.

IV. Related Work

Most closely related to our work, although not mixed-criticality, is part of the work by Regehr and Stankovic in the context of hierarchical scheduling of soft real-time tasks (see Table 1 in [11]). As part of their hierarchical scheduling framework they derived a map how guarantees provided by one kind of scheduler translate into the guarantees that a nested scheduler may give for its tasks. For example, a scheduler receiving $X = 50\%$ CPU share may translate this guarantee into multiple proportional share guarantees up to the point where the sum of shares Y_i is at most X. A particularly interesting case and direction of future work for this work is Surplus Fair Scheduling (SFS) [15]. Given multiple shares, SFS is able to produce a new set of shares carrying a combination of the received guarantees.

Zhang and Burns [12] investigate the schedulability of multiple earliest deadline first layers on top of a fixed-priority scheduler. Although not per se mixed criticality, Zhang's analysis can easily be extended to criticality monotonic settings where each nested EDF scheduler is responsible for tasks of one criticality level. To decide MC-schedulability, all that has to be done is to repeat the analysis for each criticality level l assuming the tasks in all higher prioritized EDF schedulers do not exceed $C_i(l)$. SCs and the consideration of slack in Zhang's analysis would even allow for occasional criticality inversions by raising selected tasks above higher criticality EDF levels provided there is enough slack in these levels.

Queued SCs generalize dual priority scheduling [16] when the transition to the next SC is not limited to thread-triggered events but for example to the release of the thread in the first place. In this case, $\hat{P}_i$ serves as trigger to switch to the higher band priority.

V. Conclusions and Future Work

As a first step in the direction of evaluating fixed SC-priority scheduling algorithms where tasks have multiple time quanta to choose from, we have investigated the mapping of five mixed-criticality algorithms. We found that although initially not all algorithms could be supported, small changes to the use and interface of SCs allowed us to map all investigated algorithms to SC-based scheduling. These changes are deadline enforcement, a separate refill period, support for sporadic jobs in the form of queued SCs and group enable/disable functionality through enabled tokens. Most of these extensions appeared recurringly as demands in discussions about Fiasco.OC's SC-based scheduling interface and are feasible to be implemented both in microkernel-based systems and elsewhere. Proper integration in a microkernel-based system regarding isolation and security characteristics are subject to evaluation.

Directions for future work include a more elaborate and formal handling of the question what scheduling guarantees can be given from a combination of fixed-priority time quanta and an investigation of further algorithms including and beyond mixed-criticality.

Acknowledgments

This work was in part funded by the DFG through the cluster of excellence "Center for Advancing Electronics Dresden", by the EU and the state Saxony through the ESF young researcher group "IMData" and by the national science foundation through grant NSF CNS-0931985.

References

[1] U. Steinberg, J. Wolter, and H. Härtig, "Fast Component Interaction for Real-Time Systems," in *17th Euromicro Conference on Real-Time Systems*, Palma de Mallorca, Spain, July 2005.

[2] J.-Y. Chung, J. W. S. Liu, and K.-J. Lin, "Scheduling periodic jobs that allow imprecise results," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1156–1173, Sep. 1990.

[3] C.-J. Hamann, J. Löser, L. Reuther, S. Schönberg, J. Wolter, and H. Härtig, "Quality Assuring Scheduling - Deploying Stochastic Behavior to Improve Resource Utilization," in *22nd IEEE Real-Time Systems Symposium (RTSS)*, London, UK, Dec. 2001.

[4] A. Lackorzyński, A. Warg, M. Völp, and H. Härtig, "Flattening hierarchical scheduling," in *Proceedings of the tenth ACM international conference on Embedded software*, ser. EMSOFT '12. New York, NY, USA: ACM, 2012, pp. 93–102.

[5] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Real-Time Systems Symposium*. Tucson, AZ, USA: IEEE, December 2007, pp. 239–243.

[6] H. Li and S. K. Baruah, "An Algorithm for Scheduling Certifiable Mixed-Criticality Sporadic Task Systems," in *RTSS*. IEEE Computer Society, 2010, pp. 183–192.

[7] S. Baruah, A. Burns, and R. Davis, "Response-time analysis for mixed criticality systems," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, 2011, pp. 34–43.

[8] S. K. Baruah, V. Bonifaci, G. D'Angelo, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie, "Mixed-criticality scheduling of sporadic task systems," in *Proceedings of the 19th European conference on Algorithms*, ser. ESA'11. Springer-Verlag, 2011, pp. 555–566.

[9] S. Baruah and S. Vestal, "Schedulability analysis of sporadic tasks with multiple criticality specifications," in *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, ser. ECRTS '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 147–155.

[10] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie, "Scheduling real-time mixed-criticality jobs," *Computers, IEEE Transactions on*, vol. 61, no. 8, pp. 1140–1152, 2012.

[11] J. Regehr and J. A. Stankovic, "HLS: A framework for composing soft real-time schedulers," in *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*. Washington, DC, USA: IEEE Computer Society, 2001.

[12] F. Zhang and A. Burns, "Analysis of hierarchical edf pre-emptive scheduling," in *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, 2007, pp. 423–434.

[13] N. Audsley, "On priority assignment in fixed priority scheduling," *Information Processing Letters*, vol. 79, no. 1, pp. 39–44, 2001.

[14] S. Baruah and A. Burns, "Implementing mixed-criticality systems in ada," in *Reliable Software Technologies — Ada-Europe'11*. Springer, 2011, pp. 174–188.

[15] A. Chandra, M. Adler, P. Goyal, and P. Shenoy, "Surplus fair scheduling: A proportional-share cpu scheduling algorithm for symmetric multiprocessors," in *4th Symposium on Operating System Design and Implementation (OSDI)*, San Diego, CA, USA, Oct. 2000, pp. 45–58.

[16] R. Davis and A. Wellings, "Dual priority scheduling," in *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*, 1995, pp. 100–109.