Support for Limited-Preemptive Fixed-Priority Scheduling – an evolutionary step still facing research challenges –

Reinder J. Bril
Technische Universiteit Eindhoven (TU/e),
The Netherlands
R.J.Bril@TUe.NL

ABSTRACT
Fixed-priority preemptive scheduling (FPPS) has been the defacto standard used in industry for real-time systems for the past decades. Unfortunately, arbitrary preemptions during execution may lead to inefficient memory use and high run-time overheads. Fixed-priority scheduling (FPS) with limited preemptions (LP-FPS) has therefore been proposed in the literature as a viable alternative to FPPS, that reduces memory requirements, reduces the cost of arbitrary preemptions, and may improve the schedulability of a task set. In this paper, we propose to incorporate support for LP-FPS in next generation operating systems for cyber-physical systems, facilitating an evolution of systems. Even for the single processor case, there are still analytical research challenges, however.

1. INTRODUCTION
Fixed-priority preemptive scheduling (FPPS) has been the defacto standard used in industry for real-time systems for the past decades. FPPS is supported by POSIX® [23], OSEK [27], and AUTOSAR [1] compliant operating systems, most commercial-off-the-shelf (COTS) real-time operating systems (RTOSs) support FPPS, a practitioner’s handbook for FPPS has been produced by CMU’s SEI [26], and many courses and automated timing analysis tools exist for FPPS. Unfortunately, arbitrary preemptions during execution may lead to inefficient memory use and high run-time overheads [21, 22], in particular in systems using cache memories, e.g. to bridge the speed gap between processors and main memory. Although fixed-priority non-preemptive scheduling (FPNS) may resolve these problems, it generally leads to reduced schedulability compared to FPPS [24].

Fixed-priority scheduling with limited preemptions (LP-FPS), such as fixed-priority scheduling with deferred preemption (FPDS) or co-operative scheduling [15, 12, 20] and fixed-priority scheduling with preemption thresholds (FPTS) [33, 31, 30, 25], have therefore been proposed as viable alternatives between the extremes of FPPS and FPNS. Compared to FPPS, limited-preemptive schemes (i) reduce memory requirements [31, 21] and (ii) reduce the cost of arbitrary preemptions [15, 12, 35, 8, 10]. Compared to both FPPS and FPNS, these schemes may significantly improve the schedulability of a task set [12, 31, 9]. Interestingly, the theoretical performance advantage that, for example, FPTS has over FPPS is magnified when cache-related preemption delays (CRPD) are taken into account for platforms that contain a cache [11]. As an example, advantages that FPDS and FPTS may bring compared to FPPS and FPNS for an application A and a platform P, (i) A may become schedulable on P in case it was not, (ii) in case A was schedulable on P, we may now be able to add additional applications, whereas that may not have been possible before, and (iii) A may become schedulable on a more cost-effective platform, e.g. a platform providing less memory or a slower CPU, or allowing prolonged intervals where the CPU is put to sleep, saving battery.

FPDS and FPTS are based on orthogonal refinements of the scheduling model for FPPS. For FPDS, each task is assumed to consist of a sequence (or a directed acyclic graph) of sub-jobs, where sub-jobs are non-preemptable. When a task is executing, it can be preempted only between consecutive sub-jobs, i.e. at so-called preemption points. With FPDS, non-preemptive regions (NPRs) are placed statically in the code of a task. Although floating NPRs are also conceivable [5], statically placed NPRs shows more schedulability improvements than floating NPRs, even when pre-emption costs are ignored [16]. In the remainder, we therefore ignore floating NPRs. For FPTS, each task has a so-called preemption threshold next to a priority. When a task is executing, it can only be preempted by tasks with a priority higher than its preemption threshold.

Because FPTS and FPDS have their own relative strengths and weaknesses [16], various papers have refined and combined these schemes [28, 25, 34, 13, 14, 36]. One of the most advanced LP-FPS schemes known today, termed FPVS in [14], assumes preemption thresholds for both sub-jobs as well as preemption points. One of its specializations, termed FPDS* in [14], assumes preemption thresholds for preemption points and non-preemptive execution of sub-jobs, eliminating arbitrary preemptions. Experiments in [14] show that FPDS* can achieve the same schedulability ratio as FPVS.

2. FROM STATE OF RESEARCH TO STATE OF PRACTICE

2.1 Applying LP-FPS
Although FPDS clearly outperforms FPTS from a theoretical perspective [16], applying FPDS in practice is still a challenge because appropriate positions for pre-emption points have to be identified and explicitly added in the code. Exceptions are applications that are OSEK/AUTOSAR [27, 1] compliant and are compositions of so-called runnables, the functional entities of the system. Runnables are mapped to operating system tasks during system synthesis, and are therefore natural, although potentially coarse grained, sub-jobs [36]. Significant advances for the general case including CRPD have been
made recently [29, 17], but the state-of-research is still limited, e.g. because either a linear code structure is assumed [17] or the most advanced CRPD approach has not been applied [29].

Applying generalizations of FPDS, such as FPVS or FPDS*, face similar challenges as FPDS.

Unlike FPDS, FPTS can be easily applied, even without any changes to the code when pre-emption thresholds can be assigned to tasks at integration time, e.g. by means of dedicated primitives or by means of code-wrappers using standard synchronization primitives.

### 2.2 Support for LP-FPS

Support for FPTS is specified by both the OSEK/AUTOSAR operating-system standards in the form of internal resources, and deployed in the automotive industry. FPTS may therefore be used for legacy code and viewed as an evolutionary, limited-preemptive successor of FPFS as the defacto standard in industry.

Initial experiments with support for FPDS in RTAI/Linux has been reported upon in [7], where efficiency and maintainability were among the main design aspects. Before implementing FPDS, a proof of concept of an implementation of FPNS in RTAI was made. Although usage of existing primitives of RTAI to achieve FPNS would be beneficial for the maintainability of the implementation, it would require at least 2 invocations of these primitives per job execution, resulting in additional overhead of at least 2 system calls per job. For efficiency reasons, a kernel modification was therefore preferred, providing explicit non-preemptive task support. For FPDS, a dedicated `rt_fpds_yield` function was implemented, because upon investigation of RTAI’s yield function `rt_task_yield` it turned out that this function is only intended for use with round-robin scheduling of tasks having equal priority.

Support for FPVS, and in particular its specialization FPDS*, using resource access primitives of an open source, AUTOSAR compliant operating system [3] has been presented in [6]. The `yield` function called at a preemption point has a complexity of $O(n)$, where $n$ is the number of tasks. To be more specific, a task with a priority $0 < p \leq n$ may make $n - p$ calls to release resources at a preemption point followed by $n - p$ calls to acquire resources at a preemption point. The solution is therefore effective, but not efficient. Dedicated primitives for either FPVS or FPDS* at the level of the API of a real-time operating system (RTOS) have not been prototyped yet, however, let alone incorporated in a commercial-off-the-shelf (COTS) RTOS.

### 3. RESEARCH CHALLENGES

Many works on LP-FPS assume a task model where task execution times are independent of pre-emption and pre-emption costs are negligible, even for the single processor case. For such a restrictive case, the problem of optimally assigning both priorities and preemption thresholds using a computational tractable method remains open for FPTS, and therefore also for its generalizations presented in [13, 14]. Although an optimal algorithm for FPDS is presented in [20], in the sense that the algorithm is guaranteed to find a schedulable combination of priority ordering and final NPR length if such a schedulable combination exists, that algorithm also assumes this restrictive model.

Papers that consider CRPD, such as [2] for FPPS, [11] for FPTS and [29] for FPDS, assume that priorities of tasks are given. For FPPS and CRPD, it has been shown in [32] that a deadline monotonic priority assignment algorithm is no longer optimal for task systems with constrained deadlines when CRPD is taken into account. Moreover, analysis covering more advanced CRPD models is not compatible with Audsley’s Optimal Priority Assignment (OPA) algorithm [4]. To the best of our knowledge, the problem of optimally assigning priorities using a computationally tractable method is still open.

We merely recall the limited state-of-research for including CRPD in the analysis for FPDS; see previous section. Using more advanced LP-FPS schemes inherit these challenges.

Given these open problems for seemingly basic cases, it is to be expected that the challenges are even more demanding for complex cases as presented in, for example, [19, 18].

### 4. CONCLUSION

We proposed to incorporate support for LP-FPS in next generation operating systems for cyber-physical systems (NGOSCPS) for cost-effectiveness reasons, e.g. to reduce memory requirements, reduce the cost of arbitrary preemptions, and improve the schedulability of task sets compared to FPPS. Such support is particularly attractive for resource-constrained systems, such as wireless sensor nodes, but may also be useful in settings where additional resources would otherwise give rise to cost increases or to limitations in functionality extensions. Moreover, LP-FPS facilitates an evolution of real-time systems and real-time system development.

Given the aim of LP-FPS, its implementation shall be efficient. Hence, although LP-FPS could be effectively implemented on top of an existing API of an RTOS, dedicated primitives are of paramount importance for cost-effectiveness.

Although these dedicated primitives at the level of an API could be used by programmers, e.g. for simple cases, we expect that these primitives will mainly be used by tools during system synthesis, i.e. dedicated support to identify appropriate pre-emption points, to add pre-emption points to the code, to set preemption thresholds for both sub-jobs and pre-emption points, and to set priorities for tasks. Given the research challenges mentioned above, accompanying full-fledged tools are not likely to be available for either FPVS or FPDS* on short notice. Extending the APIs of NGOSCPS with support for LP-FPS is still highly recommended, however, given the relative strength of these LP-FPS schemes compared to the defacto standard used in industry today, being FPFS.

### References


