Hybrid-priority Scheduling of Resource-sharing Sporadic Task Systems*

Sanjoy Baruah
The University of North Carolina

Nathan Fisher
Wayne State University

Abstract

A hybrid scheduling algorithm is proposed, which integrates features of the Fixed Priority (FP) and Earliest Deadline First (EDF) scheduling policies. It is shown that this hybrid scheduling algorithm is a generalization of both FP and EDF, and tends to retain most of the desirable properties and features of both individual policies. An exact (i.e., necessary and sufficient) test is derived for the preemptive uniprocessor scheduling of resource-sharing sporadic task systems using this hybrid scheduling algorithm, with access to shared resources arbitrated using the Stack Resource Policy (SRP).

1 Introduction

There has been much discussion within the real-time systems research community [23, 9] between the relative merits of Fixed Priority (FP) and Earliest Deadline First (EDF) scheduling. The pros and cons of each policy have been listed and thoroughly discussed over the years — some of the issues that have come up in this ongoing debate are summarized in Section 2 below.

Since both scheduling policies appear to have benefits and drawbacks, it would be beneficial to combine both into a single comprehensive scheduling framework. This is not a new insight, e.g., Zuberi et al. [29], and González Harbour and Palencia Gutiérrez [16], study scheduling policies based on such integration of FP and EDF. In this paper, we propose, and analyze, a hybrid scheduling policy that incorporates elements of both FP and EDF scheduling in the context of the preemptive uniprocessor scheduling of resource-sharing sporadic task systems (this task and platform model are formally defined in Section 3). We argue (Section 2) that such integration of FP and EDF combines the best features of both policies. We then derive (in Section 4) an exact (i.e., necessary and sufficient) schedulability analysis algorithm for systems scheduled according to this hybrid scheduling policy, and demonstrate that this analysis generalizes the best previously-known FP and EDF schedulability analysis algorithms. We place this work in the context of some other related research in Section 6.

*Supported in part by NSF Grant Nos. CNS-0408996, CCF-0541056, and CCR-0615197, ARO Grant No. W911NF-06-1-0425, and funding from the Intel Corporation.

2 Background and Motivation

The debate among advocates of EDF and those of FP scheduling has been long and contentious. Among the benefits claimed for FP scheduling by its advocates are the following:

1. By corresponding to the actual *importance* of a task in the overall system being designed, priorities can be made to represent a relevant property of the system. Hence priority-based scheduling exploits further knowledge of the application, and is therefore a more accurate representation of application semantics.

2. Embedded systems programmers have traditionally programmed with priorities: the FP model is simply providing them with what they desire. This helps move results on FP scheduling out of academic environments to the “real world” by making it more likely that programmers will actually make use methodologies and tools based on these results; in contrast, many potentially useful EDF results remain unused by actual programmers. This point of view was stated succinctly in 2002 by Joe Gwinn of Raytheon as follows [15]: *In the embedded realtime world, a commercial RTOS or equivalent lacking hard (winner-take-all) realtime priorities has always been dead on arrival.*

3. Although FP scheduling is not optimal in the sense that EDF is (see below), FP scheduling need not result in too large a loss of processor utilization. For instance, for systems of Liu and Layland (LL) tasks [21], the worst-case utilization bound of FP scheduling under rate-monotonic priority assignment is ≈ 0.69 [21], and the average-case bound is almost 0.88 [19].

On the other hand, advocates of EDF scheduling claim the following benefits for their preferred scheduling policy:

1. EDF offers superior utilization of the processor: every system that is FP schedulable is also EDF schedulable while the converse is not true.

2. Since EDF is known to be optimal on a preemptive uniprocessor [21, 13], it is often easier to prove schedulability of EDF-scheduled systems by exploiting this optimality. This follows from the observation that rather than needing to trace the behavior of EDF on a given system, it suffices to demonstrate the existence of a schedule –not necessarily an EDF-generated one– for the system, since any system for which a schedule exists is guaranteed to be scheduled correctly by EDF. (Of course, since such claims are typically made by advocates of EDF, it can be argued that such ease of proof is illusory, in that it lies in the eye of the beholder.)

3. The claim that there exist priority assignments under FP scheduling (such as rate-monotonic) that result in only a relatively small loss of processor utilization is a bit of a red herring, since it contradicts the claimed benefit that FP scheduling allows priorities of jobs to reflect their semantic (real-world) importance. It would be a happy coincidence if tasks in a system happened to be ranked by importance in rate-monotonic order; otherwise, rate-monotonic assignment of priorities ceases to be a meaningful reflection of application semantics, and becomes a scheduling artifact (just like deadlines for EDF). But in that case EDF should be preferred, for the superior utilization/ optimality reasons outlined above.

4. As Greg Bollella, who led the effort that developed the Real-Time Specification for Java (RTSJ) under the Java Community Process points out [5], *Priorities are a good* dispatching *metric but not[...] a good way for an application to express its temporal demands. Applications should express their temporal requirements in terms of cost, deadline, period, and minimum interarrival times. The* system *is more properly the component which should determine the execution order*.

5. In any case, using what Gwinn [15] elegantly describes as "winner-take-all" priorities to reflect semantic importance may not be too meaningful, particularly in hard-real-time systems. After all, what does it mean to denote one task as being more important than another if it is imperative that $\underline{\text{all}}$ deadlines must be met? Even in soft/ firm real-time systems, notice that assigning a task a higher priority than others implies that meeting a single deadline of the higher-priority task is more important than meeting any number of deadlines of lower-priority tasks. While there are indeed applications for which such an extreme winner-take-all interpretation of priorities are justified for some tasks (e.g., all functionalities associated with the anti-lock brakes in a car can be assigned greater priority than all functions associated with the audio entertainment system), we do not believe that applications are common with semantics that mandate such a total ordering of *all* their constituent tasks.

While each of the above reasons, in favor of FP and of EDF scheduling, were well-founded when they were first made, we believe that things have changed recently, and that some of these points are somewhat outdated:

1. As real-time systems become more important and pervasive, they are increasingly coming to be developed by general-purpose programmers. Such programmers have no particular fondness of, or desire for, the priority concept (unlike traditional embedded-systems developers, who have tended to program at a relatively low level close to the hardware). Hence, it is no longer as important as before that a priority mechanism be provided by a scheduling policy for the programmers' comfort, or to facilitate adoption of the policy.

2. On the other hand, the rapidly falling price of computing capacity no longer makes it imperative that every last bit of utilization be squeezed out of processors.

3. While superior tool and methodology support (e.g. the rate-monotonic software methodology [18] strongly supported by CMU's Software Engineering Institute and the real-time systems group at York[1]) have in the past made it easier to use FP scheduling, this advantage of FP scheduling is fast diminishing. For instance,

   (a) While the Ada95 Real-Time Systems Annex had only defined a complete facility for FP scheduling, Ada 2005 includes native support for EDF.

   (b) It has been shown [28] how EDF may be efficiently implemented within the Real-Time Specification for Java (RTSJ).

   (c) The York group is currently developing very sophisticated libraries that offer support for EDF scheduling, for public release and distribution.

Taking all the above factors into account, our opinion is that given a choice, priorities should be used rarely in systems design and only when they unequivocally and accurately represent system semantics. In particular, FP scheduling makes most sense when only parts of the system are hard-real-time, and it is possible to identify priorities ("importances") for tasks such that meeting a single deadline of the higher-priority task is more important than meeting any number of deadlines of lower-priority tasks. Even

[1]See http://www.cs.york.ac.uk/rts/

in systems when FP scheduling is used, we expect that a typical system will have relatively few distinct priorities as compared to the number of tasks.

Proposed approach. We consider the scheduling of collections of *sporadic* tasks [22, 4] upon a shared platform comprised of a single preemptive processor and some additional shared resources that may be accessed by at most one task at any given time and that are serially reusable. For such systems, we propose a *hybrid* scheduling policy, which integrates features of FP and EDF, with access to shared resources arbitrated using Baker's Stack Resource Policy [1]. We assume that there are a few (say, $p \geq 1$) distinct priority levels, which are totally ordered with respect to each other: as long as jobs in one priority are awaiting execution, no lower-priority jobs may execute. We assume that there are n tasks[2], with each task assigned to one of the p priorities; within each priority, tasks are scheduled using EDF.

Our expectation is that the system designer, who is responsible for determining the parameters that characterize the sporadic tasks, does the priority assignment. (Hence, we consider priorities to reflect the designer-determined *importance* of the task, instead of looking on them as a scheduling artifact that is manipulated by the scheduling framework to maximize the likelihood of meeting deadlines, as, e.g., the rate-monotonic or deadline-monotonic policies do,) Observe that in the absence of any priority information – e.g., when the system designer considers all tasks equally important, (or in any event is unwilling to deem even a single job of one task more important than all the jobs of all lower-priority tasks), the designer could simply use the optimal EDF algorithm by assigning all the tasks the same priority. At the other extreme, a designer wishing to emulate FPS (on a platform with at least as many priority levels as there are tasks: $p \geq n$) may do so by assigning at most one task to each priority level.

3 Formal Model

For the purposes of this paper, hybrid-priority sporadic task system τ is comprised of n sporadic tasks τ_1, τ_2, ..., τ_n, and a priority function pri. Each task τ_i is characterized by a worst-case execution requirement C_i, a relative deadline D_i, and an inter-arrival separation parameter T_i (also known as the period of the task). Each such task generates a potentially infinite sequence of jobs. Task τ_i's first job may arrive at any instant, and subsequent job arrivals are separated by at least T_i time units. Each such job has an execution requirement of at most C_i units, which must complete by a deadline that is D_i time-units after its arrival time.

There are $p \geq 1$ distinct *priority levels*, and *priority function pri* maps each task to one of these p priority levels, with the interpretation that $pri(i) = \ell$ if and only if τ_i is assigned the ℓ'th-highest priority level[3]. All jobs generated by task τ_i are assigned priority level $pri(i)$). The relative priorities of jobs is determined according to their priority levels — the smaller the value, the greater the priority; if two jobs have the same priority level, the one with the earlier (absolute) deadline is accorded greater priority; if both priority level and deadline are equal, the tie may be broken arbitrarily but consistently (i.e., always in favor of the same task's job).

The execution platform is comprised of a single preemptive processor and m non-preemptable serially reusable resources $R_1, R_2, \ldots, R_m$. Without loss of generality, we assume that the processor is of unit computing capacity. That is, any job completes one unit of execution by executing on the processor for one time unit. The resource requirements of the sporadic tasks may be specified in many ways (see, e.g., [1, 20, 24]); here we let C_{jk} denote the maximum length of time for which each job of τ_j may hold some resource that is also needed by τ_k's jobs ($j \neq k$).

We define the *utilization* $U(\tau_i)$ of τ_i to be the ratio of the task's execution requirement to its period: $U(\tau_i) \overset{\text{def}}{=} C_i/T_i$, and the *system utilization* $U(\tau)$ to be the sum if the utilizations of all the tasks: $U(\tau) = \sum_{\tau_i \in \tau} U(\tau_i)$. The system utilization $U(\tau)$ denotes the maximum fraction of the time for which the processor could be required to be executing; consequently, $U(\tau) \leq 1$ is a trivial necessary condition for τ to schedulable under any uniprocessor scheduling policy. We find it useful to consider as a special sub-case those sporadic task systems whose system utilizations are a priori bounded by a constant strictly less than one:

Definition 1 (bounded-utilization systems) *A bounded-utilization sporadic task system is one whose system utilization is a priori bounded from above by a known constant $c < 1$.*

By definition, any given sporadic task may generate infinitely many different sequences of jobs; by extension, so may any given sporadic task system. One particular job arrival sequence, which we will refer to as the synchronous arrival sequence, has proven to be of particular interest in prior research into sporadic task systems:

Definition 2 (synchronous arrival sequence) *With respect to a specified time-instant t_o, the* synchronous arrival sequence of jobs at *t_o for a sporadic task τ_i consists of the first job of τ_i arriving at t_o, and subsequent jobs arriving*

[2]Although the relationship between the values of n and p does not play a role in the algorithms we develop, we are implicitly assuming that n is typically far greater than p ($n \gg p$).

[3]We stick with the convention of having smaller numbers represent greater priority: tasks assigned priority level one have the greatest priority, and tasks assigned priority level p have the least priority.

Let $\tau_1, \tau_2, \ldots, \tau_n$ denote the tasks, and $R_1, R_2, \ldots, R_m$ denote the non-preemptable resources. Tasks are assumed to be indexed according to non-decreasing priority levels, and according to relative deadlines within the same priority level (as explained in Condition 1).

1. Each resource R_j is statically assigned a *ceiling* $\Pi(R_j)$, which is set equal to index of the lowest-indexed task that may access it:

$$\Pi(R_j) = \min\{i \mid \tau_i \text{ accesses } R_j\}$$

2. A *system ceiling* is computed each instant during run-time. This is set equal to the minimum ceiling of any resource that is currently being held by some job.
3. At any instant in time, a job generated by τ_i may begin (or resume) execution only if it is the active job of greatest priority, and i is strictly less than the system ceiling.

Figure 1. Hybrid-priority scheduling with the SRP

exactly T_i time units apart. The synchronous arrival sequence at t_o for a collection of tasks consists of the union of the synchronous arrival sequences at t_o of each individual task in the collection of tasks.

The Stack Resource Policy (SRP). The Stack Resource Policy (SRP) was proposed by Baker [1] for arbitrating access to shared non-preemptable serially reusable resources in uniprocessor real-time systems. SRP may be used in conjunction with priority-based scheduling algorithms in which the priority assigned to each job does not change between the job's arrival and the instant that it completes execution (this includes FP scheduling as well as EDF). It is shown in [1] that SRP-scheduled systems are deadlock free.

As shown in [1], SRP leverages a total ordering which can be defined on the tasks in the task system, depending upon whether a task may or may not preempt another. To facilitate the presentation of SRP in the context of hybrid-priority scheduling let us assume, without loss of generality, that ***tasks are indexed in decreasing order of priority and by increasing relative-deadline parameter within the same priority level***:

$$(i < j) \equiv (pri(i) < pri(j)) \bigvee \Big((pri(i) = pri(j)) \bigwedge (D_i \leq D_j)\Big) \quad (1)$$

(Thus, τ_1 is the task with the smallest relative deadline from among all those that are assigned the greatest –i.e., lowest-numbered– priority, and τ_n is the task with the largest relative deadline from among all those that are assigned the lowest priority.)

Suppose that a job of some task τ_i is executing at some instant in time. Observe that if a newly arrived job of some task τ_k is given priority over this job of τ_i, it must be the case that $k < i$, since either **(i)** the priority level of τ_k is strictly less than that of τ_i: $(pri(k) < pri(i))$, or **(ii)** the priority levels are the same, but τ_k's job has an earlier deadline (which can only be if τ_k has a smaller relative parameter): $((pri(k) = pri(i))$ and $(D_k < D_i))$.

In order to be able to arbitrate resource access using the SRP, it must a priori be known which tasks access which shared non-preemptable serially reusable resources.

- For each such shared resource R_j, let its *ceiling* $\Pi(R_j)$ denote the index of the smallest-indexed task that may access the resource.
- At each time-instant t during run-time, let *system ceiling* $\Pi(t)$ denote the minimum from among the ceilings of all resources that are locked at time-instant t.

Under the SRP, a job is selected to begin execution (or resume execution, if it had previously executed and been preempted) at time-instant t if and only if it is the job of greatest priority awaiting execution, and the index of the task that generated it is strictly less than the system ceiling $\Pi(t)$. (This is represented in pseudo-code form in Figure 1.) A job that is not executing while some job of lesser priority is executing is said to be *blocked* by the lesser-priority job, which is referred to as the *blocking* job.

Ending Critical Section (ECS). With respect to fixed-priority scheduling, more accurate schedulability tests can often be designed (see, e.g., [8, 12]) by giving special consideration to jobs that *end* in a critical section – i.e., jobs of tasks in which the last executable line of code is the release of the lock on a non-preemptable serially reusable resource. We will refer to such a critical section in a job, if one exists, as an *Ending Critical Section (ECS)* of the job.

4 Conditions for unschedulability

In this section we derive (Theorem 1 below) an exact – i.e., both necessary and sufficient– condition that must be satisfied by any hybrid-priority sporadic task system that is unschedulable. Our approach towards developing this test is as follows. We obtain (Lemma 6) conditions that must necessarily be satisfied by any hybrid-priority task system that may miss a deadline. We then show that these conditions are also sufficient for a deadline miss by demonstrating a job arrival sequence for any system satisfying these conditions that must necessarily miss a deadline, thereby demonstrating that they represent an exact schedulability condition.

Suppose that hybrid-priority task system τ is not schedulable. By definition, there exist collections of jobs that could legally be generated by τ, upon which deadline misses occur. We will now study a specific such collection of jobs, characterized as follows. Consider the greatest (i.e., lowest-numbered) priority level at which a deadline miss can occur, and a collection of jobs of *minimum* cardinality $\mathcal{J}$ of τ on which a deadline miss occurs at this priority level. Let t_f denote the time-instant of the earliest deadline miss in $\mathcal{J}$'s schedule. Suppose that the job that misses its deadline at t_f is generated by task τ_i. Let t_o denote the earliest arrival time of any job in $\mathcal{J}$ with priority level $< pri(i)$, or priority level $= pri(i)$ and (absolute) deadline $\leq t_f$.

An *idle instant* is said to occur in a schedule generated by our hybrid algorithm (Figure 1) at time-instant t if the processor is idled at t.

Lemma 1 *There are no idle instants in the interval* $[t_0, t_f)$*, in* $\mathcal{J}$*'s schedule.*

Proof Sketch: Because SRP is guaranteed deadlock free [1], it must be the case that all jobs that arrived prior to an idle instant t have completed execution by t. Removing from $\mathcal{J}$ all the jobs that completed execution prior to any idle instant would not affect the schedule after the idle instant; the deadline miss in the resulting reduced collection of jobs contradicts the claimed minimum cardinality of $\mathcal{J}$. ■

Definition 3 (blocking execution) *Consider the schedule for* $\mathcal{J}$*. Within the context of this schedule, a* blocking execution *is a maximally contiguous chunk of execution of any job within the interval* $[t_o, t_f)$*, that either has priority level* $> pri(i)$*, or priority level equal to* $pri(i)$ *and absolute deadline* $> t_f$*.*■

Lemma 2 *The schedule for* $\mathcal{J}$ *contains at most one blocking execution. If such a blocking execution exists, it begins at time-instant* t_o*.*

Proof: The proof is by contradiction: we assume that there are several such blocking executions, and demonstrate how we could obtain a collection of jobs of smaller cardinality than $\mathcal{J}$ which also misses a deadline.

Consider the instant t_s ($t_s > t_o$) at which the last of these blocking executions begins executing in the schedule. Let J_j denote the job which is being executed in this blocking execution, and τ_j the task that generates it. By the rules of SRP, it must be the case that

1. J_j is the job with greatest priority that is awaiting execution at time-instant t_s; and
2. at this time-instant, the system ceiling $\Pi(t_s) > j$ (i.e., no jobs of $\{\tau_1, \tau_2, \ldots, \tau_j\}$ are blocked at time-instant t_s).

Suppose that job J_j has executed for x time units prior to time-instant t_s. Consider the collection of jobs $\mathcal{J}'$ obtained from $\mathcal{J}$, by *(i)* removing all jobs that arrived prior to time-instant t_s (including J_j), and *(ii)* adding a job J'_j of τ_j that arrives at time-instant $(t_s - x)$. We make the following observations about the state of job J'_j in $\mathcal{J}'$'s schedule at time-instant t_s:

1. Job J'_j will have executed for x time-units prior to t_s, hence, its remaining execution at time-instant t_s in the schedule of $\mathcal{J}'$ is the same as that of job J_j in the schedule of $\mathcal{J}'$, at the same instant.
2. Since the absolute deadline of J'_j is $\geq$ the absolute deadline of J_j while the priority levels of both jobs is the same, the priorities of all jobs relative to each other in $\mathcal{J}$'s schedule at t_s is identical to the priorities of all jobs relative to each other in $\mathcal{J}'$'s schedule at t_s (with job J'_j replacing job J_j).

Therefore, the schedule over $[t_s, t_f)$ remains the same for $\mathcal{J}'$ and $\mathcal{J}'$ (with job J'_j replacing job J_j).

But $\mathcal{J}'$ contains strictly fewer jobs than $\mathcal{J}$, since jobs in $\mathcal{J}$ that had arrived prior to t_s are not included in $\mathcal{J}'$ (there must be at least one such job, since we have assumed that there are multiple blocking executions in $\mathcal{J}$'s schedule, of which the one beginning at t_s is only the last). This contradicts our assumption that $\mathcal{J}$ is a collection of jobs of minimum cardinality to miss a deadline. ■

For notational convenience, let us set t_o to zero[4]. As is conventional, we will use the term *interference* to denote the amount that tasks with priority-level less than $pri(i)$ execute over $[0, t_f)$, thereby preventing τ_i from executing. More specifically, for each τ_k such that $pri(k) < pri(i)$, let the interference I_k denote the amount that jobs of τ_k execute over $[0, t_f)$.

As a consequence of Lemmas 1 and 2, it follows that in order for a job of τ_i to miss a deadline at time-instant t_f upon a minimum-cardinality collection of jobs $\mathcal{J}$ legally generated by task system τ,

1. There are no idle instants in $[0, t_f)$;
2. There is at most one blocking execution over this interval. If there is such a blocking execution, then it begins at zero, and it is part of a job generated by a task τ_j with priority level $pri(j) > pri(i)$, or with priority level $pri(j) = pri(i)$ and relative deadline parameter greater than t_f: $D_j > t_f$.
3. All other jobs that execute over $[0, t_f)$ have priority level strictly less that $pri(i)$, or priority level equal to $pri(i)$ and deadline $\leq t_f$.

We thus see that $\mathcal{J}$'s schedule consists of *(i)* zero or one blocking executions; *(ii)* interference by greater-priority tasks — i.e., executions of jobs of priority level less than $pri(i)$ (collectively represented by the term $\sum_{k:pri(k)<pri(i)} I_k$); and *(iii)* executions of jobs of priority

[4]While this may require the job that generates the blocking execution to arrive at a negative time-instant, there is no loss of generality since the entire time line could be considered shifted rightwards).

level $pri(i)$ and deadlines $\leq t_f$. In Lemmas 3, 4 and 5 below, we provide upper bounds on each of these three terms.

Definition 4 (blocking function) *For any priority level ℓ and any interval-length L, the blocking function $B(\ell, L)$ is defined as follows:*

$$\begin{aligned} B(\ell, L) = \max\{C_{j,k} \mid \\ [(pri(j) > \ell) \textit{ or } ((pri(j) = \ell) \textit{ and } (D_j > L))] \\ \textbf{and } [(pri(k) < \pi) \textit{ or } ((pri(k) = \pi) \textit{ and } (D_\ell \leq L))]\} \quad (2) \end{aligned}$$

Informally, $B(\ell, L)$ represents the largest blocking that could be experienced by a job of priority level ℓ and relative deadline $\leq L$, by either a job of lower priority level (i.e., of a priority level $> \ell$), or a job of the same priority level (i.e., of priority level ℓ) but relative deadline $> L$.

Lemma 3 *The duration of the blocking execution in $\mathcal{J}$'s execution over $[0, t_f)$ is bounded from above by $B(pri(i), t_f)$.*

Proof: Follows directly from Definition 4 — the blocking execution must belong to a job J_j of a task τ_j such that either $pri(j) > pri(i)$, or $pri(j) = pri(i)$ and $D_j > t_f$. And in order for it to be able to block under the SRP, it is necessary that it hold a resource needed by some task τ_ℓ, such that $pri(\ell) < pri(i)$, or $pri(\ell) = pri(i)$ and $D_\ell \leq t_f$. ■

Lemma 4 *The cumulative execution requirement of all the jobs of priority level $pri(i)$ in $\mathcal{J}$ with arrival times and deadlines in $[0, t_f)$ is bounded from above by the cumulative execution requirement of the synchronous arrival sequence of all tasks in priority level $pri(i)$, starting at time-instant 0, over $[0, t_f)$.*

Proof: If any such job can have its arrival-time moved earlier, it can only contribute more to the cumulative execution requirement. ■

Lemma 5 *The total execution by the jobs in $\mathcal{J}$ of priority greater than τ_i's over $[0, t_f)$ is bounded from above by the amount that the synchronous arrival sequence of all jobs in priority levels $1, \ldots, (pri(i) - 1)$, starting at time-instant 0, would execute over $[0, t_f)$.*

Proof Sketch: This is proved in essentially the same manner as the "critical instant" proof in pure FP scheduling (see, e.g., [10]) for a textbook description), by demonstrating that the total execution cannot decrease if any such higher-priority job has its arrival-time moved forward. We provide a few additional details below, for the sake of completeness.

We distinguish between two separate cases, depending upon whether or not τ_i's job is holding a lock at time-instant t_f.

1. Suppose that τ_i's job is not holding a lock at time t_f. In that case, it cannot possibly be blocking the execution of any job of any task τ_k that has $pri(k) < pri(i)$ — all the jobs of such tasks that arrived during $[0, t_f)$ would execute before τ_i's job could resume execution. Hence, I_k for each such τ_k is maximized when each job arrives as soon as legally permitted to do so – i.e., under the synchronous arrival sequence at time-instant zero.

2. Suppose that τ_i's job is holding some lock[s] at time t_f. It is then possible that some jobs of some tasks τ_k, with $pri(k) < pri(i)$, are blocked from executing due to the potential effect on the system ceiling $\Pi(t)$ of the locks held by τ_i's job. But the (negative) impact of such blocking is felt by τ_k's jobs, not τ_i's. Hence, moving τ_k's jobs arrivals forward in time, if possible, cannot increase (although it may decrease) the amount of execution received by τ_i So once again, the interference I_k for each τ_k is maximized when each job of τ_k arrives as soon as legally permitted to do so – i.e., under the synchronous arrival sequence at time-instant zero.

Hence regardless of whether τ_i's job is within a critical section or not at time-instant t_f, we see that the I_k is maximized for each k if τ_k's jobs arrive according to the synchronous arrival sequence at zero. ■

We have thus established that if τ is not hybrid-priority schedulable and τ_i's job is the greatest-priority job that could possibly miss a deadline, a deadline miss for a job of τ_i will occur for the synchronous arrival sequence of jobs starting at time zero for all tasks with priority level $\leq pri(i)$, plus at most a single blocking computation that starts at time-instant zero and ends at time-instant $B(pri(i), t_f)$. More formally,

Lemma 6 *If τ is not hybrid-priority schedulable and τ_i's job is a greatest-priority job that could possibly miss a deadline, then a job of τ_i misses a deadline at some time-instant t_f if:*

1. *All the tasks in τ of priority-level $\leq pri(i)$ generate jobs according to the synchronous arrival sequence starting at time-instant zero;*

2. *t_f is of the form $(k \times T_i + D_i)$ for some integer $k \geq 0$; and*

3. *There is one blocking execution over the time-interval $[0, B(pri(i), t_f))$.*

■

The converse of Lemma 6 is easily seen to be true. That is, τ is unschedulable if the conditions 1-3 of Lemma 6 are satisfied, since the particular job-arrival sequence that is defined by these conditions bears witness to the fact that τ is not hybrid-priority schedulable. We consequently have the following exact (necessary and sufficient) characterization of hybrid-priority schedulability:

Theorem 1 *A necessary and sufficient condition for τ to not be hybrid-priority schedulable is that there exists a $\tau_i \in \tau$ and a time-instant $t_f > 0$ such that:*

1. *All the tasks in τ of priority-level $\leq pri(i)$ generate jobs according to the synchronous arrival sequence starting at time-instant zero;*
2. *t_f is of the form $(k \times T_i + D_i)$ for some integer $k \geq 0$; and*
3. *There is one blocking execution over the time-interval $[0, B(pri(i), t_f))$.*

5 Schedulability testing

Theorem 1 above identified necessary and sufficient conditions for a resource-sharing hybrid-priority sporadic task system to not be schedulable. In this section, we build and analyze an exact schedulability test based upon this result. In Section 5.1 below, we use the results of Section 4 above to derive an exact – necessary and sufficient – algorithm for schedulability analysis of hybrid-priority sporadic task systems. In Section 5.2, we discuss the relationship of our algorithm to algorithms that have previously been proposed for schedulability analysis of sporadic task systems scheduled using "pure" FP or pure EDF.

5.1 A schedulability-testing algorithm

In Theorem 1 above, we identified very precise conditions that must be satisfied, in order that τ not be hybrid priority schedulable. We will now exploit these conditions to devise a schedulability test. Our test proceeds as follows.

1. Initially, $\ell \leftarrow 1$. It is trivially true that no job generated by any task with priority-level $< \ell$ can possibly miss a deadline.
2. Under the assumption that no job generated by any task with priority-level $< \ell$ can possibly miss a deadline, determine whether any job generated by some task with priority-level ℓ could miss a deadline.
3. If the answer is "yes," then we declare that τ is not schedulable, and are done. If the answer is "no" and all tasks in τ have been considered (i.e., the largest priority-level of any task in τ is ℓ), we declare τ schedulable and are done. Otherwise, we increment ℓ and repeat the step above.

We now focus upon the second step listed above. Let us assume that no job of tasks with priority-level $< \ell$ can possibly miss a deadline. By Theorem 1, a deadline miss by a priority-level ℓ job is possible if and only if a deadline miss occurs in the synchronous arrival sequence of all tasks with priority-level $\leq \ell$ starting at time-instant zero, coupled with the maximum blocking (as given by condition 3 of Theorem 1). Furthermore, by condition 2 of Theorem 1, the possible values of t_f in the statement of Theorem 1 are

$$\bigcup_{pri(i)=\ell} \{k \times T_i + D_i \mid k \geq 0\} . \tag{3}$$

How many such possible values can there be? Since the interval $[0, t_f)$ contains no idle instant (Lemma 1), all potential values of t_f are bounded from above by the length of the longest possible busy interval for task system τ (note that this may, in fact, be a loose bound since only those tasks that have priority level $\leq pri(i)$, plus the blocking execution, contribute to the busy interval). Techniques are known for computing the length of the largest possible busy interval for a given system of sporadic tasks. An obvious bound is the least common multiple of the period parameters. In general, the least common multiple of the periods may be exponential in the representation of the task systems. By using techniques that are virtually identical to the ones used in the analysis of "pure" EDF scheduling [4], better bounds may be obtained for *bounded-utilization* sporadic task systems (see Definition 1). For such systems, it has been shown [3, 4, 14] that the length of the largest busy interval is bounded from above by a function that is pseudo-polynomial in the representation of task system τ.

For each such candidate value of t_f, we may determine whether a deadline miss would occur by

1. Computing $B(\ell, t_f)$, the maximum size of the blocking computation that may be encountered, according to Equation 2.
2. Simulating the schedule on the synchronous arrival sequence starting at time zero for all the tasks with priority level $\leq \ell$ out to t_f, keeping in mind that the interval $[0, B(\ell, t_f))$ is consumed by a blocking computation. Each such simulation could be done in time polynomial in the length of the interval being simulated (and hence pseudo-polynomial in the representation of the task system).

Hence, checking whether a deadline is missed for a given t_f takes pseudo-polynomial time, and there are pseudo-polynomially many distinct values of t_f that need to be tested. Since the composition of a pair of pseudo-polynomial functions is itself pseudo-polynomial, this immediately yields a pseudo-polynomial algorithm for determining whether any priority-level ℓ deadline can possibly be missed. This observation, in conjunction with the fact that there are polynomially many possible values for ℓ (since the number of priority levels that need to be considered is necessarily no more than the number of tasks), immediately yields the following result:

Theorem 2 *Exact schedulability analysis of hybrid-priority resource-sharing bounded-utilization sporadic task systems takes time pseudo-polynomial in the representation of the task system.*■

Additional optimizations. The focus in this section (indeed, in this paper), has been on demonstrating that exact schedulability analysis of bounded utilization hybrid-priority sporadic task systems can be done in pseudo-polynomial time. The goal has not been to come up with the most efficient implementation possible within this broad class of pseudo-polynomial run-times. Neither has it been to do the "tightest" analysis (again, in terms of run-time) of the algorithm we have derived. For instance, our algorithm re-generates the schedule for every candidate value of t_f, at each priority-level ℓ — we went with this description because it seemed the simplest, while retaining the desired pseudo-polynomial time complexity. It would be very simple to improve upon the implementation by re-using some of the results from one simulation run in subsequent runs, so that only relatively small parts of the schedule would have to be re-generated during any one run[5]. There are a large number of such possible optimizations that would collectively improve the run-time of the schedulability test — we are working on describing them all in an extended version of this paper, currently under preparation.

5.2 Relationship to EDF and FP schedulability analysis

The algorithm we have derived in Section 5.1 above generalizes the currently best-known algorithms for schedulability analysis of pure FP scheduled or pure EDF scheduled systems, as we explain below.

FP. Response Time Analysis (RTA) [17, 27, 18, 8] is the algorithm of choice for the schedulability analysis of resource-sharing sporadic task systems that are scheduled using fixed priorities. As with our approach, RTA is also based on the idea of generating schedules to determine whether deadlines are missed, and also has pseudo-polynomial time complexity. However, the schedule generation is done implicitly via the solution of recurrence equations rather than explicitly, and typically has very fast convergence time in practice. It is known that for systems in which all tasks have their deadline parameters no larger than their periods (i.e., $D_i \leq T_i \forall i$), RTA need only simulate the schedule out until the first deadline of each task — while this remains pseudo-polynomial in the representation of the task system, $\max_i\{D_i\}$ may typically be quite a bit smaller than the maximum length of a busy interval.

However, it is known [8] that RTA is not exact when jobs may have ending critical sections (ECS's) – under such circumstances, RTA becomes a sufficient but not necessary test. Fixes were proposed [8, 26, 25] to render it exact, but these fixes were subsequently shown to be incorrect [6, 7]. These errors have since been corrected [12]; however, the corrected algorithm presented in [12] requires that the simulation be conducted out to the end of the busy interval even when $D_i \leq T_i \ \forall i$. Specializing the algorithm presented in this paper to the special case where all tasks are assigned distinct priorities yields essentially the same algorithm as the one in [12] – the sole difference is that we have chosen to present the algorithm as a simulation-based one rather than on the basis of solving recurrence equations (of equivalent computational complexity).

EDF. The Enhanced Processor Demand Test (EPDT) [2, 20, 24] does exact schedulability analysis of EDF-scheduled uniprocessor resource-sharing sporadic task systems. The algorithm presented in this paper is a (non-trivial) extension of the algorithm in [2], which is also simulation-based. It was shown in [2] that EPDT, too, has pseudo-polynomial run-time for bounded-utilization sporadic task systems.

6 Additional Related work

As we have stated in Section 1, ours is not the first attempt at integrating the FP and EDF scheduling policies. We now briefly describe some previous such attempts, and explain how our work differs from these prior efforts.

The Combined Static and Dynamic (CSD) scheduler was proposed by Zuberi, Pillai, and Shin [29] as a scheduler for the EMERALDS small-footprint RTOS. In addition to the fact that CSD does not allow for the sharing of non-preemptable serially reusable resources, its design criteria were also quite different from the ones we consider here. In CSD, hybrid priorities is essentially a means of using EDF to enhance the schedulability of systems that are not schedulable under (the preferred) FP scheduling. Among other differences to our hybrid policy, CSD requires that all EDF-scheduled tasks have highest priority (in our framework, this maps to there being at most one task in every priority level other than priority-level one, and all the re-

[5]Since there are pseudo-polynomially many runs, note that doing so does not improve upon the pseudo-polynomial run-time bound stated in Theorem'2.

maining tasks being assigned priority level one).

In [11], Crespo et al. propose a scheduler for use in control applications, that adopts a strategy opposite to CSD's – EDF-scheduled tasks are all assigned to the lowest priority level. This choice is again dictated by the unique requirements of their application domain: operating system and control kernel processes are assigned high, fixed, priorities, while control and other tasks are scheduled at the lowest priority using EDF.

González Harbour and Palencia Gutiérrez [16] considered the implementation of EDF-scheduled systems upon platforms that only offer support for fixed priorities. They envisioned a hierarchical scheduler implemented upon such a platform, in which different sub-systems could be implemented upon different (fixed) priorities, with each such subsystem having the choice of an EDF or a FP scheduler. For such implementations, they derived formulas for computing the worst-case response time of each task in the system. Our work differs from this in several ways. First, our motivation is completely different from that of [16]: while they consider hierarchically scheduled systems, we are considering integrated systems in which there are relatively few priorities, and in which the priorities are used for conveying semantic information about the application being designed. Second, we have an integrated schedulability test that is simulation based, while [16] derives formula-based worst-case response times of individual tasks. Third, there are some significant holes (although not errors) in the proofs in [16] – this is mainly a consequence of the fact that results for "pure" EDF have been imported and used directly for the schedulability analysis of hybrid systems, without demonstrating that these results can indeed be applied for such hybrid systems. Fourth, the approach in [16] is pessimistic (sufficient but not necessary) for the same reason that original RTA is pessimistic for task systems including jobs that have ending critical sections (ECS's) — the proposed optimizations of [26, 25, 12] have not been incorporated into the analysis in [16]. And finally, the approach in [16] mandates that each task have its relative deadline parameter be no larger than its period ($D_i \leq T_i \forall i$) — this is a direct consequence of the fact that they use the response-time equations from FP analysis, which hold for such tasks only, while our schedulability test places no such restrictions.

7 Conclusions

We believe that that priorities have a restricted, though important when applicable, role to play in real-time systems design and implementation. To this end, we have proposed here a scheduling policy that is EDF-based, but allows for the use of (hopefully, just a few) distinct priorities to reflect semantic information about the system being designed. This scheduling policy may be used upon platforms comprised of a single shared preemptive processor plus additional non-preemptable shared resources that are serially re-usable. We have derived and analyzed an exact schedulability test for the scheduling of sporadic task systems under our hybrid scheduling policy. We have demonstrated that this test generalizes the previously-known best tests for FP and EDF schedulability analysis, at comparable computational complexity.

Acknowledgements. Many of the ideas in Section 2 were motivated by discussions with Ted Baker, Alan Burns, Giorgio Buttazzo, Lui Sha, and Tullio Vardanega.

References

[1] BAKER, T. P. Stack-based scheduling of real-time processes. *Real-Time Systems: The International Journal of Time-Critical Computing 3* (1991).

[2] BARUAH, S. Resource sharing in EDF-scheduled systems: A closer look. In *Proceedings of the IEEE Real-time Systems Symposium* (Rio de Janeiro, December 2006), IEEE Computer Society Press, pp. 379–387.

[3] BARUAH, S., HOWELL, R., AND ROSIER, L. Feasibility problems for recurring tasks on one processor. *Theoretical Computer Science 118*, 1 (1993), 3–20.

[4] BARUAH, S., MOK, A., AND ROSIER, L. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th Real-Time Systems Symposium* (Orlando, Florida, 1990), IEEE Computer Society Press, pp. 182–190.

[5] BOLLELLA, G. Priorities considered harmful (scheduling algorithms considered). Usenet posting, 2002. Archived at URL http://cio.nist.gov/esd/emaildir/lists/rtj-discuss/msg00356.html .

[6] BRIL, R. Existing worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption is too optimistic. Tech. Rep. 06-05, Technische Universiteit Eindhoven (TU/e), Feb. 2006.

[7] BRIL, R. Existing worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption refuted. In *Work in Progress (WiP) session of the 18th Euromicro Conference on Real-Time Systems* (July 2006), pp. 1–4.

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

[9] BUTTAZZO, G. Rate-monotonic vs. EDF: Judgement day. *Real-Time Systems: The International Journal of Time-Critical Computing 29*, 1 (2005), 5–26.

[10] BUTTAZZO, G. C. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, second ed. 2005.

[11] CRESPO, A., ALBERTOS, P., BALBASTRE, P., VALLS, M., AND SIM, M. L. J. Schedulability issues in complex embedded control systems. In *IEEE Computer Aided Control Systems Design Symposium* (2006), IEEE Computer Society Press, pp. 1200–1205.

[12] DAVIS, R. L., BURNS, A., BRIL, R. J., AND LUKKIEN, J. J. Controller area network (CAN) schedulability analysis: Refuted, revisited, and revised. *Real-Time Systems: The International Journal of Time-Critical Computing 35* (2007), 239–272.

[13] DERTOUZOS, M. Control robotics : the procedural control of physical processors. In *Proceedings of the IFIP Congress* (1974), pp. 807–813.

[14] GEORGE, L., RIVIERRE, N., AND SPURI, M. Preemptive and non-preemptive real-time uniprocessor scheduling. Tech. Rep. RR-2966, INRIA: Institut National de Recherche en Informatique et en Automatique, 1996.

[15] GWINN, J. Priorities considered harmful (scheduling algorithms considered). Usenet posting, 2002. Archived at URL http://cio.nist.gov/esd/emaildir/lists/rtj-discuss/msg00353.html .

[16] HARBOUR, M. G., AND GUTIRREZ, J. C. P. Response time analysis for tasks scheduled under EDF within fixed priorities. In *Proceedings of the IEEE Real-Time Systems Symposium* (Cancun, Mexico, December 2003), IEEE Computer Society Press, pp. 200–209.

[17] JOSEPH, M., AND PANDYA, P. Finding response times in a real-time system. *The Computer Journal 29*, 5 (Oct. 1986), 390–395.

[18] KLEIN, M., RALYA, T., POLLAK, B., OBENZA, R., AND HARBOUR, M. G. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, Boston, 1993.

[19] LEHOCZKY, J., SHA, L., AND DING, Y. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the Real-Time Systems Symposium - 1989* (Santa Monica, California, USA, Dec. 1989), IEEE Computer Society Press, pp. 166–171.

[20] LIPARI, G., AND BUTTAZZO, G. Schedulability analysis of periodic and aperiodic tasks with resource constraints. *Journal Of Systems Architecture 46*, 4 (2000), 327–338.

[21] LIU, C., AND LAYLAND, J. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM 20*, 1 (1973), 46–61.

[22] MOK, A. K. *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983. Available as Technical Report No. MIT/LCS/TR-297.

[23] MOSSE, D., BAKER, T., BARUAH, S., BUTTAZZO, G., BURNS, A., SHA, L., AND STANKOVIC, J. Fixed or dynamic priority? that is the question. In *Real-Time Systems Symposium (Panel discussion)* (Lisbon, Portugal, December 2004), IEEE Computer Society Press, p. 9.

[24] PELLIZZONI, R., AND LIPARI, G. Feasibility analysis of real-time periodic tasks with offsets. *Real-Time Systems: The International Journal of Time-Critical Computing 30*, 1–2 (May 2005), 105–128.

[25] TINDELL, K. W., BURNS, A., AND WELLINGS, A. Calculating controller area network (CAN) message response times. *Control Engineering Practice 3*, 8 (1995), 1163–1169.

[26] TINDELL, K. W., HANSSON, H., AND WELLINGS, A. J. Analysing real-time communications: Controller area network (CAN). In *Proceedings of the IEEE Real-Time System Symposium* (December 1994).

[27] WELLINGS, A., RICHARDSON, M., BURNS, A., AUDSLEY, N., AND TINDELL, K. Applying new scheduling theory to static priority pre-emptive scheduling. Tech. Rep. RTRG/92/120, Real Time Systems Group, Department of Computer Science, University of York, 1992. Submitted to the UK Software Engineering Journal.

[28] ZERZELIDIS, A., AND WELLINGS, A. Getting more flexible scheduling in the RTSJ. In *Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)* (2006), IEEE Computer Society Press, pp. 3–10.

[29] ZUBERI, K. M., PILLAI, P., AND SHIN, K. G. EMERALDS: A small-memory real-time microkernel. In *Operating Systems Review 34(5): Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (Decmeber 1999), pp. 277–291.