Extending RTAI/Linux with Fixed-Priority Scheduling with Deferred Preemption

Mark Bergsma, Mike Holenderski, Reinder J. Bril and Johan J. Lukkien
Faculty of Computer Science and Mathematics
Technische Universiteit Eindhoven (TU/e)
Den Dolech 2, 5600 AZ Eindhoven, The Netherlands

Abstract

Fixed-Priority Scheduling with Deferred Preemption (FPDS) is a middle ground between Fixed-Priority Preemptive Scheduling and Fixed-Priority Non-preemptive Scheduling, and offers advantages with respect to context switch overhead and resource access control. In this paper we present our work on extending the real-time operating system RTAI/Linux with support for FPDS. We give an overview of possible alternatives, describe our design choices and implementation, and verify through a series of measurements that indicate that a FPDS implementation in a real-world RTOS is feasible with minimal overhead.

1 Introduction

Fixed-Priority Scheduling with Deferred Preemption (FPDS) [4–7, 9] has been proposed in the literature as an alternative to Fixed-Priority Nonpreemptive Scheduling (FPNS) and Fixed-Priority Preemptive Scheduling (FPPS) [11]. Input to FPPS and FPNS is a set of tasks of which instances (jobs) need to be scheduled. FPDS is similar to FPNS but now tasks have additional structure and consist of (ordered) subtasks. Hence, in FPDS each job consists of a sequence of subjobs; preemption is possible only between subjobs. The benefits of FPDS, derived from FPNS are (i) less context-switch overhead thanks to fewer preemptions (ii) the ability to avoid explicit resource allocation and subsequent complex resource-access protocols. The fact that subjobs are small leads to FPDS having a better response time for higher priority tasks.

FPDS was selected as a desirable scheduling mechanism for a surveillance system designed with one of our industry partners. [10] With response times found to be too long under FPNS, FPDS was considered to have the same benefits of lower context switch overhead compared to FPPS with its arbitrary preemptions.

In this paper our goal is to extend a real-time Linux version with support for FPDS. For this purpose we selected the Real-Time Application Interface (RTAI) extension to Linux [1]. RTAI is a free-software community project that extends the Linux kernel with hard real-time functionality. We aim to keep our extensions efficient with respect to overhead, and as small and non-intrusive as possible in order to facilitate future maintainability of these changes. Our contributions are the discussion of the RTAI extensions, the implementation[1] and the corresponding measurements to investigate the performance of the resulting system and the introduced overhead.

The work is further presented as follows. We start with an overview of related work and recapitulation of FPDS, followed by a summary of the design and features of RTAI. Then we analyze how FPDS should be dealt with in the context of RTAI. We present our investigation, design and proof of concept implementation of FPDS in RTAI. This result is analyzed through a series of measurements. We conclude with a summary and future work.

2 Related work

As a recapitulation [4–7,9], in FPDS a periodic task τ_i with computation time C_i is split into a number of non-preemptive sub tasks $\tau_{i,j}$ with individual computation times $C_{i,j}$. The structure of all subtasks defining an FPDS task is defined by either the programmer through the use of explicit *preemption points* in the source, or by automated tools at compile time, and can have the form of a simple ordered *sequence*, or a directed acyclic graph (DAG) of subtasks. See Figure 1

[1]This work is freely available at http://wiki.wikked.net/wiki/FPDS

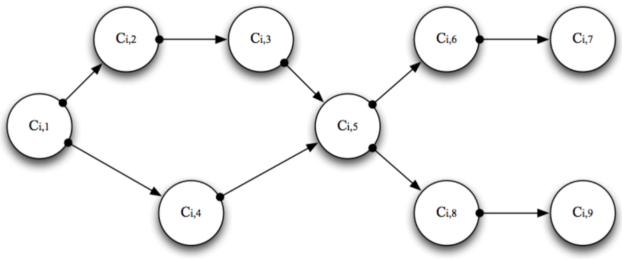

Figure 1. FPDS task with a DAG structure

for an example of the latter.

[9] presents a *rate-monotonic with delayed preemption (RMDP)* scheduling scheme. Compared to traditional rate-monotonic scheduling, RMDP reduces the number of context switches (due to strict preemption) and system calls (for locking shared data). One of the two preemption policies proposed for RMDP is *delayed preemption*, in which the computation time C_i for a task is divided into fixed size quanta c_i, with preemption of the running task delayed until the end of its current quanta. [9] provide the accompanying utilization based analysis and simulation results, and show an increased utilization of up to 8% compared to traditional rate-monotonic scheduling with context switch overheads.

Unlike [9], which introduces preemption points at fixed intervals corresponding to the quanta c_i, our approach allows to insert preemption points at arbitrary intervals, convenient for the tasks.

[3, 4] correct the existing worst-case response time analysis for FPDS, under arbitrary phasing and deadlines smaller or equal to periods. They observe that the critical instance is not limited to the first job, but that the worst case response time of task τ_i may occur for an arbitrary job within an i-level active period. They provide an exact analysis, which is not uniform (i.e. the analysis for the lowest priority task differs from the analysis for other tasks) and a pessimistic analysis, which is uniform.

The need for FPDS in industrial real-time systems is emphasized in [10], which aims at combining FPDS with reservations for exploiting the network bandwidth in a multimedia processing system from the surveillance domain, in spite of fluctuating network availability. It describes a system of one of our industry partners, monitoring a bank office. A camera monitoring the scene is equipped with an embedded processing platform running two tasks: a video task processing the raw video frames from the camera, and a network task transmitting the encoded frames over the network. The video task encodes the raw frames and analyses the content with the aim of detecting a robbery. When a robbery is detected the network task transmits the encoded frames over the network (e.g. to the PDA of a police officer). In data intensive applications, such as video processing, a context switch can be expensive: e.g. an interrupted DMA transfer may need to retransmit the data when the transfer is resumed. Currently, in order to avoid the switching overhead due to arbitrary preemption, the video task is non-preemptive. Consequently, the network task is activated only after a complete frame was processed. Often the network task cannot transmit packets at an arbitrary moment in time (e.g. due to network congestion). Employing FPDS and inserting preemption points in the video task in convenient places will activate the network task more frequently than is the case with FPNS, thus limiting the switching overhead compared to FPPS and still allowing exploitation of the available network bandwidth.

[10] also propose the notion of *optional preemption points*, allowing a task to check if a higher priority task is pending, which will preempt the current task upon the next preemption point. At an optional preemption point a task cannot know if a higher priority task will not arrive later, however if a higher priority task is already pending, then the running task may decide to adapt its execution path, and e.g. refrain from initiating a data transfer on a exclusive resource that is expensive to interrupt or restart. Optional preemption points rely on being able to check for pending tasks with low overhead, e.g. without invoking the scheduler.

3 RTAI

RTAI[2] is an extension to the Linux kernel, which enhances it with hard real-time scheduling capabilities and primitives for applications to use this. RTAI provides hard real-time guarantees alongside the standard Linux operating system by taking full control of external events generated by the hardware. It acts as a *hypervisor* between the hardware and Linux, and intercepts all hardware interrupt handling. Using the timer interrupts RTAI does its own scheduling of real-time tasks and is able to provide hard timeliness guarantees.

Although RTAI has support for multiple CPUs, we choose to ignore this capability in the remainder of this

[2]The code base used for this work is version 3.6-cv of *RTAI* [1].

document, and assume that our FPDS implementation is running on single-CPU platforms.

3.1 The scheduler

RTAI Linux system follows a *co-scheduling* model: hard real-time tasks are scheduled by the RTAI scheduler, and the remaining idle time is assigned to the normal Linux scheduler for running all other Linux tasks. The RTAI scheduler supports the standard Linux *schedulables* such as (user) process threads and kernel threads, and can additionally schedule RTAI kernel threads. These have low overhead but they cannot use regular OS functions.

The scheduler implementation supports preemption, and ensures that always the highest priority runnable real-time task is executing.

Primitives offered by the RTAI scheduler API include periodic and non-periodic task support, multiplexing of the hardware timer over tasks, suspension of tasks and timed sleeps. Multiple tasks with equal priority are supported but need to use cooperative scheduling techniques (such as the `yield()` function that gives control back to the scheduler) to ensure fair scheduling.

3.2 Tasks in RTAI

RTAI supports the notion of tasks along with associated priorities. Tasks are instantiated by creating a schedulable object (typically a thread) using the regular Linux API, which can then initialize itself as an RTAI task using the RTAI specific API. Priorities are 16 bit integers with 0 being the highest priority.

Although the terminology of *jobs* is not used in RTAI, all necessary primitives to support periodic tasks with deadlines less than or equal to periods are available. Repetitive tasks are typically represented by a thread executing a repetition, each iteration representing a job. An invocation of the `rt_task_wait_period()` scheduling primitive separates successive jobs. Through a special return value of this function, a task will be informed if it has already missed the time of activation of the next job, i.e. the deadline equal to the period.

In each task control block (TCB) various properties and state variables are maintained, including a 16 bit integer variable representing the running state of the task. Three of these bits are used to represent mutually exclusive running states (*ready*, *running*, *blocked*), whereas the remaining bits are used as boolean flags that are not necessarily mutually exclusive, such as the flag *delayed* (waiting for the next task period), which

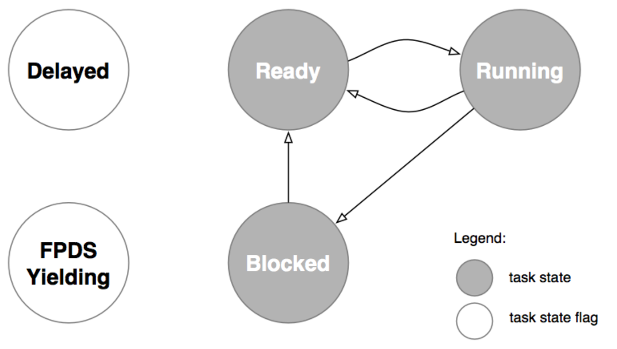

Figure 2. RTAI task states and flags

can be set at the same time as *ready* in the RTAI implementation. This implies that testing the *ready* state is not sufficient for determining the readiness of a task. See Figure 2 for an overview of the task states relevant for our work, including a new bit flag *FPDS Yielding* which we will introduce for our FPDS implementation in Section 5.4.

3.3 Scheduler implementation

In order to provide some context for the design decisions and implementation considerations that will follow, we briefly describe the implementation of the existing RTAI scheduler.

RTAI maintains a *ready queue* per CPU, as a *priority queue* of tasks that are ready to run (i.e., *released*), sorted by task priority. Periodic tasks are maintained with release times of their next job in a separate data structure, the so-called *timed tasks*. This data structure can be an ordered linked list or a red-black tree. If at any moment the current time passes the release time of the head element of the timed tasks list, the scheduler migrates this task to the ready queue of the current CPU. In practice this does not happen instantly but only upon the first subsequent invocation of the scheduler, e.g. through the timer interrupt, and therefore having a maximum latency equal to the period of the timer. The scheduler then selects the head element from the ready priority queue for execution, which is the highest priority task ready to run. The currently running task will be preempted by the newly selected task if it is different. The scheduler ensures that at any given time, the processor executes the highest priority task of all those tasks that are currently ready to execute, and therefore it is a FPPS scheduler.

The implementation of the scheduler is split over two main scheduler functions, which are invoked from different contexts, but follow a more or less similar structure. The function `rt_timer_handler()` is called from within the timer interrupt service routine, and is therefore time-triggered. The other functions, `rt_schedule()` is event-triggered, and performs scheduling when this is requested from within a system call. Each of the scheduler functions performs the following main steps:

1. Determination of the current time
2. Migration of runnable tasks from the timed tasks queue to the ready queue
3. Selection of the highest priority task from the ready queue
4. Context switch to the newly selected task if it is different from the currently running task

After a new task is selected, the scheduler decides on a context switch function to use, depending on the type of tasks (kernel or user space) being switched in and out. The context switch is then performed immediately by a call to this function.

4 Mapping FPDS tasksets

For the case of FPDS we need a different formulation of the taskset. This is because we now must indicate additional subtask structure within each task. There are several ways to approach this.

First, in the task specification we can mark subtask completion by an API call. Many operating systems already implement a primitive that can be used for this, viz., a `yield()` function. In fact, in a cooperative scheduling environment this would be exactly the way to implement FPDS. When a currently running task calls `yield()`, it signals the kernel that it voluntarily releases control of the CPU, such that the scheduler can choose to activate other tasks before it decides to return control to the original task, according to the scheduler algorithm. For the current case of RTAI we would need to modify `yield()` since it currently performs just cooperative scheduling among tasks of the same priority and we would need to ensure that tasks cannot be preempted outside `yield()` functions when in *ready* state.

Second, we can simply use the regular task model for the subtasks. However, this would imply significant overhead in the form of subtask to subtask communication, because the subtasks need to cooperate to maintain the precedence constraints while scheduling these subtasks, which are otherwise implied within the execution of a single task.

Finally, we can develop special notation for this purpose by special data structures and interaction points to be filled in by the user. This, however, would probably not differ a lot from the first case. The advantage would be that, unlike in the first two approaches, the kernel would be aware of the details about the subtask structure which is important for internal analysis by the system, for monitoring or for optimization.

In the first case the API calls play the role of explicit preemption points. These can be programmed directly, but also automated tools could generate preemption points transparently to the programmer guided by other primitives and cues in the source code such as critical sections. Moreover, the `yield()` approach incurs low overhead and limits the modifications to the kernel. We therefore decide for the first approach.

5 Design and implementation

While designing the implementation of our chosen FPDS task model, we have a number of aspects that lead our design choices. First of all, we want our implementation to remain *compatible*; our extensions should be conservative and have no effect on the existing functionality. Any FPDS tasks will need to explicitly indicate desired FPDS scheduling behaviour. *Efficiency* is important because overhead should be kept minimal in order to maximize the schedulability of task sets. Therefore we aim for an FPDS design which introduces as little run-time and memory overhead as possible. Due to the need of keeping time, we do not disable interrupts during FPDS tasks, so the overhead of *interrupt handling* should be considered carefully as well. Because we want to be able to integrate our extensions with future versions of the platform, our extensions should be *maintainable*, and written in an *extendable* way, with flexibility for future extensions in mind.

We aim for a FPDS implementation that is non-preemptive only with respect to other tasks; i.e. a task will not be *preempted* by another task, but can be *interrupted* by an interrupt handler such as the timer ISR.

The process of implementing FPDS in RTAI/Linux was done in several stages. Because the existing scheduler in RTAI is an FPPS implementation with no direct support for non-preemptive tasks, the first stage consisted of a proof of concept attempt at implementing FPNS in RTAI. The following stages then built upon this result to achieve FPDS scheduling in RTAI in accordance with the task model and important design aspects described above.

5.1 FPNS design

The existing scheduler implementation in RTAI is FPPS: it makes sure that at every moment in time, the highest priority task that is in *ready* state has control of the CPU. In contrast, FPNS only ensures that the highest priority ready task is started upon a job finishing, or upon the arrival of a task whenever the CPU is idle. For extending the FPPS scheduler in RTAI with support for FPNS, the following extensions need to be made:

- Tasks, or individual jobs, need a method to indicate to the scheduler that they need to be run non-preemptively, as opposed to other tasks which may want to maintain the default behaviour.
- The scheduler needs to be modified such that any scheduling and context switch activity is deferred until a non-preemptive job finishes.

Alternatively, arrangements can be made such that at no moment in time a ready task exists that can preempt the currently running FPNS task, resulting in a schedule that displays FPNS behaviour, despite the scheduler being an FPPS implementation. Both strategies will be explored.

5.1.1 Using existing primitives

Usage of the existing RTAI primitives for influencing scheduling behaviour to achieve FPNS would naturally be beneficial for the *maintainability* of our implementation.

When investigating the RTAI scheduler primitives exported by the API [12], we find several that can be used to implement FPNS behaviour. These strategies range from the blocking of any (higher priority) tasks during the execution of a FPNS job, e.g. through suspension or mutual exclusion blocking of these tasks, to influencing the scheduler decisions by temporary modifications of task priorities. What they have in common however is that at least 2 invocations of these primitives are required per job execution, resulting in RTAI in additional overhead of at least two system calls per job. Some of these methods, such as explicit suspension of all other tasks, also have the unattractive property of requiring complete knowledge and cooperation of the entire task set.

5.1.2 RTAI kernel modifications

As an alternative to using existing RTAI primitives which are not explicitly designed to support FPNS, the notion of a non-preemptible task can be moved into the RTAI kernel proper, allowing for modified scheduling behaviour according to FPNS, without introducing extra overhead during the running of a task as induced by the mentioned API primitives. Looking ahead to our goal of implementing FPDS, this also allows more fine grained modifications to the scheduler itself, such that *optional* preemption points become possible in an efficient manner: rather than trying to disable the scheduler during an FPNS job, or influencing its decisions by modifying essential task parameters such as priorities, the scheduler would become aware of non-preemptible or deferred preemptible tasks and support such a schedule with intelligent decisions and primitives. It does however come at the cost of implementation and maintenance complexity. Without availability of documentation of the design and implementation of the RTAI scheduler, creating these extensions is more difficult and time consuming than using the well documented API. And because the RTAI scheduler design and implementation is not stable, as opposed to the API, continuous effort will need to be spent on maintaining these extensions with updated RTAI source code, unless these extensions can be integrated into the RTAI distribution. Therefore we aim for a patch with a small number of changes to few places in the existing source code.

An important observation is that with respect to FPPS, scheduling decisions are only made differently during the execution of a non-preemptive task. Preemption of any task must be initiated by one of the scheduling functions, which means that one possible implementation of FPNS would be to alter the decisions made by the scheduler if and only if a FPNS task is currently executing. This implies that our modifications will be *conservative* if they change scheduling behaviour during the execution of non-preemptive tasks only.

During the execution of a (FPNS) task, interference from other, higher priority tasks is only possible if the scheduler is invoked through one of the following ways:

- The scheduler is invoked from within the timer ISR
- The scheduler is invoked from, or as a result of a system call by the current task

The first case is mainly used for the release of jobs - after the timer expires, either periodically or one-shot, the scheduler should check whether any (periodic) tasks should be set *ready*, and then select the highest priority one for execution. The second case applies when a task does a system call which alters the state of the system in such a way that the schedule *may* be affected, and thus the scheduler should be called to determine this. With pure FPNS, all scheduling work can be deferred until the currently running task finishes execution. Lacking any optional preemption points, under no circumstances should the current FPNS task be preempted. Therefore, the condition of a currently running, ready FPNS task should be detected as early on in the scheduler as possible, such that the remaining scheduling work can be deferred until later, and the task can resume execution as soon as possible, keeping the overhead of the execution interruption small.

In accordance with our chosen task model in Section 4, we decide to modify the kernel with explicit non-preemptive task support, as described in section 5.1.2.

5.2 FPNS implementation

Towards an FPNS aware RTAI scheduler, we extend the API with a new primitive named `rt_set_preemptive()`, consistent with other primitives that can alter parameters of tasks, that accepts a boolean parameter indicating whether the calling task should be preemptible, or not. This value will then be saved inside the task's control block (TCB) where it can be referenced by the scheduler when making scheduling decisions. This *preemptible* flag inside the TCB only needs to be set once, e.g. during the creation of the task and not at every job execution, such that there is no additional overhead introduced by this solution.

Execution of the scheduler should be skipped at the beginning, if the following conditions hold for the currently running task:

- The (newly added) *preemptible* boolean variable is unset, indicating this is a non-preemptive task;
- The *delayed* task state flag is not set, indicating that the job has not finished;
- The *ready* task state flag is set, indicating that the job is ready to execute and in the ready queue.

We have added these tests to the start of both scheduler functions `rt_schedule()` and `rt_timer_handler()`, which resulted in the desired FPNS scheduling for non-preemptible tasks. For preemptive tasks, which have the default value of 1 in the *preemptible* variable of the TCB, the scheduling behaviour is not modified, such that the existing FPPS functionality remains unaffected.

5.3 FPDS design

Following the description of FPNS in the previous section, we move forward to the realisation of a FPDS scheduler, by building upon these concepts. An FPDS implementation as outlined in Section 4, where subtasks are modeled as non-preemptive tasks with preemption points in between, has the following important differences compared to FPNS:

- A job is not entirely non-preemptive anymore; it may be preempted at predefined preemption points, for which a scheduler primitive needs to exist.
- The scheduling of newly arrived jobs can no longer be postponed until the end of a non-preemptive job execution, as during a (optional) preemption point in the currently running job information is required about the availability of higher priority ready jobs.

There are several ways to approach handling interrupts which occur during the execution of nonpreemptive subjob. First, the interrupt may be recorded with all scheduling postponed until the scheduler invocation from `yield()` at the next preemption point, similar to our FPNS implementation, but at much finer granularity.

Alternatively, all tasks which have arrived can be moved from the pending queue to the ready queue directly (as is the case under FPPS), with only the context switch postponed until the next preemption point. This has the advantage that there is opportunity for optional preemption points to be implemented, if the information about the availability of a higher priority, waiting task can be communicated in an efficient manner to the currently running FPDS task for use at the next optional preemption point. These two alternatives can be described as *pull* versus *push* models respectively. They represent a tradeoff, and the most efficient model will most likely depend on both the task sets used, and the period of the timer.

On our platform we could not measure the difference between these two alternatives; any difference in efficiency between the two approaches was lost in the noise of our experiment. Therefore we opted to go with the last alternative, as this would not require extensive rewriting of the existing scheduler logic in RTAI, and thereby fit our requirements of *maintainability* and our extensions being *conservative*. The efficiency differences between these approaches may however be relevant on other platforms, as described in [10], based on [8].

5.4 FPDS implementation

It would appear that using a standard `yield()` type function, as present within RTAI and many other operating systems, would suffice for implementing a preemption point. Upon investigation of RTAI's yield function (`rt_task_yield()`) it turned out however that it could not be used for this purpose unmodified. This function is only intended for use with round-robin scheduling between tasks having equal priority, because under the default FPPS scheduler of RTAI there is no reason why a higher priority, ready task would not already have preempted the current, lower priority task. However with the non-preemptive tasks in FPDS, a higher priority job may have arrived but not been context switched in, so checking the ready queue for *equal* priority processes is not sufficient. An unconditional call to scheduler function `rt_schedule()` should have the desired effect, as it can move newly arrived tasks to the ready queue, and invoke a preemption if necessary. However, the modified scheduler will evaluate the currently running task as non-preemptive, and avoid a context switch. To indicate that the scheduler is being called from a preemption point and a higher priority task is allowed to preempt, we introduce a new bit flag `RT_FPDS_YIELDING` to the task state variable in the TCB, that is set before the invocation of the scheduler to inform it about this condition. The flag is then reset again after the scheduler execution finishes.

Due to the different aim of our FPDS yield function in comparison to the original yield function in RTAI we decided not to modify the existing function, but create a new one specific for FPDS use instead: `rt_fpds_yield()`. The FPDS yield function is simpler and more efficient than the regular yield function, consisting of just an invocation of the scheduler function wrapped between the modification of task state flags. This also removed the need to modify the existing code which could introduce unexpected regressions with existing programs, and have a bigger dependency on the existing code base, implying greater overhead in maintaining these modifications in the future.

5.4.1 Scheduler modifications

Working from the FPNS implementation of Section 5.1, the needed modifications to the scheduling functions `rt_schedule()` and `rt_timer_handler()` for FPDS behaviour are small. Unlike the FPNS case, the execution of the scheduler cannot be deferred until the completion of a FPDS (sub)task if we want to use the push mechanisms described in Section 5.3, as the scheduler needs to finish its run to acquire this information. Instead of avoiding the execution of the scheduler completely, for FPDS we only defer the invocation of a context switch to a new task, if the currently running task is not at a preemption point.

The existing `if` clause introduced at the start of the scheduler functions for FPNS is therefore moved to a section in the scheduler code between parts 3 and 4 as described in Section 3.3, i.e. at which point the scheduler has decided a new task should be running, but has not started the context switch yet. At this point we set a newly introduced TCB integer variable `should_yield` to *true*, indicating that the current task should allow itself to be preempted at the next preemption point. This variable is reset to *false* whenever the task is next context switched back in.

With these modifications, during a timer interrupt or explicit scheduler invocation amidst a running FPDS task, the scheduler will be executed and wake up any timed tasks. If a higher priority task is waiting at the head of the ready queue, a corresponding notification will be delivered and the scheduler function will exit without performing a context switch. To allow an FPDS task to learn about the presence of a higher priority task waiting to preempt it, indicated by the *should_yield* variable in its TCB in kernel space which it however does not have permissions for to read directly, we introduced a new RTAI primitive `rt_should_yield()` to be called by the real-time task, which returns the value of this variable. In the current implementation this does however come at the cost of performing a system call at every preemption point.

In a later version of our FPDS implementation in RTAI, not yet reflected in the measurements in this paper, we improved the efficiency of preemption points by removing the need for this system call. The location of the *should_yield* variable was moved from the TCB in kernel space to user space in the application's address space, where it can be read efficiently by the task. Upon initialization, the FPDS task registers this variable with the kernel, and makes sure the corresponding memory page is locked into memory. The contents of this variable are then updated exclusively by the kernel, which makes use of the fact that updates are only necessary during the execution of the corresponding FPDS task, when its address space is already active and loaded in physical memory. This design is similar to the one described in [2] for implementing non-preemptive sections.

5.5 Abstraction

For *simplicity* of usage of our FPDS implementation, we created a dynamic library called *libfpds* which can be linked to a real-time application that wants to use FPDS. A preemption point can then be inserted into the code by inserting an invocation of `fpds_pp()`, which internally performs `rt_should_yield()` and `fpds_yield()` system calls as necessary. Alternatively, the programmer can pass a parameter to indicate that the task should not automatically be preempted if a higher task is waiting, but should merely be informed of this fact.

6 Key performance indicators

Our implementation should be checked for the following elements, which relate to the design aspects mentioned in Section 5:

- The interrupt latency for FPDS. This is intrinsic to FPDS, i.e. there is additional blocking due to lower priority tasks. It has been dealt with in the analysis in [3, 4];
- The additional *run-time* overhead due to additional code to be executed. This will be measured in Section 7;
- The additional *space* requirements due to additional data structures and flags. Our current implementation introduces only two integer variables to the TCB, so the space overhead is minimal;
- The number of *added*, *changed*, and *deleted* lines of code (excluding comments) compared to the original RTAI version. Our extension adds only 106 lines and modifies 3 lines of code, with no lines being removed;
- The *compatibility* of our implementation. Because our extensions are conservative, i.e. they don't change any behaviour when there are no non-preemptive tasks present, compatibility is preserved. This is also verified by our measurements in Section 7.1.

7 Measurements

We performed a number of experiments to measure the additional overhead of our extensions compared to the existing FPPS scheduler implementation. The hardware used for these tests was an Intel Pentium 4 PC, with 3 Ghz CPU, running Linux 2.6.24 with (modified) RTAI 3.6-cv.

7.1 Scheduling overhead

The goal of our first experiment is to measure the overhead of our implementation extensions for existing real-time task sets, which are scheduled by the standard RTAI scheduler, i.e. following FPPS. For non-FPDS task sets, scheduling behaviour has not been changed by our conservative implementation, but our modifications may have introduced additional execution overhead.

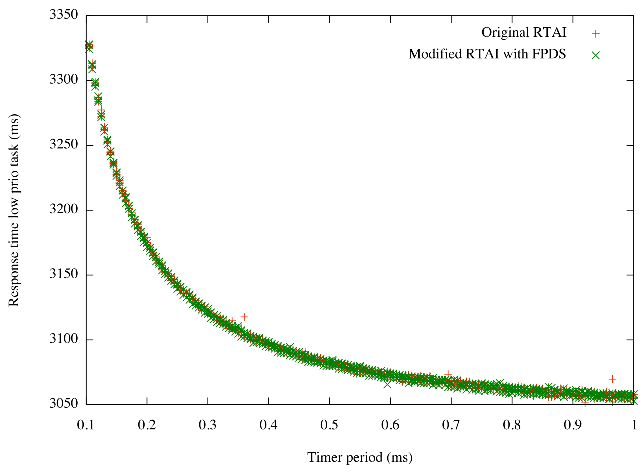

Figure 3. Overhead of the modified kernel for FPPS task sets

As our example task set we created a program with one non-periodic, low priority, long running FPPS task τ_l, and one high priority periodic FPPS task τ_h. τ_l consists of a `for` loop with a parameterized number of iterations m, to emulate a task with computation time C_l. The computation time of the high priority task, C_h, was 0; the only purpose of this empty task is to allow for measurement of overhead of scheduling by presenting an alternative, higher priority task to the scheduler. T_h was kept equal to the period of the timer, such that a new high priority job is released at every scheduler invocation from the timer event handler.

Since, from the perspective of an FPPS task, the only modified code that is executed is in the scheduler functions, we measured the response time of task τ_l under both the original and the modified FPDS real-time RTAI kernel, varying the period of the timer interrupt, and thereby the frequency of scheduler invocations encapsulated by the timer event handler. The results are shown in Figure 3.

As expected, there is no visible difference in the overhead of the scheduler in the modified code compared to the original, unmodified RTAI kernel. For an FPPS task set the added overhead is restricted to a single `if` statement in the scheduler, which references 3 variables and evaluates to *false*. This overhead is unsubstantial and lost in the noise of our measurements. We conclude that there is no significant overhead for FPPS task sets introduced by our FPDS extensions.

7.2 Preemption point overhead

With an important aspect of FPDS being the placement of preemption points in task code between subtasks, the overhead introduced by these preemption points is potentially significant. Depending on the frequency of preemption points, this could add a substantial amount of additional computation time to the FPDS task. In the tested implementation, the dominating overhead term is expected to be the overhead of a system call (C_{sys}) performed during every preemption point, to check whether the task should yield for a higher priority task. The system call returns the value of a field in the TCB, and performs no additional work.

We measured the overhead of preemption points by creating a long-running, non-periodic task τ_l with fixed computation time C_l implemented by a `for` loop with $m = 100M$ iterations, and scheduled it under both FPPS and FPDS. The division into subtasks of task τ_l has been implemented by invoking a preemption point every n iterations, which is varied during the course of this experiment, resulting in $\lceil m/n \rceil$ preemption point invocations.

For the FPPS test the same task was used, except that every n iteration interval only a counter variable was increased, instead of the invocation of a preemption point. This was done to emulate the same low priority task as closely as possible in the context of FPPS.

The response time R_l was measured under varying intervals of n for both FPPS and FPDS task sets. The results are plotted in Figure 4.

Clearly the preemption points introduced in the lower priority task introduce overhead which does not exist in a FPPS system. The extra overhead amounts to about 440 μs per preemption point invocation, which corresponds well with a measured value C_{sys} of 434 μs per general RTAI system call overhead which we obtained in separate testing. This suggests that the overhead of a preemption point is primary induced by the `rt_should_yield()` system call in the preemption point implementation, which is invoked unconditionally.

7.3 An example FPDS task set

Whereas the previous experiments focussed on measuring the overhead of the individual extensions and primitives added for our FPDS implementation, we performed an experiment to compare the worst case response time of a task set under FPPS and FPDS as well. The task set of the previous experiment was extended with a high priority task τ_h with a non-zero computation time C_h. For this experiment we varied the period of the high priority task. To keep the workload of the low priority task constant, we fixed the interval n of preemption points to a value (5000) frequent enough to allow preemption by τ_h without it missing any deadlines under all values of $T_h \geq 1.2ms$ under test. The response time of the low priority task R_l is plotted in Figure 5.

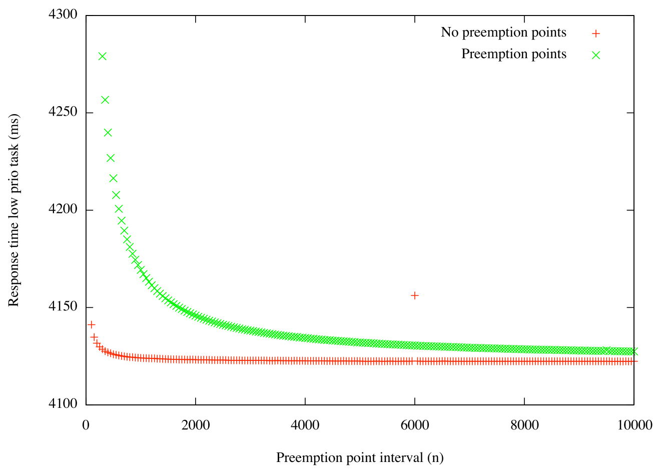

Figure 4. Overhead of preemption points

The relative overhead appears to depend on the frequency of high priority task releases and the resulting preemptions in preemption points, as the number of preemption points invoked in the duration of the test is constant. The results show an increase in the response time of τ_l for FPDS of at most 17% with a mean around 3%. The large discrepancy of the results can probably be attributed to the unpredictable interference from interrupts and the resulting invalidation of caches. Considering the relatively low mean overhead of FPDS, we would like to identify the factors which contribute to the high variation of the task response time, and investigate how these factors can be eliminated (see Section 8).

8 Conclusions and future work

In this paper we have presented our work on the implementation of FPDS in the real-time operating system, RTAI/Linux. We have shown that such an implementation in a real-world operating system is feasible, with only a small amount of modifications to the existing code base in the interest of future maintainability. Furthermore, a set of experiments indicated that our modifications introduced no measurable overhead for

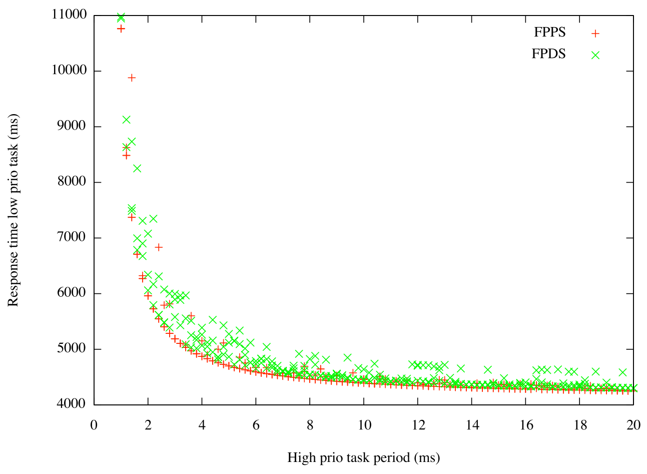

Figure 5. A task set scheduled by FPPS and FPDS

FPPS task sets, and only small mean overhead introduced by converting a FPPS task set into FPDS.

As a follow up to this work, we would like to further investigate the tradeoffs between regular preemption points and optional preemption points. In particular we would like to gain quantitive results exposing the tradeoffs in moving tasks to the ready queue during the execution of a non-preemptive subjob, with respect to saved system calls, invalidation of caches and other overheads.

Finally, we would like to research how additional information about FPDS task structures in the form of a DAG can benefit the scheduling decisions. A DAG will specify the worst-case computation times of subtasks and thus form a sort of contract between the tasks and the scheduler, allowing to combine FPDS with reservations. Methods for monitoring and enforcing these contracts need to be investigated.

References

[1] RTAI 3.6-cv - The RealTime Application Interface for Linux from DIAPM, 2009.

[2] B. B. Brandenburg, J. M. Calandrino, A. Block, H. Leontyev, and J. H. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 342–353. IEEE Computer Society, 2008.

[3] R. Bril, J. Lukkien, and W. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited. In *Proc. 19th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 269–279, July 2007.

[4] R. Bril, J. Lukkien, and W. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited – with extensions for ECRTS'07 –. Technical Report CS Report 07-11, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven (TU/e), The Netherlands, April 2007.

[5] A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. In S. Son, editor, *Advances in Real-Time Systems*, pages 225–248. Prentice-Hall, 1994.

[6] A. Burns. Defining new non-preemptive dispatching and locking policies for ada. *Reliable SoftwareTechnologies —Ada-Europe 2001*, pages 328–336, 2001.

[7] A. Burns, M. Nicholson, K. Tindell, and N. Zhang. Allocating and scheduling hard real-time tasks on a parallel processing platform. Technical Report YCS-94-238, University of York, UK, 1994.

[8] A. Burns, K. Tindell, and A. Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Transactions on Software Engineering*, 21(5):475–480, May 1995.

[9] R. Gopalakrishnan and G. M. Parulkar. Bringing real-time scheduling theory and practice closer for multimedia computing. *SIGMETRICS Perform. Eval. Rev.*, 24(1):1–12, 1996.

[10] M. Holenderski, R. J. Bril, and J. J. Lukkien. Using fixed priority scheduling with deferred preemption to exploit fluctuating network bandwidth. In *ECRTS '08 WiP: Proceedings of the Work in Progress session of the 20th Euromicro Conference on Real-Time Systems*, 2008.

[11] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[12] G. Racciu and P. Mantegazza. *RTAI 3.4 User Manual, rev 0.3*, 2006.