Real-time System Overheads: a Literature Overview

Mike Holenderski

October 16, 2008

Abstract

In most contemporary systems there are several jobs concurrently competing for shared resources, such as a processor, memory, network, sensors or other devices. Sharing a resource between several jobs requires synchronizing the jobs, specifying when which job will have access to the resource. A common synchronization method is *scheduling*. Executing a schedule requires switching resource assignments between the jobs, which is usually referred to as *context switching*. The overheads associated with scheduling and context switching are part of the system overheads.

Initially, in the spirit of keeping things simple, real-time systems analysis abstracted from many details, including the overheads incurred by the operating system. This has led to inherently optimistic results, i.e. accepting collections of jobs, which if executed on a real system will fail to meet all the constraints.

In this paper we consider a less idealized platform by taking some practical aspects into account. We present an overview of literature dealing with real-time system overheads, in particular the scheduling and context switch overheads. We focus on sharing a single timely resource, such as a processor, in the context of Fixed Priority Preemptive Scheduling. We treat in detail the overheads due to preemption, and due to blocking of tasks for different resource access protocols (which is the main contribution of this paper).

Contents

1 Introduction

In most contemporary systems there are several *jobs* concurrently competing for *shared resources*, such as a processor, memory, network, sensors or other devices. In the remainder of this paper we assume the resources are mutually exclusive, i.e. at any particular time at most one job can be using the resource.

Sharing a resource between several jobs requires *scheduling* the jobs, specifying when which job will have access to the resource. Executing a schedule requires switching between the jobs, which is usually referred to as *context switching*. Both scheduling and context switching are taken care of by the *platform*, which encompasses the real-time operating system, the underlying architecture and the hardware *resources*, such as processor, memory or interconnect.

At the core of real-time systems analysis lies the question of whether a given collection of jobs is schedulable or not, i.e. whether all the jobs will be able to execute on a given platform meeting their real-time constraints.

Initially, in the spirit of keeping things simple, real-time systems analysis considered idealized platforms and abstracted from many details, including the overheads incurred by the operating system. This has led to inherently optimistic results, i.e. accepting collections of jobs, which if executed on a real system will fail to meet all the constraints. A simple, though potentially expensive, solution is over-dimensioning the system resources.

In this paper we consider a less idealized platform by taking some practical aspects into account. We present an overview of literature dealing with real-time system overheads, in particular the scheduling and context switch overheads. We focus on sharing a single bandwidth resource [1] (e.g. processor) between jobs specified by a task model.

We start off with defining some terminology in Section 2. We state and motivate our goal for this paper in Section 3 and continue with a discussion of real-time system overheads due to preemption in Section 4 and due to blocking (which is our main contribution) in Section 5. In Section 5.6 we elaborate on real-time system overhead due to tick-scheduling, and conclude with a summary and future work in Section 6.

2 Some definitions

A real-time system guarantees that the tasks in the system receive their required resources on time (i.e. before their deadline), so that the jobs can complete before their deadline. In this paper we consider the *task model*, where the *requirement* of a job for a *particular resource* is specified by an abstract entity called *task*. We say that jobs are generated by tasks.

2.1 Task model

A task τ_i is described by a *phasing* φ_i, *(worst-case) computation time* C_i and *deadline* D_i, which means that a job generated by task τ_i at time φ_i requires at least C_i time of the resource before deadline D_i, i.e. before time $\varphi_i + D_i$.[2]

A *periodic task* generates jobs at regular intervals and is further described by a period T_i, with job τ_{ij} representing the job released in the jth period at time $\varphi_i + jT_i$ with the deadline of $\varphi_i + jT_i + D_i$.

A *sporadic task* generates jobs with a *minimum* inter-arrival time and is further described by a period T_i, with job τ_{ij}representing the job released in the jth period at time $\alpha_i(j) \geq \alpha_i(j-1) + T_i$ with the deadline of $\alpha_i(j) + D_i$, where $\alpha_i(j)$ is the arrival time of the jth job of task τ_i.

Aperiodic tasks are those which generate a single job released at time φ_i with the deadline of $\varphi_i + D_i$. When there is no confusion we will refer to a job generated by an aperiodic task τ_i simply as τ_i.

In the remainder of this paper we assume that for all tasks $D_i \leq T_i$.

The i index in τ_i refers to the priority of the task, where smaller i represents higher priority. We assume there is a single task per priority and that there are altogether n tasks in the task set. We further assume that the jobs do not self-suspend.

[1] In contrast to spatial resources, such as memory.

[2] Note that here we restrict ourselves to the processor resource. The computation time requirement C_i can be generalized to a *reservation* time to accommodate other timely resources as well, such as network.

Jitter Real-time literature distinguishes several kinds of jitter. Among others

- *activation jitter* is the fluctuation of event arrival relative to the arrival of the corresponding task (as specified by its period) and is defined by the environment. We ignore activation jitter in the remaining discussion.
- *release jitter* is the fluctuation of task release relative to task arrival and depends on the platform, such as ready queue management. We treat the release jitter in Section 2.2.

2.2 Platform model

In this paper we treat the platform as a composition of an operating system and the underlying hardware, providing the necessary functionality to execute a given workload (modeled by a task set described above). At the core of an operating system lies the kernel. A real-time operating systems is usually based on the micro-kernel architecture, where the kernel is stripped down to the bare essentials and provides functionality to handle: timer interrupts, external interrupts and system calls ([Liu 2000] presents a nice overview of kernels).

System calls can be regarded as non-preemptive subjobs and we ignore these for the time being in our discussion of Fixed Priority Preemptive Scheduling (FPPS). We also omit external interrupts and focus on the timer interrupts. Scheduling external interrupts is discussed in [Zhang and West 2006].

A platform is responsible for managing resources shared between several tasks. It therefore has to *schedule* these tasks on the resources, avoiding conflicts. One of the key questions is when the scheduling takes place.

Non-preemptive scheduling In a *non-preemptive* system (also called *co-operative* [Audsley et al. 1996]) resources can be reassigned to jobs only at job boundaries. When a job completes, the scheduler checks whether new periodic or sporadic tasks have arrived and schedules the next job, according to some policy.

Event-triggered scheduling In an *event-triggered* preemptive system scheduling decisions are taken whenever an external event releases a sporadic task, an internal clock event releases a periodic task or a previously running job completes or blocks. This is implicitly assumed in most of the standard literature dealing with FPPS [Buttazzo 2005], together with the assumption of negligible scheduling overhead.

Time-trigerred scheduling If we assume *tick scheduling* as described in [Liu 2000] (also referred to as *time-trigerred* scheduling), then the scheduling decisions are only taken at regular intervals, called *clock interrupts* (also referred to as clock ticks). A clock interrupt can be regarded as a special timer interrupt. When an even occurs in-between clock interrupts, then the even handler is delayed until the next scheduler invocation. Section 5.6 takes a closer look at what exactly happens when an event occurs and how the additional delays can be accounted for in the response time analysis.

2.3 Feasibility analysis

In real-time systems one is usually interested in whether a given task set can be executed on time on a given configuration (of resources). There are several *feasibility analysis* methods, e.g. utilization based Liu & Layland test [Liu and Layland 1973] and Hyperbolic Bound [Bini et al. 2003] analysis for Rate Monotonic scheduling, and Worst Case Response Time analysis [Audsley et al. 1993] for Fixed Priority Preemptive Scheduling in general (with $D_i \leq T_i$).

2.3.1 Utilization based analysis

is usually easiest to perform. It involves computing a polynomial function in *utilization factors* $U_i = C_i/T_i$ of all tasks and comparing the result to a specific bound. E.g. [Liu and Layland 1973]

$$\sum_{i=1}^{n} U_i \leq n(2^{1/n} - 1)$$

or [Bini et al. 2003]

$$\prod_{i=1}^{n}(U_i + 1) \leq 2$$

It is usually pessimistic, in that it only provides a *sufficient* condition, which may cause poor processor utilization and sometimes reject task sets which could actually be scheduled.

2.3.2 Response time analysis

(also referred to as time demand analysis [Liu 2000]) accepts a task set if for all tasks their response time is smaller or equal to their deadline.

The *response time* of a job $\tau_{ij}$is given by

$$R_{ij} = C_i + B_{ij} + I_{ij} \tag{1}$$

where B_{ij} and I_{ij} represent the time $\tau_{ij}$is active but cannot execute due to *blocking* and *preemption*, respectively.

Worst Case Response Time analysis is more complicated than utilization based analysis, but it can provide a less pessimistic bound. It involves, for every task, finding the worst possible response time and comparing it to the deadline. The worst case response time occurs at a critical instant. Since the Worst Case Response Time analysis focuses on job τ_{ij}at the critical instance only (ignoring the cases with smaller R_{ij}), the j index is usually ignored and Equation 1 becomes

$$WR_i = C_i + B_i + I_i \tag{2}$$

where WR_i is the Worst Case Response Time of task τ_i. The analysis deals with finding the values for B_i and I_i. Most literature deals with estimating the contribution of the I_i term, i.e. the total preemption time of job τ_{ij} . The standard method for computing R_i (assuming fixed B_i) is given by the iterative equation [Joseph and Pandya 1986, Burns 1994]

$$\begin{array}{rcl} WR_i^0 & = & \sum_{j \in hp(i)} C_j \\ WR_i^{n+1} & = & C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{WR_i^n}{T_j} \right\rceil C_j \end{array} \tag{3}$$

where $hp(i)$ is the set of tasks with priority higher than i, until $WR_i^n > D_i$ (meaning the task set is not schedulable) or $WR_i^n = WR_i^{n+1}$ (meaning the task set is schedulable).

2.4 Scheduling overhead vs. context switch overhead

Litereature distinguishes two kinds of overhead [Audsley et al. 1995]:

Scheduling overhead (also referred to as real-time clock overhead [Burns et al. 1993b]) models the overhead incurred by the clock interrupt handler which is responsible for scheduling, i.e. moving newly arrived and preempted jobs between the queues and selecting the running job. It also includes managing timing services, accounting for CPU usage and initiating preemption for the sake of multitasking [Liu 2000]. It is influenced by the (implementation of) the scheduling algorithm, e.g. Fixed Priority Scheduling experiences less scheduling overhead, than Dynamic Priority Scheduling.

The scheduling complexity is at most linear in the number of tasks, but it can be smaller with appropriate hardware support.

Context switch overhead represents the overhead associated with preempt- ing current job, saving its context, loading the context of the next job and resuming the next job. The context switch into task τ_i is performed at the kernel (i.e. highest) priority level, but only when task τ_i is ready to execute [Klein et al. 1993]. The context switch into τ_i behaves as follows:

1. A zero length action is executed at priority i.[3]

[3]This "virtual" step makes sure that the context switchoverhead is included in the worst-case computation time of task τ_i.

2. The context switch is executed at priority 0 (i.e. kernel).
3. The rest of τ_i is executed at priority i.

In a *non-preemptive* system, where resources can be reassigned to jobs only at job boundaries, the scheduler is only invoked at the end of a job (assuming the interrupts are disabled).

In an *event-triggered* preemptive system the scheduling overhead can be incurred whenever an event occurs. The scheduler is usually associated with the event interrupt handler. A drawback of this approach is that the scheduler has to be executed at every scheduling point. In real systems where the scheduler overhead is not negligible, a frequently executing scheduler can deplete the available resources.[4]

In a *time-triggered* preemptive system the scheduling overhead is incurred at a regular interval. The scheduler is usually associated with a timer handler and is invoked periodically. As a disadvantage a task arriving in between the scheduler invocations will experience release jitter [Audsley et al. 1996].

As mentioned in Section 2.2, in this paper we consider time-triggered systems. Figure 1 illustrates the scheduling and context switch overheads in a time-triggered preemptive system.

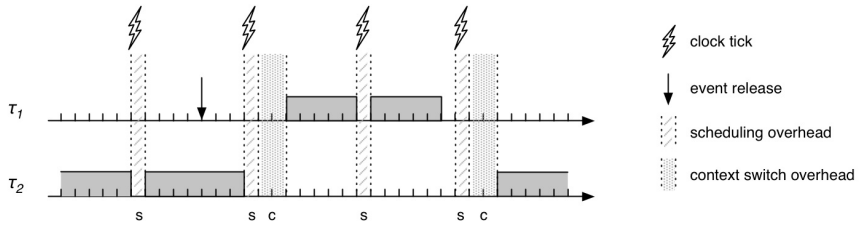

Figure 1: Scheduling (s) and context switch (c) overheads in a time-triggered preemptive system

It shows two tasks: τ_1 with $C_1 = 10$ and τ_2 with $C_2 > 17$ (the exact value is not relevant here). Furthermore, the clock interrupt handler is invoked regularly (indicated by the lightning) with period $T_0 = 8$ and computation time $C_0 = 1$. The execution of task τ_2 is regularly preempted by the clock handler. Somewhere before the second clock tick τ_1 is released. During the second clock interrupt handler invocation the scheduler notices the newly arrived τ_1 and switches from τ_2 to τ_1. After executing for 10 time units, τ_1 is finished and releases the processor until the next clock tick. During the next scheduler invocation the processor is switched back to τ_2.

Note the idle gap between the time $\tau_1$is finished and the next clock tick. It is there because we assume *idling scheduling*. This guarantees that the scheduler will be invoked at regular intervals, which is assumed in the following sections.

3 Goal for this paper

The goal of this paper is to identify which parts of the real-time system overheads domain are already sufficiently covered in the literature and where are the gaps and potential research opportunities. We proceed by identifying the key research questions and subsequently discussing the results found in the literature.

3.1 Research questions

The main research questions regarding scheduling and context switch overheads deal with

- Modeling
  - How can we model the overheads? As a high priority task or extend the computation time?
  - When does the context switchin the model occur? At the beginning or end of the task?

[4][Mok 1983]introduced *sporadic tasks* to bound the inter-arrival time between the jobs of the same task and therefore bound the depletion.

  - Which job should be charged the overhead cost, i.e. which task should have its computation time C increased with the the context switchcost?
  - How many context switches occur during runtime?
  - How long is each context switch?
- Reducing the number of context switches
  - by design
  - by modification of existing system

In the following sections we will address these research questions, in the context of Fixed Priority Preemptive Scheduling (FPPS) on a single CPU resource.

3.2 Motivation

We motivate our work with references to case studies in the literature exposing the system overheads in real systems. As the real-time system overheads have been addressed in literature before, the main contribution of this paper is a summary of these results.

3.2.1 Performance

Scheduling overhead In their experimental results (based on sorting an array under Linux on various Intel platforms) [Tsafrir 2007] show an overall 0.5% - 1.5% slowdown due to scheduling overhead.

A different study of interrupt servicing overheads in Linux on an ARM platform [David et al. 2007] shows an *average* increase of only 0.28% - 0.38% of the total running time of the tasks.

Context switch overhead A study of *average* context switch overheads in Linux on an ARM platform [David et al. 2007] shows only 0.17% - 0.25%.

FPGAs give the possibility to reconfigure the hardware during runtime, which can be regarded as a context switch. The delays, however, run into milliseconds, with Virtex II taking 25ms for complete reconfiguration [Butel et al. 2004]. The newer models hope to reduce the latency through partial reconfiguration, allowing to reconfigure smaller parts of the FPGA independently. However, it proves difficult due to strongly intertwined layout of the components on the FPGA.

It is important to notice that the percentages mentioned above refer to the *average* scheduling and context switch overheads. [Burns et al. 1994] show that the *worst-case* overhead can be much larger and present analytical results based on the data from a real Olympus satellite control system, showing a worst-case scheduling and context switch overhead (together) of upto 10.88% of the total utilization.

In certain type of applications, e.g. multimedia applications, the system may exhibit very high context switch cost, due to high latency of the I/O devices [Echague et al. 1995][5]. Therefore, given the relatively high worst-case overheads, it makes sense to consider these overheads in systems where the computation time estimates are accurate and the utilization needs to be maximized close to 100%.

3.2.2 Maintenance

Switching context on a DSP, where there is no platform support for context switching, includes manually (i.e. by the programmer) storing and loading all affected registers, which is complicated and error-prone.

[5] Note that it highly depends on the chosen operating system.

4 Real-time system overheads due to preemption

In this section we focus on two real-time system overheads: the scheduling overhead discussed in Section 4.1 and the context switching overhead discussed in Section 4.2.

4.1 Scheduling overhead

As mentioned in Section 2.2, different scheduling strategies are possible. Let us take a look from the perspective of an operating system at what happens when a task arrives, either due to an external interrupt for aperiodic or sporadic tasks, or due to a clock interrupt for a periodic task. We introduce a common terminology, which will allow us to relate the existing literature on scheduling overheads.

4.1.1 Event-triggered scheduling

Figure 2: Event handling with event-triggered scheduling

t_0: An event occurs and is detected by a sensor. The sensor dispatches an interrupt.

t_1: The corresponding hardware interrupt handler is dispatched on the CPU and the interrupt handler (also called *scheduler* or *immediate interrupt service*) starts executing, C_{int} representing the fixed overhead of every clock handler invocation. The delay between t_0 and t_1 depends on the platform and the operating system (e.g. bus availability).

t_2: The interrupt handler requires C_q time to insert the new task (i.e. the corresponding *scheduled interrupt handling routine*) into the ready queue. Note that during C_{int} and C_q a higher priority interrupt may arrive. Two strategies are possible:

- Interrupts disabled at t_1 and enabled at t_3, giving rise to non-preemptive interrupts. The drawbacks include longer latency for higher priority interrupts and possibly missing interrupts (if the new interrupts are not queued) [Schoen et al. 2000]. The interrupt handler may experience blocking due to a non-preemptive lower priority interrupt handler.
- Interrupt handlers are split into two phases: *prologue* and *epilogue* [Schoen et al. 2000]. The Prologue is short and non-preemptive, while the main work of the interrupt handler is performed by the preemptive epilogue. Reducing the length of non-preemptive sections of the interrupt handler serves the purpose of *interrupt transparency*. The interrupt handler may experience blocking due to a lower priority handler or preemption due to higher priority handlers, however the blocking duration is shorter than in the case of non-preemptive interrupts.

t_3: Depending on the task selected to run by the scheduler, either the running task is resumed or a new task is switched in.

Since the scheduler is invoked upon every task arrival, this may lead to a large overhead.

4.1.2 Time-triggered scheduling

In time-triggered systems the scheduling overhead is incurred at periodic intervals. Depending on the period of the scheduler relative to the inter-arrival time of tasks, this can lead to a higher or lower overhead compared with event-triggered scheduling.

Periodic task arrival The time-triggered scheduling models systems where periodic tasks execute a "delay until" kernel routine at the end of their computation, upon which the task is placed in the pending queue (also referred to as the *delay queue*). The periodically invoked scheduler (also referred to as *clock handler* by the literature on tick-scheduling) polls for the release of any task, by inspecting the pending queue. It can be modeled by the following pseudo-code:

now := kT_{clk}
while pending-queue.head.release-time $\leq$ now **do**
insert pending-queue.head in ready-queue
end
$k := k + 1$

Where k is the clock invocation counter. The first line updates the real-time clock. The while loop releases any pending tasks which were scheduled to be released before the current time and places them in the ready queue. The last line selects the next task to run. Figure 3 depicts periodic event handling with tick scheduling.

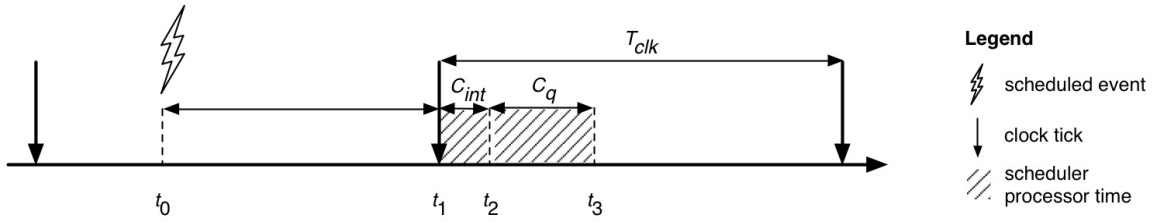

Figure 3: Periodic event handling with tick scheduling

t_0**:** A periodic event was scheduled to occur.

t_1**:** The clock interrupt handler (or scheduler) starts executing, C_{int} representing the fixed overhead of every clock handler invocation, including updating the real-time clock.

t_2**:** All tasks which were scheduled to arrive before the current real-time clock value are moved from the delay queue to the ready queue, taking C_q time. Similarly to the event triggered scheduling, during C_{int} and C_q the interrupt handler may be interrupted by higher priority handlers or blocked by lower priority handlers.

t_3**:** Depending on the task selected to run by the scheduler, either the running task is resumed or a new task is switched in.

Periodic tasks experience a *release delay* between their intended release and the time when they are actually "noticed" by the scheduler, of $0 \leq t_1 - t_0 \leq T_{clk}$.

Sporadic task arrival Tick scheduling is usually used in the context of *periodic* tasks. We can generalize it to include *sporadic* tasks, where upon arrival a sporadic task inserts itself into the pending queue, where it waits until the next scheduler invocation. In this way both periodic and sporadic tasks are handled by the scheduler in the same way. Sporadic tasks experience a *release delay* between the release of the corresponding interrupt handler and the time when they are actually "noticed" by the scheduler, of $0 \leq t_2 - t_1 \leq T_{clk}$.

t_0**:** An event occurs.

t_1**:** The corresponding hardware interrupt handler is dispatched and corresponding job is placed in the *pending* queue, which takes C_p time. The delay between t_0 and t_1 depends on the platform and the operating system.

t_2**:** The clock interrupt handler (or scheduler) starts executing, C_{int} representing the fixed overhead of every clock handler invocation.

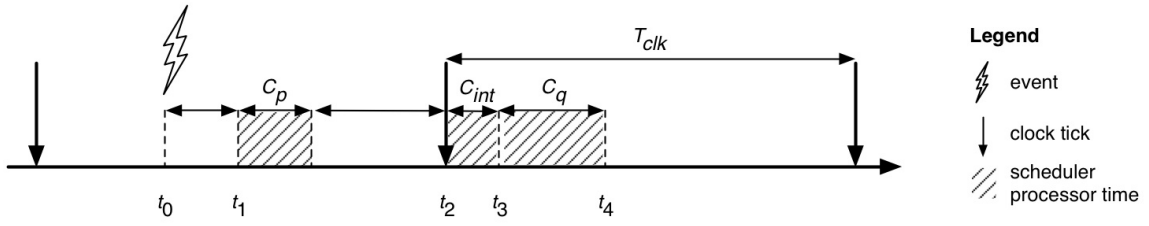

Figure 4: Sporadic event handling with tick scheduling

t_2**:** The clock handler requires C_q time to move all jobs, which have arrived since the last clock tick, from the pending queue to the ready queue.

t_3**:** Depending on the task selected to run by the scheduler, either the running task is resumed or a new task is switched in.

Simplifying tick-scheduling with discrete time model assumption For simplicity reasons, some literature assumes a *discrete time model*, where the time for an event to be registered by the platform is negligible (i.e. for sporadic tasks $t_1 - t_0 = 0$), the environment is discrete and synchronized with the system (i.e. events occur at multiples of clock ticks, and consequently for periodic tasks $t_1 - t_0 = 0$ and for sporadics $t_2 - t_0 = 0$), and the associated jobs can be placed directly into the ready queue (i.e. $C_{int} = C_p = C_q = 0$). In this case the resource requirements can be expressed in terms of the number of clock ticks, i.e. $\varphi_i, C_i, D_i, T_i \in \mathbb{N}$. Note that under the discrete time model, event-triggered and time-triggered scheduling are equivalent.

The analysis based on the discrete time model assumption is inherently optimistic, since in a real system the handler of an event occurring in between two invocations of the scheduler can be delayed for up to almost one clock tick, until the next scheduler invocation. Consequently, if the scheduler does not take the delay into account, the event handler may miss its deadline.

4.1.3 Mix of event- and time-triggered scheduling

Many real-time systems manage a mixture of event triggered sporadic tasks and time-triggered periodic tasks.

4.1.4 Modeling as reduced resource capacity

[Burns et al. 1993b] suggest to reduce the processor capacity to accommodate the real- time clock overhead, but they do not mention by how much. [Buttazzo 2005] suggests a similar approach by adding $U_t = \tau_0/T_0$ to the total utilization of the task set, where τ_0 and T_0 are the worst-case computation time and the period of the clock interrupt handler, and use the utilization based analysis to compute the feasibility of the task set.

4.1.5 Modeling as extended computation time

[Burns and Wellings 2001] includes the C_q overhead of moving the task τ_i from the pending queue to the ready queue in C_i. Since the overhead may depend on the size of the queue, the worst-case overhead must be added to C_i.

They do not take into account the overhead of selecting the highest priority task. If the ready-queue is kept sorted, then the additional overhead is a small constant and can be either ignored or added to C_i.

4.1.6 Modeling as a single high priority task

A simple way to account for the scheduling overhead is to extend the task set with a highest priority task τ_0 with period $T_0 = T_{clk}$ and computation time $C_0 = C_{clk}$, where T_{clk} is the clock period and $C_{clk} =$

$C_{int} + C_q$ is the worst-case computation time required by the clock interrupt handler [Burns et al. 1995, Liu 2000]. It can be then included in the response time analysis by extending Equation 1 with:

$$R'_i = R_i + \left\lceil \frac{R'_i}{T_{clk}} \right\rceil C_{clk} \tag{4}$$

[Tindell and Clark 1994, Audsley et al. 1996] provide a finer grained definition of the C_{clk} term, namely

$$C_{clk} = MC_{int} + MC_{single} + (N - M)C_{multi} \tag{5}$$

where N is the number of actual tasks in the system, C_{int} includes the time needed to update the system clock, C_{single} and C_{multi} are the costs of moving the first and subsequent tasks to the ready queue, and M is the number of clock ticks that fit within the shortest task period T_{min}, given by

$$M = \left\lceil \frac{T_{min}}{T_{clk}} \right\rceil$$

[Burns et al. 1995] present results obtained form the Olympus space satellite case study, where $C_{int} = 16\mu s$, $C_{single} = 88\mu s$, and $C_{multi} = 40\mu s$. Therefore, the clock overhead can have large variation, depending on whether no task is moved to the ready queue and when 20 tasks are moved.

The MC_{int} term in Equation 5 is an example of preemption, since the clock is updated at the highest priority. The remaining terms may be due to blocking, since a lower priority task can be responsible for the delay when the clock handler needs to move it to the ready queue, leading to priority inversion (see Section 5).

4.1.7 Modeling as multiple high priority tasks

Immediately after introducing Equation 4, [Burns et al. 1995] show an example, where this model leads to a very pessimistic utilization bound, since to guarantee schedulability C_{clk} has to be the worst-case scheduling overhead. They present a more detailed model, differentiating between the longest time to move a task to the ready queue C_q and the fixed time C_{int} needed by the scheduler, e.g. for incrementing the clock. They extend the task set with notional tasks corresponding to the real tasks, with the same period and computation time C_q, and extend the response time R_i with:

$$R'_i = R_i + \sum_{j \in \Gamma_p} \left\lceil \frac{R'_i}{T_j} \right\rceil C_q + \left\lceil \frac{R'_i}{T_{clk}} \right\rceil C_{int} \tag{6}$$

where Γ_p is the set of all periodic tasks in the system.

[Burns and Wellings 2001] extend the response time of a task with the overhead due to the interrupt handlers (executing at highest priority) responsible for handling the arrival of *sporadic* tasks, by extending the response time R_i with

$$R'_i = R_i + \sum_{k \in \Gamma_s} \left\lceil \frac{R'_i}{T_k} \right\rceil C_q \tag{7}$$

where Γ_s is the set of sporadic tasks. They ignore the C_{int} overhead, or assume event-triggered scheduling with the C_{int} overhead included in C_q. However, if the additional term in Equation 7 is added to the Equation 6, then the interrupt handler overhead is taken into account.

[Burns et al. 1995] continue refining their model by the number of periodic task arrivals which are noticed by the polling time-triggered scheduler. They claim that the number of times the scheduler must move a periodic task from the delay queue to the ready queue during time interval t is bounded by:

$$K(t) = \sum_{j \in \Gamma_p} \left\lceil \frac{\left\lceil \frac{t}{T_{clk}} \right\rceil T_{clk}}{T_j} \right\rceil \tag{8}$$

assuming $\forall j \in \Gamma_p : T_j < T_{clk}$. They also bound the number of times the scheduler will be invoked during interval t by

$$L(t) = \left\lceil \frac{t}{T_{clk}} \right\rceil \tag{9}$$

They combine these and extend the response time for task τ_i by

$$R'_i = R_i + L(R_i)C_{int} + \min(K(R_i), L(R_i))C_{single} + \max(K(R_i) - L(R_i), 0)C_{multi} \quad (10)$$

where C_{single} and C_{multi} are the costs of moving the first and subsequent tasks to the ready queue, respectively.

[Tindell and Clark 1994] investigate scheduling tasks with release jitter. They come up with the same equation for the time triggered scheduling overhead as in Equation 10, but with the following term for $K(t)$:

$$K(t) = \sum_{j \in hp(i)} \left\lceil \frac{t + J_i}{T_j} \right\rceil \quad (11)$$

where J_i is the release jitter of task τ_i.

4.1.8 Modeling the release delay

In case of time-triggered scheduling, the release delay ($t_2 - t_0$ in Figure 4 and $t_1 - t_0$ in Figure 3) can be modeled by adding the clock handler period T_{clk} to the response time, to cover the case when a pending task is scheduled to arrive just after a clock handler invocation. In event triggered scheduling the release delay is 0.

Note that the question about the granularity of the clock interrupt leads to a trade-off: for a short clock tick the response delay for an event will be short, but the overall overhead due to a frequent clock handler will be larger (assuming C_{int} is independent of the number of pending jobs).

4.1.9 Reducing the scheduling overhead

The scheduling overhead can be reduced in two ways: by reducing the number of scheduler invocations and by reducing the cost of individual scheduler invocations. The latter deals with architectural issues, such as cache and memory management, interconnect, topology etc. and is outside of our scope. In this section we focus on reducing the number of scheduler invocations.

Reducing the number of scheduler invocations In FPPS a higher priority task can arrive at an arbitrary moment in time and the scheduler should allow it to preempt the running task immediately.

In event-triggered scheduling the scheduler is invoked immediately after an event occurred.

In the time-triggered approach, pending tasks are placed in the ready queue and the running task is selected at the frequency of the scheduler task. If the scheduler period is longer than the average inter-arrival time of tasks, then time-triggered approach can reduce the number of scheduler invocations.

In FPNS a higher priority task cannot preempt the running task. If tasks are made non-preemptive by turning off the interrupts, then the whole scheduling overhead is incurred at task boundaries. If the operating system intercepts all interrupts and postpones the scheduling until after the running task has finished, then the task is inserted into the ready queue directly upon arrival, and scheduled only at task boundaries.

4.2 Context switch overhead

The basic assumption states that a context switch occurs at the beginning and end of each preemption. The context switch overhead is charged to the higher priority task. For example, a job i preempting job j will be charged the context switch overhead at the beginning and the end of each preemption. Note that, since we assume jobs cannot suspend them selves, each job preempts *at most one other job.*

4.2.1 Modeling as extended computation time

The most common way of accounting for context switch overhead is to increase the computation time of each job by the overhead. The computation time for task τ_i becomes:

$$C_i := C_i + C_{in} + C_{out}$$

where C_{in} and C_{out} are the *worst-case* context switch times in and out of *any* task [Katcher et al. 1993, Burns et al. 1993a, Audsley et al. 1996], including the cost of storing the registers. In a less general case, [Klein and Ralya 1990, Burns et al. 1993b] assume the in and out overheads are equal and thus use

$$C_i := C_i + 2C_{cs}$$

[Davis 1993, Audsley et al. 1994, Burns et al. 1994] note that this model is pessimistic. When a task is released with a priority *not* higher than the currently running task, then the context switch *into* the preempting task is no longer required, as it is accounted for in the context switch *out* of a higher priority task. [6] [Burns et al. 1994] claim in their case study an average gain of 2.4% in resource capacity by ignoring the obsolete switching overhead into the preempting task.

Another approach is to charge the preempted task (rather than the pre- emptee) [Buttazzo 2005]. The number of context switches is equal to twice the number of times the job will be preempted. This model is equivalent (scheduling wise) to the one charging the preempting task at the beginning and the end.

[Burns et al. 1993a, Gerber and Hong 1993] observe that the context switch out of the job happens after the last observable event. Therefore the context switch out of the job can occur after the deadline. The formula for WR_i becomes

$$WR_i^{n+1} = C_i^D + B_i + \sum_{j \in hp(i)} \left\lceil \frac{WR_i^n}{T_j} \right\rceil C_j^T$$

where C_i^D is the computation time up to the last observable event (includes only the context switch into the job), and C_i^T is the total computation time including both context switches. B_i includes the context switch times for the lower priority tasks blocking τ_i. Note that $C_i^D + C_{out} \leq C_i^T$, because a task τ_i may do some house-keeping activities between the last observable event and its completion (the context switch).

[Gerber and Hong 1993] provide a language with primitives allowing to specify real-time con- straints, and a set of compiler optimizations which aim at reorganizing the code in such a way that the observable events will be executed as early as possible, leaving the unobservable events for after the deadline.

4.2.2 Modeling as wrapping tasks

Context switch overhead can also be expressed as two jobs wrapping the charged job. In case of periodic tasks, the context switch tasks have priorities $i-1$ and $i+1$ and the same phasing as task τ_i.

It is easily seen that under Fixed Priority Scheduling this model corresponds with the extended computation time model for context switch overhead, i.e. the context switches occur at the same time in both models. It also does not influence the design choices.

In the context of hierarchical scheduling, modeling switching overhead as wrapping tasks allows charging the context switch overhead to a different budget, for example the system budget. In case of extended-computation-time model the context switch overhead is always charged to the preempting job. On the down side, the wrapping tasks model introduces two additional tasks per existing task, complicating the schedulability analysis, compared to the extended computation time model.

4.2.3 Reducing the context switch overhead

The context switch overhead can be reduced in two ways: by reducing the number of context switches and by reducing the cost of individual context switches. The latter deals with architectural issues, such as cache and memory management, interconnect, topology etc. and is outside of our scope. In this section we focus on reducing the number of context switches by reducing the number of preemptions.

Reducing number of preemptions on task level A task set can be divided into tasks with timing characteristics defined by the environment (e.g. interrupt arrival time for sporadic tasks), and tasks with some freedom in choosing some of the timing characteristics. The latter tasks can be assigned arbitrary

[6] The lowest priority task never preempts another task, hence the context switch into the lowest priority task can always be ignored.

phasings, as long as their deadlines are met. These unrestricted tasks can be assigned phasing which will reduce the number of preemptions between the tasks.

[Davis 1993] suggest to exploit slack time to reduce the number of preemptions. When a lower priority task is preempted by a higher priority task, and there is slack time available, then the lower priority task can be allowed to continue until the slack is exhausted or the lower priority task has completed, in which case one context switch due to preemption is saved. This is an example of deferred preemption.

Reducing number of preemptions on budget level According to [Bril 2007], budgets are design artifacts and offer freedom in choosing the budget parameters, such as their period, phasing and capacity. Hence preemptions can be avoided by carefully choosing these parameters.

5 Real-time system overheads due to blocking

So far we have regarded only systems where tasks require only a single resource, typically a processor. Now we move to systems where tasks may require additional *logical* resources, leading to more complicated synchronization issues with respect to sharing a single processor. These resources are typically granted on a *non-preemptive and mutually exclusive basis* [Liu 2000]. The part of the task accessing such a resource is referred to as a *critical section* .

A system with *non-preemptive* critical sections can experience a situation first described in [Lampson and Redell 1980] which was later named the *priority inversion* problem [Sha et al. 1990]. It refers to the situation when a high priority task is *blocked* on a shared resource (other than the scheduled processor) locked by a lower priority task, e.g. a system call, where the kernel behaves like a shared resource locked by the task [Burns et al. 1995]. This can lead to "unbounded" blocking time (see the remark below), when middle priority tasks execute after a high priority task blocks, or even deadlock, when several locks are nested in reverse order between the high and low priority tasks.

There are several strategies which aim at controlling priority inversion. They trade off progress (prevent deadlock) against performance (limit the time a high priority task is blocked by a low priority task). All these strategies change the priority of the lower priority task, which holds a lock on the resource blocking the higher priority task. The difference between them lies in when the priority is changed and to which level. In all cases the priority is switched back at the end of the critical section. Note that since the clock interrupt handler does not share critical sections with other tasks, under all protocols below the scheduling task never blocks.

Remark on the "unbounded" priority inversion [Sha et al. 1990, Rajkumar 1991, Klein et al. 1993, Liu 2000] state that the blocking time can be uncontrolled, indefinite, arbitrarily long or unbounded and imply that it can be infinitely long.

> "Unfortunately, a direct application of synchronization mechanisms like the Ada rendezvous, semaphores, or monitors can lead to uncontrolled priority inversion: a high priority job being blocked by a lower priority job for an indefinite period of time." [Sha et al. 1990]

They must implicitly assume that the task set is already *not* schedulable without the critical sections, or the presence of high priority aperiodic tasks.

Otherwise, if we assume a schedulable task set with resource utilization $U \leq 1$ and only periodic and sporadic tasks, the blocking time is bounded by at most the hyper period. This can be shown using the standard example for arguing that priority inversion is unbounded [Sha et al. 1990].

Suppose that τ_1, τ_2, and τ_3 are three tasks arranged in descending order of priority. We assume that tasks τ_1 and τ_3 contend for the same shared resource S. Assume a situation where once τ_3 has obtained a lock on S, τ_1 interrupts τ_3 and subsequently attempts to lock S and blocks. [Sha et al. 1990] argue that task τ_2 may possibly lead to infinitely long blocking of task τ_1.

However, if the utilization of the given task set is $U \leq 1$, then, since tasks τ_1 and τ_3 have non zero computation times, task τ_2 (or in fact any arbitrary subset of "middle" tasks with intermediate priorities) will have utilization of $U_{middle} < 1$. This guarantees that there will be gaps in the execution of the middle tasks. In those gaps τ_3 will be able to make progress and thus eventually unlock S. At that moment τ_1 will continue, since it is ready and has the highest priority.

The remainder of this section deals with the B_i term in Equation 2 and how different priority inversion protocols influence the real-time system overheads.

5.1 Non-preemptive critical sections

In case of non-preemptive critical sections, once a job enters a critical section it will have to complete it before a high priority job can interrupt it. This can be implemented as either

- promoting the low priority job to the highest priority of all tasks as soon as it enters a critical section
- switching off all interrupts until the critical section is completed, including the scheduler timer interrupt.

The introduced delay defines the minimum latency of interrupts and tasks. Note that the amount of time spent in the longest non-preemptable critical section is the shortest scheduling latency that can be guaranteed.

Scheduling overhead If non-preemptive critical sections are implemented by promoting the priority of the task accessing a critical section, one needs to decide which level to promote it to. If it is promoted to the highest level among user-tasks, then the scheduler task with a higher priority than any user-task may still preempt a task in its critical section.

Switching off all interrupts avoids the periodic scheduler overhead, but may also cause a faulty task to fail the entire system. The scheduler overhead is incurred only on critical section boundaries. Also, the system needs an accurate (hardware) clock to update the real-time clock after a critical section has finished.

Context switch overhead Non-preemptive critical sections avoid context switches due to blocking.

5.2 Priority Inheritance Protocol

For a description of the protocol see [Sha et al. 1990].

Scheduling overhead Priority Inheritance Protocol does not influence the scheduling overhead.

Context switch overhead Under priority inheritance protocol a task τ_i can be blocked by at most one critical section of *each* lower priority task τ_j which can block τ_i. Therefore the number of context switches for task τ_i with $N - i$ tasks with priority lower than i is bounded by $2(N - i)$, where N is the total number of tasks (one for context switch into and one out of the blocking task).

5.3 Priority Ceiling Protocol

For a description of the protocol see [Sha et al. 1990].

Scheduling overhead Priority Ceiling Protocol does not influence the scheduling overhead.

Context switch overhead With priority ceiling protocol a job can be blocked by at most a single critical section of a lower priority job [Sha et al. 1990, Rajkumar 1991, Sha et al. 1994]. Therefore the number of context switches due to blocking of task τ_i is bounded by 2: *in* and *out* of the lower priority task locking a critical section with a higher priority ceiling.

5.4 Highest Locker

For a description of the protocol see [Klein et al. 1993].

Scheduling overhead Highest Locker does not influence the scheduling overhead.

Context switch overhead ighest locker protocol guarantees that a job can be blocked at most one time by a lower priority job and that this blocking can occur only at the beginning, before the job starts executing, which is already included in the context switch cost due to preemption. Therefore the context switch overhead due to blocking can be ignored.

5.5 Stack Resource Policy

For a description of the protocol see [Baker 1991].

Scheduling overhead Stack Resource Policyr does not influence the scheduling overhead.

Context switch overhead Since no job can be blocked after it starts [Baker 1991, Buttazzo 2005], the context switch overhead due to blocking can be ignored. All the context switches are accounted for during preemption.

5.6 Adjusting the response time analysis due to tick-scheduling

As mentioned in Section 2.2, tick-scheduling gives rise to a delay between the time an event occurs and the time it's handler starts executing. So far we have ignored this extra overhead in the discussion, which assumed event-triggered scheduling. [Liu 2000] suggests to include the B and C overhead in the analysis by extending the worst-case response time analysis for task τ_i with the following:

1. add the time needed to move each *higher* priority job from the pending to the ready queue
2. add the time needed to move each *lower* priority job from the pending to the ready queue
3. adjust the blocking time due to non-preemptive critical sections to an integral number of clock periods (see Section 5)

Moving higher priority jobs form the pending queue to the ready queue can be modeled by extending the computation time of every higher priority task $\tau_j \in hp(i)$ by C_p, where C_p is the time needed to move a single job from the pending queue to the ready queue.

Moving a lower priority job $\tau_j \in lp(i)$ form the pending queue to the ready queue can be modeled by adding a task τ_k to the set of higher priority tasks, with $T_k = T_j$ and $C_k = C_p$

Let θ_k be the maximum execution time of critical sections in lower priority task τ_k, and let us assume that $B_i = \theta_k$. To adjust the blocking time due to lower priority jobs to an integral number of clock periods, we substitute B_i with:

$$B_i' = T_{clk} + \left\lceil \max_{k \in lp(i)} \theta_k / T_{clk} \right\rceil T_{clk}$$

The first T_{clk} term represents the delay when a job of τ_i arrives just after the $(x-1)$th clock interrupt. The second term adjusts the blocking time due to lower priority jobs by considering the situation when a lower priority task arrives just before the xth clock interrupt and finishes just after yth clock interrupt $(x \leq y)$, and the job of τ_i has to wait for an additional $(y-x)T_{clk}$ between xth and yth clock interrupt and T_{clk} after the yth clock interrupt, as shown in Figure 5.

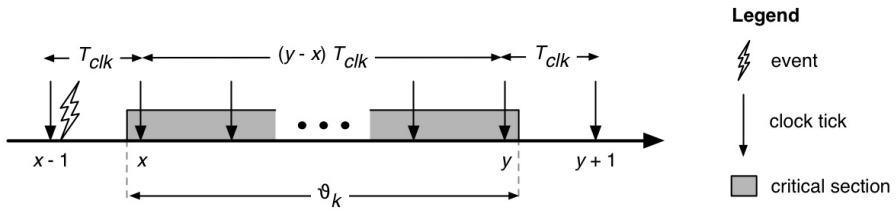

Figure 5: Overhead due to tick-scheduling

6 Conclusion

We have presented a literature overview dealing with system overheads in the context of Fixed Priority Preemptive Scheduling on a single timely resource. We have focused on the cost of scheduling and context switching. We presented different modeling techniques, covering the number and duration of the overheads and accountability for the overhead, and techniques for reducing the overheads. Given experimental and analytical results of worst-case system overheads, the research into these overheads is still relevant for hard real-time systems, whenever high utilization is required.

6.1 Future work

In this paper we have focused on traditional non-hierarchical FPPS. It would be interesting to investigate system overheads in the context of hierarchical systems, in particular how the scheduling and context switch overheads influence the granularity of the budgets.

We have limited our discussion to FPPS scheduling on a single CPU resource. We expect that scheduling and context switching overheads are even more relevant in multi resource systems and leave it for future research.

In this paper we have looked into some practical aspects of real-time scheduling. We have assumed FPPS, where jobs can be preempted at arbitrary moments in time. However, this is not the case in real systems, where some subjobs, e.g. system calls, are not preemptable. This calls for an alternative to FPPS and Fixed Priority with Deferred Preemption seems a promising approach. There is the question of how the system overheads influence the granularity of non-preemptable subjobs, and how to deal with situations where it can be more efficient to avoid a context switch by letting a subjob execute non-preemptively.

7 Acknowledgments

We would like to thank Reinder Bril, Johan Lukkien and Pieter Cuijpers for the insightful discussions and their comments on earlier versions of this document.

References

[Audsley et al. 1993] N. Audsley, A. Burns, M. Richardson, K. Tindell, A. J. Wellings. *Applying new scheduling theory to static priority pre-emptive scheduling.* Software Engineering Journal, vol. 8(5):pp. 284–292, 1993.

[Audsley et al. 1996] N. C. Audsley, I. J. Bate, A. Burns. *Putting fixed priority scheduling theory into engineering practice for safety critical applications.* In *RTAS '96: Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, p. 2. IEEE Computer Society, Washington, DC, USA, 1996.

[Audsley et al. 1995] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, A. J. Wellings. *Fixed priority pre-emptive scheduling: an historical perspective.* Real-Time Syst., vol. 8(2-3):pp. 173–198, 1995.

[Audsley et al. 1994] N. C. Audsley, R. I. Davis, A. Burns. *Mechanisms for enhancing the flexibility and utility of hard real-time systems.* In *Proceedings of the Real-Time Systems Symposium, 1994*, pp. 12–21. 1994.

[Baker 1991] T. P. Baker. *Stack-based scheduling for realtime processes.* Real-Time Syst., vol. 3(1):pp. 67–99, 1991.

[Bini et al. 2003] E. Bini, G. C. Buttazzo, G. M. Buttazzo. *Rate monotonic analysis: The hyperbolic bound.* IEEE Trans. Comput., vol. 52(7):pp. 933–942, 2003.

[Bril 2007] R. J. Bril. *Towards pragmatic solutions for two-level hierarchical scheduling. part i: A basic approach for independen applications.* Tech. Rep. CS-07-19, Eindhoven University of Technology, 2007.

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

[Burns et al. 1995] A. Burns, K. Tindell, A. Wellings. *Effective analysis for engineering real-time fixed priority schedulers.* IEEE Trans. Softw. Eng., vol. 21(5):pp. 475–480, 1995.

[Burns et al. 1993a] A. Burns, K. Tindell, A. J. Wellings. *Fixed priority scheduling with deadlines prior to completion.* Tech. Rep. YCS 212, University of York, 1993a.

[Burns et al. 1994] A. Burns, A. Wellings, C. H. Forsyth, C. M. Bailcy. *A performance analysis of a hard real-time system.* Tech. Rep. YCS-94-224, Department of Computer Science, University of York, UK., 1994.

[Burns and Wellings 2001] A. Burns, A. J. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Burns et al. 1993b] A. Burns, A. J. Wellings, A. D. Hutcheon. *The impact of an ada run-time system's performance characteristics on scheduling models.* In *Ada-Europe '93: Proceedings of the 12th Ada-Europe International Conference*, pp. 240–248. Springer-Verlag, London, UK, 1993b.

[Butel et al. 2004] P. Butel, G. Habay, A. Rachet. *Managing partial dynamic reconfiguration in virtex-ii pro fpgas.* Tech. rep., Xilinx, 2004.

[Buttazzo 2005] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* 2nd ed. Springer, 2005.

[David et al. 2007] F. M. David, J. C. Carlyle, R. H. Campbell. *Context switch overheads for linux on arm platforms.* In *ExpCS '07: Proceedings of the 2007 workshop on Experimental computer science*, p. 3. ACM, New York, NY, USA, 2007.

[Davis 1993] R. Davis. *Scheduling slack time in fixed priority pre-emptive systems.* Tech. Rep. YCS-93-217, Department of Computer Science, University of York, UK, 1993.

[Echague et al. 1995] J. Echague, I. Ripoll, A. Crespo. *Hard real-time preemptively scheduling with high context switch cost.* ecrts, vol. 00:p. 184, 1995.

[Gabriels and Gerrits 2007] R. Gabriels, D. Gerrits. Accounting for overhead in fixed priority preemptive scheduling, 2007.

[Gerber and Hong 1993] R. Gerber, S. Hong. *Semantics-based compiler transformations for enhanced schedulability.* In *Proceedings of the Real-Time Systems Symposium, 1993*, pp. 232–242. 1993.

[Joseph and Pandya 1986] M. Joseph, P. Pandya. *Finding Response Times in a Real-Time System.* The Computer Journal, vol. 29(5):pp. 390–395, 1986.

[Katcher et al. 1993] D. I. Katcher, H. Arakawa, J. K. Strosnider. *Engineering and analysis of fixed priority schedulers.* IEEE Trans. Softw. Eng., vol. 19(9):pp. 920–934, 1993.

[Klein and Ralya 1990] M. H. Klein, T. Ralya. *An analysis of input/output paradigms for real-time systems.* Tech. Rep. CMU/SEI-90-TR-19, Carnegie-Mellon University, 1990.

[Klein et al. 1993] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, M. G. Harbour. *A practitioner's handbook for real-time analysis.* Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[Lampson and Redell 1980] B. W. Lampson, D. D. Redell. *Experience with processes and monitors in mesa.* Commun. ACM, vol. 23(2):pp. 105–117, 1980.

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

[Liu 2000] J. W. S. W. Liu. *Real-Time Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[Mok 1983] A. K. Mok. *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment.* Ph.D. thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983.

[Rajkumar 1991] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach.* Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[Schoen et al. 2000] F. Schoen, W. Schroeder-Preikschat, O. Spinczyk, U. Spinczyk. *On interrupt-transparent synchronization in an embedded object-oriented operating system.* In *ISORC '00: Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, p. 270. IEEE Computer Society, Washington, DC, USA, 2000.

[Sha et al. 1990] L. Sha, R. Rajkumar, J. P. Lehoczky. *Priority inheritance protocols: An approach to real-time synchronization.* IEEE Trans. Comput., vol. 39(9):pp. 1175–1185, 1990.

[Sha et al. 1994] L. Sha, R. Rajkumar, S. S. Sathaye. *Generalized rate-monotonic scheduling theory: a framework for developing real-time systems.* Proceedings of the IEEE, vol. 82(1):pp. 68–82, 1994.

[Tindell and Clark 1994] K. Tindell, J. Clark. *Holistic schedulability analysis for distributed hard real-time systems.* Microprocess. Microprogram., vol. 40(2-3):pp. 117–134, 1994.

[Tsafrir 2007] D. Tsafrir. *The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops).* In *ExpCS '07: Proceedings of the 2007 workshop on Experimental computer science*, p. 4. ACM, New York, NY, USA, 2007.

[Zhang and West 2006] Y. Zhang, R. West. *Process-aware interrupt scheduling and accounting.* In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pp. 191–201. IEEE Computer Society, Washington, DC, USA, 2006.