Linear Time On-Line Feasibility Testing Algorithms for Fixed-Priority, Hard Real-Time Systems

Michael E. Thomadakis and J. C. (Steve) Liu

Computer Science Department

Texas A&M University

College Station

TX 77843–3112

{miket,jcliu}.cs.tamu.edu

Technical Report TR00-006

January 21, 2000

Index Terms *Firm-Aperiodic Tasks; Fixed Priority Preemptive Scheduling; FPF/LDL Schedules; Hard Periodic Real-Time Tasks; Idle-Capacity Management; On-Line Feasibility Algorithms; Workload Processes.*

Linear Time On-Line Feasibility Testing Algorithms for Fixed-Priority, Hard Real-Time Systems

Michael E. Thomadakis and J. C. (Steve) Liu
Computer Science Department
Texas A&M University
College Station
TX 77843–3112
miket.cs.tamu.edu

January 21, 2000

Abstract

In this paper we develop linear time, on-line feasibility testing algorithms for the guaranteed scheduling of firm aperiodic tasks in fixed-priority real-time systems. Firm aperiodic tasks, unlike critical periodic tasks which are guaranteed off-line, can be guaranteed dynamically only if they pass on-line feasibility tests. In this paper we derive feasibility tests drawing upon the *Workload-Matrix* method, an innovative technique which determines the idle capacity exactly within arbitrary intervals of periodic schedules, in time $\Theta(n)$, where n is the number of periodic tasks. To the best of our knowledge, this is the first algorithm reported in the literature, which can accomplish this task in linear time. Previous state-of-the-art methods based on the static or dynamic slack stealing, or the EDL method by Silly and Chetto, require pseudo-polynomial time for the determination of spare capacity to guarantee one single task and they incur slack variable maintenance overhead continuously.

Experimental results show that with $n = 160$ periodic tasks, the actual computation time for the admission control procedure is less than 30μ-secs on a $250MHz$ SGI Origin 200 workstation and less than 90μ-secs on a SUN Ultra-Sparc I, $143MHz$ machine. The proposed methodology has been successfully integrated into firm aperiodic task scheduling algorithms for hard real-time systems.

Index Terms: *Firm-Aperiodic Tasks; Fixed Priority Preemptive Scheduling; FPF/LDL Schedules; Hard Periodic Real-Time Tasks; Idle-Capacity Management; On-Line Feasibility Algorithms; Workload Processes.*

1 Introduction

In this work we develop linear time on-line feasibility testing algorithms which can be used in the admission control procedure of *firm* aperiodic tasks. We consider fixed-priority scheduling schemes in which periodic real-time tasks with hard deadlines are assigned fixed-priorities and are guaranteed off-line. Firm tasks generate no utility for the system if they finish after their deadlines. They have unknown parameters and arrive at unpredetermined instances of time, and thus, they cannot be guaranteed off-line. The system affords an *on-line guarantee* to each new firm task at the time of its arrival. The guaranteed scheduling of firm tasks involves three logical steps. First, upon the arrival of a new firm aperiodic J_a at time r_a, the system verifies that the schedule of already guaranteed tasks will remain feasible if J_a is admitted. In the second step, if feasibility is verified, the system performs *schedule maintenance*, where it determines the points in time at which the affected tasks will receive the processor. Finally, the system dispatches the tasks according to the modified schedule. The real-time system may admit a new firm task, only if, there is sufficient idle processor capacity to accommodate its execution requirements. The on-line feasibility testing then reduces to a method which determines the idle capacity, denoted as $Z^S[r_a, d_a)$, where d_a is the firm deadline of J_a. In this paper we focus on the on-line feasibility testing stage of the admission control procedure.

The efficient determination of the exact idle processor capacity in fixed and dynamic priority schedules has been a challenging problem. Presently, state-of-the-art algorithms, such as, Slack Stealing [1, 2] or EDL methods [3, 4], require pseudo-polynomial time complexity in order to test the feasibility of one firm task on-line. Several methods have been devised to compute a lower bound on the idle capacity approximately [5, 6, 7]. However, approximate methods can at times become overly pessimistic and reject several schedulable firm tasks. Both exact and approximate methods incur high run-time overheads for continuous slack variable maintenance even when there are no firm tasks present. Section 2 presents a detailed account on previously proposed methods for the on-line feasibility problem.

In this paper we develop algorithms based on exact feasibility conditions, which guarantee and schedule on-line firm aperiodic tasks, in linear time. Our primary concern, besides exactness, is low complexity and execution time overheads. For a system with n periodic tasks, $t_i,\ i = 1, \ldots, n$, we develop a scheme where idle capacity $Z_i^S(t)$ is determined as a deterministic process of time, defined as the total idle time in $[0, t)$ in a schedule S of the first i periodic tasks. This is obtained as $t - W_i^S(t)$, where $W_i^S(t)$ is defined as the total busy time in schedule S in $[0, t)$, for the first i periodic tasks. In the following sections we provide computational procedures for the exact calculation of $W_i^S(t)$ for arbitrary t. A fixed priority schedule of the first i periodic tasks is **legal** if all job instances τ_{ij} of task τ_i are assigned to the processor after they are released at times r_{ij}. The schedule is **feasible** if all instances τ_{ij} always receive the processor for their C_i computation requirements units, in every $[r_{ij}, d_{ij})$, where d_{ij} is the absolute deadline of instance τ_{ij}. In essence, for the same periodic task set $\boldsymbol{\tau}$ infinitely many feasible schedules can be defined, when the task parameters are rational numbers. Different schedules consist of different sequences of busy and idle periods, where periodic tasks are serviced over the busy periods. Firm tasks can only be scheduled in the idle periods of some feasible periodic schedule, otherwise some instance τ_{kl} will miss its deadline. We formally define two specific fixed-priority periodic schedules, called the *Fixed-Priority First* (FPF) and the *Latest-Deadline Last* (LDL). The FPF is a non-idling schedule where instances are dispatched to the processor at their earliest possible time after their release, according to the Highest-Priority Next (HPN) dispatching rule. The LDL is an idling schedule where instances are dispatched at their latest possible time, while they all can meet their deadlines. The FPF and the LDL are the two unique extremes fixed-priority schedules for a task set $\boldsymbol{\tau}$. We prove the *Workload Dominance* theorem by which

$W_i^{\mathrm{F}}(t) \geq W_i^{\mathrm{L}}(t)$, where F stands for FPF and L for LDL. When a firm task J_a requests C_a computation units in $[r_a, d_a)$ if $d_a - r_a - (W_n^{\mathrm{F}}(d_a) - W_n^{\mathrm{F}}(r_a)) < C_a$ the system cannot accept J_a under the FPF periodic schedule. By the Workload Dominance theorem if we start dispatching the tasks at time r_a under the LDL schedule, then it is possible to shift additional idle capacity within $[r_a, d_a)$ allowing the possibility to guarantee task J_a. In subsequent sections we derive the two extreme schedules formally and develop efficient computational procedures for the $W_i^{\mathrm{F}}(t)$ and $W_i^{\mathrm{L}}(t)$.

The rest of this paper is organized as follows. Section 2 discusses previous on-line feasibility methods. Section 3 derives the two extreme schedules. Section 4 develops two computational procedures for $W_i^S(t)$. Section 5 presents experimental results on the effectiveness and actual execution time efficiency of our idle capacity determination algorithms. Finally, Section 6 concludes this paper and highlights several scheduling algorithms which utilize the feasibility testing methods.

2 Previous Work on Feasibility Algorithms

A number of on-line feasibility testing algorithms have been proposed in the past. In this section we provide an overview of these methods and discuss their time and space complexities. We include feasibility testing methods for dynamic priority systems for comparison purposes. Feasibility algorithms fall broadly into two categories, namely, the *slack-stealing* and the *schedule construction* methods. A third approach, the *server-based* methods, preallocate off-line portions of the schedule via the use of aperiodic servers, such as, the *Poll* server (PS) [8, 9], the *Priority Exchange* (PE), the *Extended-Priority Exchange* (EPE) [10, 11], the *Deferrable* (DS) [10, 12], and the *Sporadic* (SpS) [13] server. These methods represent important milestones as algorithms for the scheduling of aperiodic tasks. However, they are not feasibility algorithms per-se, and, thus, we do not discuss them any further in this work.

The *Slack Stealing* (SS) is a class of algorithms based on the observation that a periodic task instance τ_{ij} can be delayed at time t by an amount of time equal to its *slack* $S_i(t)$ without missing its deadline. When slack is available at all priority levels $S_{\min} = \min_{i=1}^{n}\{S_i(t)\}$ then a firm aperiodic task J_a can take precedence over all task during $[t, t + S_{\min})$. The need to provide on-line guarantees to incoming firm aperiodic tasks J_a, requires the SS algorithms to search all the consecutive slack intervals until they can find total spare capacity exceeding the execution requirements C_a. This leads to pseudo-polynomial time algorithms. Even after J_a is admitted, SS continuously searches for the immediately next slack interval to grant to J_a until it finishes. Furthermore, slack correctness requires the continuous maintenance of $\Theta(n+m)$ slack variables, at the completion of every task and some times at each task switching. The high admission overhead and the continuous state maintenance makes SS algorithms very hard to scale as the number of aperiodic requests increases. Therefore, as the aperiodic tasks become more fine-grained, the system spends proportionally more time deciding if a firm is schedulable than serving aperiodic requests.

The *Static Slack Stealing* (SSS) [14, 2], upon arrival of a task J_a determines the amount of slack within time interval $[r_a, r_a + D_a]$ based on precomputed information about slack. Task admission and subsequent scheduling of J_a each take $O((H + D_a)n^2)$ time [14], where n is the number of periodic tasks and H is their hyper-period, which is pseudo-polynomial time. SSS algorithms place heavy run-time execution and storage requirements and require continuous maintenance of $\Theta(n + m)$ state variables. Davis *et al.*, [1, 15] proposed an exact *dynamic* slack stealing (EDSS) method. EDSS locates slack dynamically at run time using a forward search, fixed-point iteration algorithm. The EDSS combines the preprocessing step of the SSS with dynamic slack determination and, thus, has very high run-time computation overhead. The EDSS determines the amount of slack at all n priority levels, in time $O(G \cdot n^2)$ where G is a linear function of the total number of busy periods in $[t, t + \max_i\{d_i(t)\})$, which is pseudo-polynomial. The EDSS theoretically must run at each scheduler clock-tick to keep current the slack variables. It runs at least at the completion of each periodic task instance and it requires $O(n)$ slack variable maintenance at each context-switching time. Davis *et al.*, [5, 16, 17] also proposed approximate DSS (ADSS) algorithms which compute pessimistic estimates of the slack $S_i(t)$. Slack computation at all n levels for admission control and the times at which the firm aperiodic task can run, each take $O(n^2 + n)$ time. On-line admission control for one firm aperiodic with a heuristically preselected priority takes $\Theta(n + m)$ time. Again, the slack for particular task must be recomputed at the completion of each one of its instances in $O(n)$ time, and $O(n + m)$ slack variables must be maintained at every context switching time. For larger periodic task sets the continuous slack maintenance becomes significant source of overhead. Another problem is that in the proposed method the priority of a candidate aperiodic task J_a is heuristically preselected to be compatible with the priorities of the tasks that precede and follow J_a in deadline order. If the deadline D_a of J_a is larger than the deadlines D_i of all the periodic tasks, $D_a > \max_{i=1}^{n}\{D_i\}$, the EDSS and ADSS algorithms degrade into background processing. Finally, ADSS computes lower bounds on the slack and it is thus pessimistic. More recently, Binns [18] proposed algorithms based on SSS for the scheduling of sporadic tasks, resulting from *incremental* and *design-to-time* periodic tasks. An admission control decision for a sporadic task J_s at a priority level k, requires time $O(n) + (t_k - 1) \cdot \max\{O(n(n-1)), O(n\left\lceil \frac{T_n}{T_1} \right\rceil)\}$, where t_k is an integer number, the "transform" factor of T_k [18].

McElhone [19, 7] investigated two basic algorithms which test the schedulability of one firm aperiodic task, on-line. The first one is pessimistic and takes $\Theta(n^2)$ time to compute a lower bound on the spare capacity. The second is an exact algorithm taking pseudo-polynomial time. When the deadline D_a of the firm task is $D_a > \max_{i=1}^{n}\{D_i\}$ of the deadlines of the periodic tasks, the algorithms essentially schedule the firm at background priority.

For comparison purposes we also overview the complexity of firm task scheduling based on dynamic priority algorithms. Chetto and Chetto [3] and Schwan and Zhou [20] investigated on-line admission tests, based on the Earliest Deadline First (EDF) algorithm. Their methods requires time proportional to the number N of all periodic jobs released within the least common multiple of the periodic task periods, called the hyper-period

H, and thus, they are pseudo-polynomial in time complexity. Further, Schwan and Zhou methods require all periodic and aperiodic tasks to go through continuous schedulability testing, leading to unnecessarily high overheads. The method by Schwan *et al.*, is a schedule construction type.

Silly and Chetto[21] proposed developed feasibility algorithms, based on the *EDL method*, to guarantee firm tasks, taking $O(N + m(r_a))$ time and storage $O(N)$, where N is the total number of invocations of all the periodic tasks within one hyper-period, and $m(r_a)$ is the number of pending aperiodic tasks at the time of admission. Chetto *et al.*, [22] presented an algorithm to admit groups of m firm aperiodic tasks related by precedence constraints. This algorithm takes $O(N^2)$ time where N is the total number of periodic and aperiodic tasks active in $[d', H + D')$, where d' is the earliest deadline in the current group, D' is the latest deadline of all pending aperiodic tasks at the time of the group admission and H is the hyper-period of the system. Tia in [23] proposed a technique to compute slack in dynamic priority schedules of periodic tasks requiring $\Theta(N^2 + N)$ storage overhead, which is impractical for all practical purposes. Silly in [4] presented an algorithm to schedule a single soft aperiodic task in time $O(n\frac{\max_i\{D_i\}}{\min_i\{T_i\}})$, where $\max_i\{D_i\}$ and $\min_i\{T_i\}$ are the maximum deadline and minimum period, respectively, among all periodic tasks. The storage complexity is $O(N)$, where N is the total number of periodic jobs released in one hyper-period. Other methods for feasibility testing include the methods by Ripoll *et al.*, [24, 25] and Buttazzo *et al.*, [26, 27] for dynamic priority systems. On-line feasibility algorithms based on their methods have pseudo-polynomial complexity in order to guarantee a single task.

Several schedule construction methods have been developed in the past. Buttazzo *et al.*, in [28, 29] proposed the RED method, where tasks are serviced by the EDF discipline and feasibility testing is performed dynamically on every task in the system, periodic and aperiodic. The time complexity is $O(M)$ where M is the total number of tasks in the ready queue and it can be pseudo-polynomial.

3 Real-Time Task Model and Scheduling Framework

3.1 System Model and Schedules for Periodic Tasks

Periodic tasks with fixed periods and hard deadlines are very common in real-time systems [30, 31]. We consider n hard real-time periodic tasks τ_i, denoted as $\boldsymbol{\tau} = \{\tau_i(C_i, T_i, D_i, r_i);\ i = 1, \ldots, n\}$. For each task τ_i, $C_i > 0$ is the worst case (maximum) execution time, $T_i > 0$ is the request period, D_i is its relative deadline, with $C_i \leq D_i \leq T_i$ and $r_i \geq 0$ is its initiation displacement relative to time origin $t = 0$. Task τ_i generates an infinite sequence of jobs $\tau_{ij},\ j = 1, 2, \ldots$. Job τ_{ij} is released at time instances $r_{ij} = r_i + (j-1)T_i$ and must complete by time $d_{ij} = r_{ij} + D_i$. The *hyper-period* H of a periodic task set is defined as the least common multiple (LCM) of the request periods T_i, that is $H := \mathrm{LCM}\{T_1, T_2, \ldots, T_n\}$. When all $r_1 = r_2 = \cdots = r_n = a \geq 0$, the task system is called *synchronous*, and, without loss of generality, we assume $a = 0$. When $r_j \neq r_k$, for some j, k, then the system is called *asynchronous*, and we let $r = \max_{i=1}^{n}\{r_i\}$.

Hard deadline, non-periodic activities in are considered as *sporadic* and firm *aperiodic* tasks. Sporadic tasks $J_{si}(C_{si}, T_{si}, D_{si}, r_{si})$, embody activities with a minimum period of occurrence T_{si}. The other sporadic task parameters are the same as those of a periodic one. For the purpose of analysis we consider the sporadic tasks as periodic ones, and we can guarantee them off-line arriving at their maximum frequency.

Firm aperiodic tasks, denoted as $J_{ai}(C_{ai}, D_{ai}, r_{ai})$, characterize dynamic activities with parameters that become known at their release times. The system cannot guarantee all firm aperiodic tasks. We consider the aperiodic tasks as a sequence $\boldsymbol{J}_a := \{J_{ai}(C_{ai}, D_{ai}, r_{ai});\ i = 1, 2, \dots\}$, where task J_{ai} arrives at time r_{ai}, requests C_{ai} units of computation and has to complete by time $d_{ai} = r_{ai} + D_{ai}$. The scheduler calculates dynamically the total idle processor capacity $Z(r_{ai}, d_{ai})$, taking into account periodic, sporadic and already admitted hard aperiodic tasks that have to execute within the interval $[r_{ai}, d_{ai}]$. It accepts J_{ai}, *only if*, $C_{ai} \leq Z(r_{ai}, d_{ai})$, otherwise it rejects J_{ai}, immediately.

A *continuous time* schedule $\mathcal{S}$ of a task system $\boldsymbol{\tau}$ is a mapping of time $t \in \mathbb{R}$ (the real numbers) into the index set of the tasks $\{1, 2, \dots, n\}$, and $\{0\}$, symbolically,

$$\mathcal{S} \,:\, \mathbb{R} \longrightarrow \{0, 1, 2, \dots, n\}. \tag{1}$$

In other words, $\mathcal{S}(t)$ is the index of the task assigned to the processor at time t. When $\mathcal{S}(t) = 0$, no task is assigned to the processor and the system is idle. We let $s_i(t)$ be an indicator function for task τ_i, so that $s_i(t) := 1$ if $\mathcal{S}(t) = i$, and 0, otherwise. A scheduling algorithm $A^{\mathcal{S}}$ is a method that assigns a unique task τ_i to $\mathcal{S}(t)$ for each $t \geq 0$. From the schedule definition we exclude instantaneous processor assignments, as uninteresting cases.

3.2 System Workload Deterministic Processes

In this and subsequent sections we will use the following quantities repeatedly,

$$n_i(t) = \left\lfloor \frac{t - r_i}{T_i} \right\rfloor + 1,$$
$$m_i(t) = (t - r_i) \pmod{T_i}.$$

$n_i(t)$ is the current period number of τ_i at t, and $m_i(t)$ is the corresponding *translated* remainder, for $i = 1, 2, \dots, n$. For any $t \geq r_i$, $t = r_i + (n_i(t) - 1) \cdot T_i + m_i(t)$ and $r_{i,n_i(t)} \leq t < r_{i,n_i(t)+1}$. For a given periodic task set $\boldsymbol{\tau}$ we define the *released workload* functions $v_i(t), V_i(t)$ as,

$$v_i(t) := n_i(t) \cdot C_i \qquad V_i(t) = \sum_{k=1}^{i} v_k(t). \tag{2}$$

$v_i(t), V_i(t)$ is the total amount of work that task τ_i and tasks $\tau_1, \dots, \tau_i$, respectively, require of the system in the time interval $[0, t]$. $V_i(t)$ are right-continuous, step-functions with jump points at the release times of the tasks.

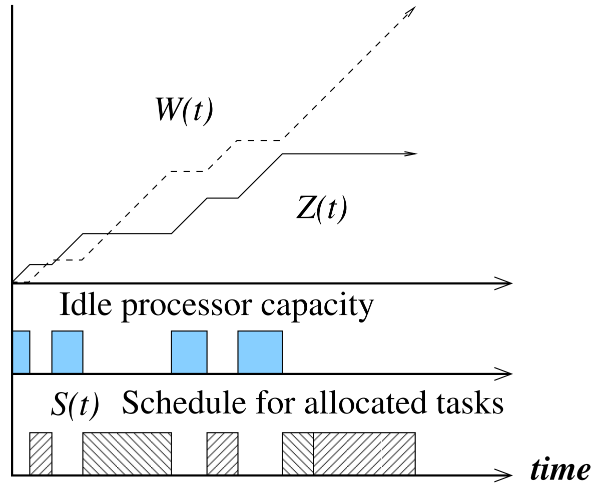


Figure 1: Illustration of Serviced Workload $W^S(t)$ and Idle Capacity $Z^S(t)$ Processes on Schedule $\mathcal{S}(t)$

$V_i(t)$ have identical increments every H time units (see Subsection III.C.3in [32]). For the hard aperiodic tasks $\boldsymbol{J}_{ai}$ we define the released workload $V_a(t)$, as $V_a(t) := \sum_{k=1}^{m} C_{ak},\ r_{am} \leq t < r_{a,m+1}$. The aggregate released workload $\boldsymbol{V}(t)$ of the system in $[0,t]$, is simply $\boldsymbol{V}(t) = V_n(t) + V_a(t),\ t \geq 0$.

Given a schedule $\mathcal{S}$ on $\boldsymbol{\tau}$, we define the *serviced workload* processes $w_i(t), W_i(t)$, that measure the amount of work the system has completed in the time interval $[0,t]$. We let $w_i(t)$ and $W_i(t)$ denote the total time the system has spent processing task τ_i and $\tau_k,\ k = 1,\ldots,i$, respectively, in the interval $[0,t]$, defined as

$$w_i(t) := \int_0^t s_i(x)dx \qquad W_i(t) = \sum_{k=1}^{i} w_k(t). \tag{3}$$

$w_i(t)$ and $W_i(t)$ are piece-wise continuous, linear, non-decreasing functions of t. For each i, when $s_i(t) = 1$, $w_i(t)$ and $W_i(t)$ are increasing with slope at a 45^o, and when $s_i(t) = 0$, $w_i(t)$ stays parallel to the x-axis. $W_i(t)$ is increasing with slope at a 45^o, if any of the tasks $\tau_k,\ k = 1,2,\ldots,i$ is receiving service at t. Note that $W_n(t_2) - W_n(t_1)$ is the total time that the processor is *busy* in the time interval $[t_1,t_2]$.

Given the process $W_i(t)$ we define the *Idle Processor Capacity* processes $Z_i(t)$, as the total amount of time the processor is idle, during the interval $[0,t)$, as,

$$Z_i(t) := t - W_i(t),\ i = 1,2,\ldots,n;\ t \geq 0. \tag{4}$$

Fig. 1 illustrates the serviced $W(t)$ and idle-capacity $Z(t)$ processes on some schedule $\mathcal{S}(t)$ of arbitrary tasks. In a schedule of periodic tasks, $Z_i(t_2) - Z_i(t_1)$ measures the available processor capacity remaining after the jobs of tasks $\tau_1,\tau_2,\ldots,\tau_i$, receive service in $[t_1,t_2)$.

3.3 Extreme Schedules for Periodic Task Sets

In our discussion we make the assumptions that the task system $\boldsymbol{\tau}$ is *schedulable*, that there is a known fixed priority assignment determined, for instance, by the deadline monotonic (DM) [33] rule, and that the tasks are already indexed according to this priority assignment, with smaller indices implying higher priorities. We could summarize our approach for the idle processor capacity management, as follows. Given a feasible task set $\boldsymbol{\tau}$ we derive off-line the following quantities

$$\boldsymbol{\tau} \Longrightarrow V \xrightarrow{A^{\mathcal{S}}} \mathcal{S} \Longrightarrow W \Longrightarrow Z.$$

where $\Longrightarrow$ implies that the quantity to the right can be directly obtained from the quantity to the left, and $\xrightarrow{A^{\mathcal{S}}}$ means that we can derive the quantity on the right after applying a scheduling algorithm $A^{\mathcal{S}}$. For a given task system $\boldsymbol{\tau}$, the released workload $V_i(t)$ are fixed. In this section we derive the two *extreme* schedules denoted as, $\mathcal{S}^{\mathrm{F}}$ and $\mathcal{S}^{\mathrm{L}}$, for the periodic tasks and discuss some of their key properties. In the next section, we develop efficient algorithms to compute the serviced workload $W_i(t)$ functions defined on them.

3.3.1 The Fixed-Priority-First (FPF) Non-Idling Schedule

We consider here the *Fixed-Priority First* (FPF), a non-idling schedule which directly reflects the given fixed priority assignment of the periodic tasks. FPF, denoted by $\mathcal{S}^{\mathrm{F}}$, is an *eager* schedule, dispatching the jobs τ_{ij} to the processor immediately after they arrive at r_{ij}, according to the usual fixed priority rules. FPF models the regular behavior of the system when there are no aperiodic tasks present. The derivation of $\mathcal{S}^{\mathrm{F}}$ first determines the schedule $s_1(t),\ t > 0$ and quantities $W_1(t), Z_1(t), s_{1j}, f_{1j},\ j = 1, 2, \ldots, n_1(\tilde{H})$ of task τ_1. Subsequently, the algorithm derives the same quantities for the remaining tasks inductively, one task at a time, in the order $\tau_2, \tau_3, \ldots, \tau_n$, with step i utilizing the quantities derived from step $i-1$.

Specifically, setting $V_0(t) = W_0(t) = 0$, for each $t \geq 0$, and $j = n_i(t)$, let

$$s_i(t) := \begin{cases} 1, & \text{if } i = 1, t \in [r_{ij}, r_{ij} + C_i), \\ 1, & \text{if } i > 1, Z_{i-1}(t) - Z_{i-1}(r_{ij}) < C_i \text{ and } \mathcal{S}_{i-1}(t) = 0, \\ & \text{for } t \in [r_{ij}, d_{ij}), \\ 0, & \text{otherwise;} \end{cases} \tag{5}$$

$$
\begin{aligned}
w_i(t) &= \int_0^t s_i(x)dx = \sum_{k=1}^{j-1} \int_{r_{ik}}^{d_{ik}} s_i(x)dx + \int_{r_{ij}}^{t} s_i(x)dx = (j-1)\cdot C_i + c_i(t), && (6)\\
c_i(t) &= \begin{cases} \min\{[t-r_{ij}]^+, C_1\} & \text{if } i=1, t\in[r_{ij}, d_{ij}) \text{ and } m_i(t)\le C_i,\\ \min\{[Z_{i-1}(t)-Z_{i-1}(r_{ij})]^+, C_i\}, & \text{if } i>1, t\in[r_{ij}, d_{ij}),\\ 0, & \text{otherwise;}\end{cases} && (7)\\
W_i(t) &= W_{i-1}(t) + w_i(t),\\
Z_i(t) &= Z_{i-1}(t) - w_i(t),\\
\mathcal{S}_i^{\mathrm{L}}(t) &= \sum_{k=1}^{i} k\cdot s_k(t).
\end{aligned}
$$

FPF dispatches the jobs to the processor as *early* as possible. The schedule for τ_i (5) reflects this by assigning jobs τ_{ij}, where $j = n_i(t)$, to the processor within the first C_i units of idle time in the interval $[r_{ij}, d_{ij})$. The quantity $c_i(t)$ defined by Eq. (7), determines the total service that job τ_{ij} has already received within the latest incomplete period at time t, $[r_{ij}, t)$. In section 4 we develop algorithms that compute $W_i(t)$ and $c_i(t)$ exactly and very efficiently. Once a schedule (5) is decided upon, all other quantities for τ_i can be determined directly. Job start s_{ij} and f_{ij} finish times are given by,

$$s_{ij} = \min\{x : x \ge r_{ij},\ V_{i-1}(x) - W_{i-1}(x) = 0\}, \tag{8}$$

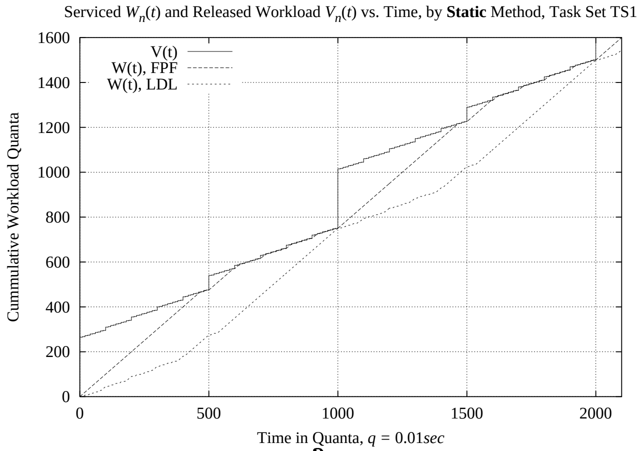
$$f_{ij} = \min\{x : x \ge r_{ij} + C_i,\ Z_{i-1}(x) - Z_{i-1}(r_{ij}) = C_i\}. \tag{9}$$

s_{ij} and f_{ij} can be evaluated either via a simple fixed-point iteration algorithm or via bisection, within interval $[r_{ij}, d_{ij})$.

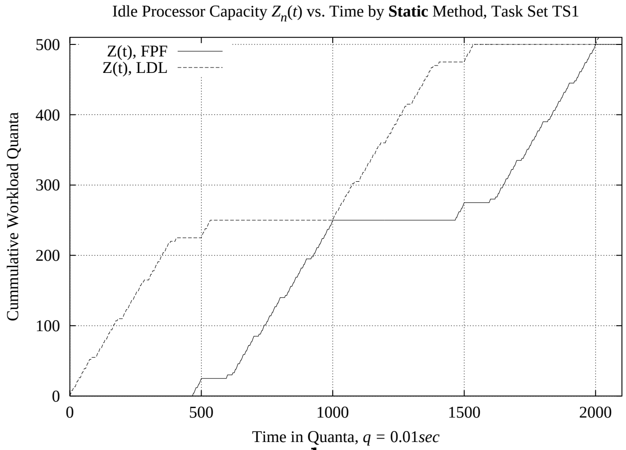
Table 1: Parameters of Synthetic Task Set 1, with $n = 5$, $q = 0.01sec$, $H = 10sec$

C_i	T_i	D_i	r_i
0.020	0.100	0.100	0.000
0.030	0.200	0.200	0.000
0.100	1.000	1.000	0.000
0.500	5.000	5.000	0.000
2.000	10.000	10.000	0.000

Fig. 2**a.** plots the released $V_n(t)$ and the serviced $W_n(t)$ workload under FPF (second from top curve), in the first two hyper-periods ($H = 1000$) of a synthetic periodic set TS1, defined in Table 1. The release of new periodic jobs forces the upward jumps to the $V_n(t)$ process. $W_n(t)$ under FPF becomes increasing with a slope of a 45^o angle, as soon as new periodic tasks are released. $W_n(t)$ remains increasing while the unfinished workload $V_n(t) - W_n(t) > 0$. Fig. 2**b.** plots the corresponding idle capacity process $Z_n(t)$ (bottom curve). As expected, $Z_n(t)$ remains parallel to x-axis while the processor services eagerly the periodic requests. When the unfinished workload $V_n(t) - W_n(t)$ at $t = 461$ becomes zero, the processor becomes idle and $Z_n(t)$ starts accumulating idle capacity. Fig. 3**a.** presents the FPF schedule of the same synthetic task set TS1. The bottom curve shows the *idle* periods ("task"τ_0). One readily notices that under FPF, for this particular example set, most of the idle



a.



b.

Figure 2: **a.** The Serviced Workload $W_n(t)$ under FPF (Top) and under LDL (Bottom) and the Released Workload $V_n(t)$ *vs.* Time. **b.** The Idle Capacity Process $Z_n(t)$ *vs.* Time, Under LDL (Top) and FPF (bottom). Synthetic Periodic Task Set TS1, the First two Hyper-periods ($H = 10sec$)

time concentrates towards the right end of each hyper-period. An aperiodic task J_a arriving, for instance, at $r_a = 0$ and with deadline at $d_a = 400$, would have to be rejected outright due to the complete absence of idle capacity during $[0, 400)$.

3.3.2 The Latest-Deadline-Last (LDL) Idling Schedule

We now consider a fixed priority, *idling* schedule for hard real-time periodic tasks we call *Latest-Deadline-Last* (LDL). In this schedule, denoted by $\mathcal{S}^{\mathrm{L}}$, we use the same fixed priority assignment as its FPF counterpart. Jobs τ_{ij} arrive at their release times r_{ij}, but they are *delayed*, each one by a different amount of time, after which they start again being dispatched by the regular preemptive, fixed-priority rules. LDL starts jobs as *late* as possible, while none misses its deadline. LDL is an idling schedule with respect to the periodic task set. We will show in the sequel how we can utilize the newly created idle periods to accommodate hard aperiodic tasks.

We derive $\mathcal{S}^{\mathrm{L}}$, and the other quantities of interest $s_i(t), W_i(t), Z_i(t),\ i = 1, 2, \ldots, n$. We proceed inductively, task-by-task, starting with the schedule $s_1(t),\ t > 0$, of τ_1 and then continuing with $\tau_2, \ldots, \tau_n$. Specifically, setting $V_0(t) = W_0(t) = 0$, for each $t \geq 0$, and $j = n_i(t)$, let

$$s_i(t) := \begin{cases} 1, & \text{if } i = 1, \text{ for } t \in [d_{ij} - C_1, d_{ij}), \\ 1, & \text{if } i > 1,\ \ Z_{i-1}(d_{ij}) - Z_{i-1}(t) < C_i, \text{ and } \mathcal{S}^{\mathrm{L}}_{i-1}(t) = 0, \\ & \text{for } t \in [r_{ij}, d_{ij}), \\ 0, & \text{otherwise;} \end{cases} \tag{10}$$

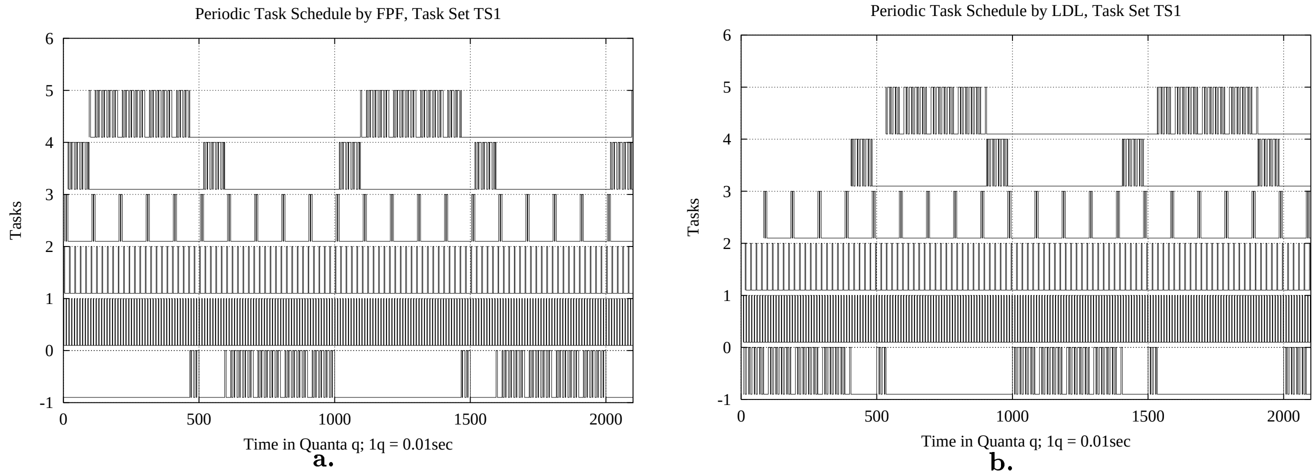


Figure 3: The Eager FPF (**a.**) and Lazy LDL (**b.**) Schedules for Periodic Task Set TS1, in the First 2 Hyper-periods ($H = 10sec$)

$$w_i(t) = \int_0^t s_i(s)ds = \sum_{k=1}^{j-1} \int_{r_{ik}}^{d_{ik}} s_i(x)dx + \int_{r_{ij}}^{t} s_i(x)dx = (j-1) \cdot C_i + c_i(t) \tag{11}$$

$$c_i(t) = \begin{cases} C_i - \min\{[d_{ij} - t]^+, C_i\}, & \text{if } i = 1, t \in [r_{ij}, d_{ij}) \text{ and } (D_i - C_i) \leq m_i(t) \leq D_i, \\ C_i - \min\{[Z_{i-1}(d_{ij}) - Z_{i-1}(t)]^+, C_i\} & \text{if } i > 1, t \in [r_{ij}, d_{ij}), \\ 0, & \text{otherwise.} \end{cases} \tag{12}$$

As usual, $W_i(t) = W_{i-1}(t) + w_i(t)$, $Z_i(t) = Z_{i-1}(t) - w_i(t)$ and $\mathcal{S}_i^{\mathrm{L}}(t) = \sum_{k=1}^{i} k \cdot s_k(t)$. As Eq. (10) indicates, LDL allocates the *last* C_i units of idle processor capacity in the interval $[r_{ij}, d_{ij})$, to job τ_{ij}. Effectively, this *delays* each job for as long as possible, without letting it miss its deadline. When τ_{ij} is released at time r_{ij} the dispatcher puts it into a *delay queue* until job start time s_{ij}. At that time it starts subjecting it again to regular fixed-priority dispatching. In order to implement the LDL scheduler, we need to remember the job start times. We can obtain start s_{ij} and finish f_{ij} times, respectively, solving,

$$s_{ij} = \max\{x : x \leq d_{ij},\ Z_{i-1}(d_{ij}) - Z_{i-1}(x) = C_i\}, \tag{13}$$

$$f_{ij} = \min\{x : x \geq (s_{ij} + C_i),\ Z_{i-1}(x) - Z_{i-1}(s_{ij}) = C_i\}, \tag{14}$$

for x with a fixed-point iteration. If $\boldsymbol{\tau}$ is schedulable, the iteration converges, for discrete time schedules in at most $(d_{ij} - r_{ij})$ steps, and for continuous time schedules in finite time, since we do not allow instantaneous processor assignments.

Fig. 2**a.** plots the serviced $W_n(t)$ workload under LDL (bottom curve), in the first two hyper-periods ($H = 1000$) of the same synthetic periodic set TS1, defined in Table 1. Under FPF, $W_n(t)$ (middle curve in Fig. 2**a.**) starts increasing immediately after the release of periodic jobs. It remains increasing with slope 1 while the unfinished workload $V_n(t) - W_n(t) > 0$. Under LDL the serviced workload initially progresses at a slower pace but as it approaches the end of a hyper-period it becomes increasing with slope equal to 1. Fig. 2**b.** shows the corresponding idle capacity curves $Z_n(t)$ under LDL (top) and FPF (bottom). We notice that, contrary to its FPF counterpart, LDL idle capacity starts accumulating much earlier in a hyper-period.

3.4 Workload Theorems

Theorem 1 (Workload Dominance) *Given a periodic task system $\boldsymbol{\tau}$, let $W_i^{\mathrm{F}}(t)$ and $W_i^{\mathrm{L}}(t)$ denote the serviced workload functions under the $\mathcal{S}^{\mathrm{F}}$* (FPF) *and $\mathcal{S}^{\mathrm{L}}$* (LDL) *schedules, respectively. The following holds*

$$W_i^{\mathrm{F}}(t) \geq W_i^{\mathrm{L}}(t), \tag{15}$$

for all $i = 1, 2, \ldots, n$ and $t > 0$.

Proof: From (6) and (11), for all $t \geq 0$, $i = 1, 2, \ldots, n$, we obtain

$$W_i^{\mathrm{F}}(t) = \sum_{j=1}^{i} \big((n_j(t) - 1) \cdot C_j + c_j^{\mathrm{F}}(t) \big),$$
$$W_i^{\mathrm{L}}(t) = \sum_{j=1}^{i} \big((n_j(t) - 1) \cdot C_j + c_j^{\mathrm{L}}(t) \big),$$

where, $c_i^{\mathrm{F}}(t)$ and $c_i^{\mathrm{L}}(t)$, are the serviced workloads for τ_i in the currently active periods, under FPF and LDL, respectively. We will use mathematical induction on the number of periodic tasks.

Basis $[i = 1]$ For any $t \geq r_1$, let $t = r_1 + (j-1)T_1 + x$, where $j = n_1(t)$ and $x = m_1(t)$, $0 \leq x < T_1$, we have, $W_1^S(t) = (n_1(t) - 1) \cdot C_1 + c_1^S(t)$, where $S =$ F or L and

$$c_1^{\mathrm{F}}(r_{1j} + x) = \min\big\{r_{1j} + x - r_{1j}, C_1\big\} = \min\big\{x, C_1\big\}, \text{ and,}$$
$$c_1^{\mathrm{L}}(r_{1j} + x) = C_1 - \min\big\{[r_{1j} + D_1 - (r_{1j} + x)]^+, C_1\big\} = C_1 - \min\big\{D - x, C_1\big\}.$$

It is easy to verify that, for all r_{1j} and x, with $r_1 \leq x < T_1$, $c_1^{\mathrm{F}}(r_{1j} + x) \geq c_1^{\mathrm{L}}(r_{1j} + x)$, and thus, $c_1^{\mathrm{F}}(t) \geq c_1^{\mathrm{L}}(t)$ and $W_i^{\mathrm{F}}(t) \geq W_i^{\mathrm{L}}(t)$.

Inductive Hypothesis $[i = k-1]$ We assume that

$$W_i^{\mathrm{F}}(t) \geq W_i^{\mathrm{L}}(t)$$

for all $t \geq r_{k-1}$ and all $i = 1, \ldots, k-1$, $k \leq n$. The only way the theorem can become false is when there is a point where the $W_k^{\mathrm{F}}(t)$ and $W_k^{\mathrm{F}}(t)$ meet and the latter one becomes larger than the former one. Without loss of generality, we assume the first point this happens is $y \in [r_{k,n_k(y)}, d_{k,n_k(y)})$ where $W_k^{\mathrm{F}}(y) = W_k^{\mathrm{L}}(y)$ and for which $W_k^{\mathrm{F}}(x) > W_k^{\mathrm{L}}(x)$, for all $x \in [r_{k,n_k(y)}, y)$ and $W_k^{\mathrm{F}}(x) < W_k^{\mathrm{L}}(x)$, for some $x > y$. Since both $W_k^{\mathrm{F}}(t)$ and $W_k^{\mathrm{L}}(t)$ are piecewise continuous, with line segments which either are parallel to the x-axis or increase with a slope of 45^o, it must be the case that for $t \in [y - \varepsilon, y + \varepsilon)$, $\varepsilon > 0$, $W_k^{\mathrm{F}}(t)$ is parallel to the the x-axis while $W_k^{\mathrm{L}}(t)$ is increasing and they intersect at $t = y$. However, when $W_k^{\rm F}(t)$ is parallel to the the x-axis the unfinished workload $V_k(y) - W_k^{\rm F}(y) = 0$, and similarly, since $W_k^{\rm L}(t)$ is increasing at $t = y$ the unfinished workload $V_k(y) - W_k^{\rm L}(y) > 0$. This would imply that $W_k^{\rm F}(y) > W_k^{\rm L}(y)$, which is a contradiction of our assumption that $W_k^{\rm F}(y) = W_k^{\rm L}(y)$. Therefore, since the assumption that there is a point y after which $W_k^{\rm F}(y+\varepsilon) < W_k^{\rm L}(y+\varepsilon)$, $\varepsilon > 0$ leads to a contradiction, we conclude that it is not possible that $W_k^{\rm F}(t) < W_k^{\rm L}(t)$. ■

Corollary 1 *Let $Z_i^{\rm F}(t)$ and $Z_i^{\rm L}(t)$ denote the idle capacity functions on the $\mathcal{S}^{\rm F}$ and $\mathcal{S}^{\rm L}$ schedules, respectively. Then, for all $t \geq 0$ and all $i = 1, 2, \ldots, n$, the following is true,*

$$Z_i^{\rm F}(t) \leq Z_i^{\rm L}(t). \tag{16}$$

Proof: It follows immediately from Theorem 1. ■

Theorem 2 (Workload Switch) *Given a periodic task system $\boldsymbol{\tau}$, assume that the system dispatches the periodic tasks in the interval $[0, a)$ under* FPF, *at a it starts dispatching the tasks by the* LDL *schedule and it continues to do so until time $t > a$. Then the serviced workload in $[a, t)$, denoted as $dW_n^{\rm FL}(a, t)$ is less than or equal to the workload serviced had the system remained under the FPF schedule, denoted as $dW_n^{\rm FF}(a, t)$, that is*

$$dW_i^{\rm FL}(a, t) \leq dW_i^{\rm FF}(a, t) \tag{17}$$

for all $i = 1, 2, \ldots, n$ and $t > a \geq 0$.

Proof: We have that

$$dW_i^{\rm FF}(a, t) = \sum_{k=1}^{i} (n_k(t) - n_k(a))C_k + \left[c_k^{\rm F}(t) - c_k^{\rm F}(a)\right]^+$$

$$dW_i^{\rm FL}(a, t) = \sum_{k=1}^{i} (n_k(t) - n_k(a))C_k + \left[c_k^{\rm L}(t) - c_k^{\rm F}(a)\right]^+$$

where, by Theorem 1, $c_k^{\rm F}(t) \geq c_k^{\rm L}(t)$. Therefore, $dW_i^{\rm FL}(a, t) \leq dW_i^{\rm FF}(a, t)$.

■

Corollary 1 and Theorem 2 have important implications in the performance of an admission control scheme of a periodic, hard real-time system. Regularly, in the absence of aperiodic tasks, a periodic task set is dispatched according to the FPF method. When a hard aperiodic $J_a(C_a, D_a, r_a)$ arrives, if $Z_n^{\rm F}(r_a + D_a) - Z_n^{\rm F}(r_a) < C_a$, where $d_a = r_a + D_a$, there is no sufficient idle capacity to guarantee J_a and it has to be rejected. However, by Theorem 2, $d_a - r_a - dW_n^{\rm FL}(r_a, d_a) \geq d_a - r_a - dW_n^{\rm FF}(r_a, d_a)$ making possible that $d_a - r_a - dW_n^{\rm FL}(r_a, d_a) \geq C_a$, thus, making J_a admissible. Intuitively, Theorem 2 suggests that, if until time r_a periodic tasks execute under FPF but within the time interval $[r_a, d_a)$ we execute them under the LDL schedule, we can only *increase* or (leave the same) the quantity of idle time within $[r_a, d_a)$. LDL delays the execution of periodic tasks with the objective to relocate the idle capacity to temporal locations that may be needed most for the firm aperiodic tasks.

4 Computation Algorithms

In this section we present two classes of algorithms for the direct computation of $W_i(t)$. First, we provide a recursive method to compute $W_i(t)$ dynamically. Second, we derive the *Workload Matrix* method that leads to algorithms that compute $W_i(t)$ in linear time. All algorithms compute the $W_i(t)$ of FPF and LDL, and the corresponding idle capacity $Z_i(t)$, precisely. There is a third method which obtains $W_i(t)$ for schedules with discrete parameters. Further details of this method can be found in[34]. Table 2 summarizes the time and storage asymptotic complexities of all three methods.

Table 2: Space and Time Complexity for Proposed FPF and LDL Algorithms

Algorithm	When	Operation	Time	Storage
Static	Off-line	$\mathcal{S}(t), W_n(t)$ generation	$\Theta(H)$	$\Theta(H)$
	On-line	Idle Capacity Computation	$\Theta(1)$	$\Theta(H)$
Recursive	On-line	Idle Capacity Computation	$\Theta(2^n)$	$\Theta(n)$
Workload Matrix	Off-line	$W[k][t_{k,n_k(t)}]$ generation	See text	N
	On-line	Idle Capacity Computation	$\Theta(n)$	N

The algorithms we develop here do not require the task parameters to be integers. Several methods proposed in the past proceed by first discretizing the time line, resulting an explosion in the size of hyper-period H. The algorithms here do no suffer from this drawback. In schedulable task sets $\boldsymbol{\tau}$, the sequence of scheduling decisions and actions $\mathcal{A}(t)$ are identical within each hyper-period, that is, $\mathcal{A}(t) = \mathcal{A}(t + j \cdot H),\ j = 0, 1, \ldots$. The released processes $V_i(t)$, are cyclic, with identical increments. For any fixed priority schedule, as for instance with FPF or LDL, the scheduling algorithm generates identical schedule $s_i(t)$ for each task τ_i, given that within the hyper-periods the workload is released in identical patterns. The serviced processes $W_i(t)$ become cyclic as well, with identical increments that repeat in every hyper-period. See Subsection III.C.3 in [32] for a discussion of the cyclicity properties of the workload processes. If the task set is synchronous, the algorithms need to analyze the schedule of the first hyper-period, that is, the interval $[0, H]$. For asynchronous task sets, they need to analyze the schedule within $[0, r + 2H]$, where $r = \max_{i=1}^{n}\{r_i\}$. Intuitively, synchronous periodic schedules repeat identically in every hyper-period. In the asynchronous task case, the schedule again repeats in every hyper-period, but the repetition starts as late as time $r + H$.

From Eq. (6), (7) for FPF and Eq. (11), (12) for LDL of Section 3.3, the serviced workload process in given by,

$$W_i^S(t) = \sum_{k=1}^{i}(n_k(t) - 1) \cdot C_k + c_k^S(t), \tag{18}$$

where $S = \mathrm{F}$ (FPF), or $S = \mathrm{L}$ (LDL). The most critical part is the computation of $c_k^S(t)$, which is the amount of completed work for job $\tau_{k,n_k(t)}$ in the current request period at time t. Specifically, for $k \geq 1, t \in [r_{kj}, d_{kj})$,

$$\begin{aligned} c_k^{\mathrm{F}} &= \min\{[Z_{k-1}^{\mathrm{F}}(t) - Z_{k-1}^{\mathrm{F}}(r_{kj})]^+, C_k\} \\ &= \min\{\left[(t - r_{k,n_k(t)}) - \left(W_{k-1}^{\mathrm{F}}(t) - W_{k-1}^{\mathrm{F}}(r_{k,n_k(t)})\right)\right]^+, C_k\} \end{aligned} \tag{19}$$

and for $k \geq 1, t \in [r_{kj}, d_{kj})$,

$$\begin{aligned} c_k^{\mathrm{L}} &= C_k - \min\{[Z_{k-1}^{\mathrm{L}}(d_{kj}) - Z_{k-1}^{\mathrm{L}}(t)]^+, C_k\} \\ &= C_k - \min\{\left[(d_{k,n_k(t)} - t) - \left(W_{k-1}^{\mathrm{L}}(d_{k,n_k(t)}) - W_{k-1}^{\mathrm{L}}(t)\right)\right]^+, C_k\} \end{aligned} \tag{20}$$

where $j = n_k(t)$ and $Z_0^{\mathrm{L}}(t) := 0$ for all t.

4.1 Recursive Computation of $W_i(t)$

In this subsection we develop algorithms to compute on-line the $W_i(t)$ directly, without relying on any pre-computed information. They are a direct formulation of the recurrent equations on $W_i(t)$ from Section 3.3. The main disadvantage of this approach is the time complexity which is exponential in n, the number of periodic tasks.

In Eq. (19) and (20) $c_k^S(t)$ denotes the total processor service job $\tau_{k,n_k(t)}$ has received within the interval $[r_{k,n_k(t)}, t]$ in schedule $\mathcal{S}^S$. Eq. (19) and (20) give rise to a recurrence relationship for $c_k^S(t)$ in terms of $W_j^S(t_j)$, $j = k-1, k-2, \ldots, 1$. The recursive computation of $c_k^S(t)$ is finite, as it eventually depends on $c_1^S(x)$ which are given by by Eq. (7) or Eq. (12), for FPF or LDL, respectively. The direct computation of $W_n^S(t)$, based on the recursions of Eq. (19) and (20) has time complexity $T_W^{\mathrm{rec}}(n) = \Theta(2^n)$, and this makes it impractical for implementation in actual systems. It gives us insight, though, into the inherent complexity of the problem of direct computation of $W_i(t)$.

The pseudo-code for the recursive algorithm algorithm for FPF $W_i(t)$ is shown in Fig. 4, and for LDL in Fig. 5. The input is a periodic task set $\boldsymbol{\tau}$, n and time t. The output is the value of $W_n(t)$. These algorithms are direct implementations of the recursive formulations for $W_i(t)$. Their complexity is $\Theta(2^n)$, and thus this implementation is naive.

4.2 Workload Matrix-based $W_i(t)$ Computation

In this subsection we derive the *Workload Matrix* algorithm for the efficient computation $W_n^S(t)$, under FPF and LDL. The culprit for the excessive computation in the recursive method is the double-recurrence for $c_k^S(t)$ in Eq. (19) and (20). Upon closer examination it becomes clear that $c_k^{\mathrm{F}}(t)$ depends on W_{k-1}^{F} evaluated at points t and $r_{k,n_k(t)}$. The points $r_{k,n_k(t)}$ are the release times of the jobs $\tau_{k,n_k(t)}$ and are well defined. We propose an

Dynamic FPF $W_n(t)$ Computation Algorithm

Function $W(n,t)$
begin
    $w = 0;$
    for $(k = 1, 2, \ldots, n)$ **do** $\Theta(2^n)$
        $w = w + \big((n_k(t) - 1)C_k + c(k,t)\big);$
    endfor
    return $(w);$
end

Function $c(k,t)$
begin
    $c_t = 0$
    if $(k = 1)$ **then** $c_t = \min\big\{[t - r_{1,n_1(t)}]^+, C_1\big\};$
    else
        $c_t = \min\big\{[(t - r_{k,n_k(t)}) - \big(W_{k-1}(t) - W(k-1, r_{k,n_k(t)})\big)]^+, C_k\big\};$ $\Theta(2^{k-1})$
    endif
    return $(c_t);$
end

Figure 4: The Pseudo-code of the Dynamic Recursive FPF $W_n(t)$ Computation Algorithm

efficient on-line $W_n^{\mathrm{F}}(t)$ computation procedure, having two steps. First, we *pre-compute* off-line $W_{k-1}^{\mathrm{F}}(r_{k,j_k})$, for $k = 2, 3, \ldots, n$, $j_k = 1, 2, \ldots, n_k(H)$, and store them in a matrix $\boldsymbol{W}^{\mathrm{F}}[k, j_k]$. We need not store W_1^{F} which can be computed directly. At run-time, setting $W_0^{\mathrm{F}}(t) = \boldsymbol{W}^{\mathrm{F}}[0, j_0] = 0$, we compute $W_n^{\mathrm{F}}(t)$ by

$$\begin{aligned} W_n^{\mathrm{F}}(t) &= \sum_{k=1}^{i} (n_k(t) - 1) \cdot C_k + c_k^{\mathrm{F}}(t) \\ c_k^{\mathrm{F}}(t) &= \min\Big\{ \big[(t - r_{k,n_k(t)}) - (W_{k-1}^{\mathrm{F}}(t) - \boldsymbol{W}^{\mathrm{F}}[k, n_k(t)])\big]^+, C_k \Big\}. \end{aligned} \tag{21}$$

The time complexity of the Workload Matrix method is $T_W^{\mathrm{wm}}(n) = \Theta(n)$. The storage complexity $S_W^{\mathrm{wm}}(n) = \Theta(\sum_{k=2}^{n} n_k(H)) = N$. We can even avoid storing the $W_2^{\mathrm{F}}(r_{3,j_k})$ since W_2^{F} can be directly computed.

In a similar fashion under LDL, $c_k^{\mathrm{L}}(t)$ depends on W_{k-1}^{L} evaluated at points t and $d_{k,n_k(t)}$, which are the deadlines of the jobs $\tau_{k,n_k(t)}$, and are also well defined. The LDL Workload matrix algorithm similarly pre-computes off-line, $W_{k-1}^{\mathrm{L}}(d_{k,j_k})$ for $k = 2, 3, \ldots, n$, $j_k = 1, 2, \ldots, n_k(H)$, and stores them in a matrix $\boldsymbol{W}^{\mathrm{L}}[k, j_k]$. At run time, setting $W_0^{\mathrm{L}}(t) = \boldsymbol{W}^{\mathrm{L}}[0, j_0] = 0$, it computes $W_n^{\mathrm{L}}(t)$ by

$$\begin{aligned} W_n^{\mathrm{L}}(t) &= \sum_{k=1}^{i} (n_k(t) - 1) \cdot C_k + C_k - \overline{c}_k^{\mathrm{L}}(t) \\ \overline{c}_k^{\mathrm{L}}(t) &= \min\Big\{ \big[(d_{k,n_k(t)} - t) - (\boldsymbol{W}^{\mathrm{L}}[k, n_k(t)] - W_{k-1}^{\mathrm{L}}(t))\big]^+, C_k \Big\}. \end{aligned} \tag{22}$$

This method has time complexity $T_W^{\mathrm{wm}}(n) = \Theta(n)$, and storage complexity $S_W^{\mathrm{wm}}(n) = \Theta(\sum_{k=2}^{n} n_k(H))$, as its FPF counterpart. To our knowledge this is the first linear time algorithm that computes idle capacity at any

Dynamic LDL $W_n(t)$ Computation Algorithm

```
Function W(n,t)
begin
    w = 0;
    for (k = 1,2,...,n) do                                          Θ(2^n)
        w = w + ((n_k(t) − 1)C_k + c(k,t));
    endfor
    return (w);
end

Function c(k,t)
begin
    c_t = 0
    if (k = 1) then c_t = min{ [(d_{1,n_1(t)} − t)]^+ , C_1 };
    else
        c_t = min{ [(d_{k,n_k(t)} − t) − (W(k−1, d_{n_k(t)}) − W(k−1,t))]^+ , C_k };   Θ(2^{k−1})
    endif
    return (C_k − c_t);
end
```

Figure 5: The Pseudo-code of the Dynamic Recursive LDL $W_n(t)$ Computation Algorithm

arbitrary point t, exactly. The Workload Matrix methods has very low actual execution time as the experiments in Section 5 demonstrate, making it an excellent algorithm for the feasibility testing of firm aperiodic tasks.

Fig. 6 and Fig. 7 show the details of the $W_i(t)$ computation for FPF and LDL, respectively, by the workload matrix method. Note that we use the same routines to initialize the entries of the workload matrix itself, one task and one row at a time. Fig. 8 and Fig. 9 show the details of the computation of the $dW_i^{\mathrm{F}}(s,t)$ and $dW_i^{\mathrm{L}}(s,t)$, respectively. $dW_i^S(s,t)$ is the total time the processor is busy in $[s,t)$ serving jobs of the first i periodic tasks.

Here we present the time it takes to generate off-line the workload-matrix on a $333MHz$ Ultra-Sparc 5 SUN workstation, for each one of the example task sets, defined in Appendix B. Table 3 presents the generation times in seconds and the size of each matrix in memory locations. We notice in Table 3 that the matrix size

Table 3: Workload-Matrix Generation Times and Matrix Sizes

Name	Time	Size	Name	Time	Size
TS1	0.000063289	164	TS2	0.000090236	204
TS3	0.001493509	2373	T1	0.000162340	351
T2	0.000147966	270	T3	0.000106712	111
T4	0.027065545	21656	T5	0.000669799	1897
T7	0.000156195	284	T71	0.000701562	748

depends on the number of periodic tasks in the set and on the set parameters, as well. For modest task sets

Dynamic FPF $W_n(t)$ Computation Algorithm by the Workload Matrix

Proc *FPF-Eff-W*(t)
begin
  $c_t = \min\{[(t - r_{1,n_1(t)})]^+, C_1\}$;
  $V_t = n_1(t) \cdot C_1$;
  $W_t = (V_t - C_1) + c_t$;
  for $(k = 2, 3, \ldots, n)$ **do** $\Theta(n)$
    $c_t = \min\left\{\left[(t - r_{k,n_k(t)}) - \left(W_t - \boldsymbol{W}[k, r_{n_k(t)}]\right)\right]^+, C_k\right\}$;
    $v_t = n_k(t) \cdot C_k$;
    $W_t = W_t + (v_t - C_k) + c_t$;
    $V_t = V_t + v_t$;
  endfor
  return (W_t);
end

Figure 6: The Pseudocode of the Dynamic $W_n(t)$ Computation Algorithm Based on the Workload Matrix Method, for FPF

the matrix generation time is relatively low. This makes it possible to even generate a matrix on-line, in case dynamic adjustments in the periodic workload are necessary.

5 Feasibility Testing Experiments

We have performed a number of experiments to evaluate the proposed feasibility testing schemes. In the first experiment we illustrate the increase of idle processor capacity as the periodic schedule switches from FPF to LDL. In Fig. 10 the set of periodic tasks, presented in the beginning of the paper, is switched two times each time from FPF to LDL, and then back to FPF. Fig. 11 presents the idle capacity $Z_n(t)$ and Fig. 12 the corresponding serviced workload $W_n(t)$ processes, respectively. In Fig. 11 the top and the bottom curves of the envelope represent the idle process Z_n^{L} of LDL and Z_n^{F} of FPF, respectively. Before any non-periodic task arrives at the system, periodic tasks run under the FPF schedule. At time instant $t = 1200$, when the schedule is switched from FPF to LDL for the first time, the available processor capacity begins to increase, as the middle line shows, between the top and bottom curves of the envelop in Fig. 11. We note that when at time $t = 1200$ periodic tasks switch to the LDL rule, no task misses its deadline. After the task set is switched back (from LDL) to FPF at time $t = 1400$, the idle capacity function resumes its growth pattern of the FPF mode. Similar results are observed when the non-periodic task is inserted for the second time, during the time interval $[2200, 2400]$. Fig. 12 shows the corresponding changes on the workload process. Until time $t = 1200$, the system services the periodic tasks as early as possible under the FPF schedule (top curve). When a non-periodic task arrives the service rate to the periodic tasks starts decreasing, as the scheduler under the LDL schedule

Dynamic LDL $W_n(t)$ Computation Algorithm by the Workload Matrix

Proc *LDL-Eff-W*(t)
begin
  $\overline{c}_t = \min\{[(d_{1,n_1(t)} - t)]^+, C_1\}$;
  $V_t = n_1(t) \cdot C_1$;
  $W_t = V_t - \overline{c}_t$;
  for $(k = 2, 3, \ldots, n)$ **do** $\Theta(n)$
    $\overline{c}_t = \min\left\{ \left[(d_{k,n_k(t)} - t) - \left(\boldsymbol{W}[k, d_{n_k(t)}] - W_t\right)\right]^+, C_k\right\}$;
    $v_t = n_k(t) \cdot C_k$;
    $W_t = W_t + v_t - \overline{c}_t$;
    $V_t = V_t + v_t$;
  endfor
  return (W_t);
end

Figure 7: The Pseudocode of the Dynamic $W_n(t)$ Computation Algorithm Based on the Workload Matrix Method for LDL

lets them execute at their latest possible points in time. It is worth noting that, in the initial FPF schedule (Fig. 2) there is no available capacity in $[1000, 1466]$ or in $[2000, 2466]$, at all. Any aperiodic task with arrival and deadline time within these intervals would had to be rejected. We notice in Fig. 11 that when we switch at $t = 1200$ and $t = 3200$ to LDL, idle capacity increases in these intervals.

5.1 Increase of Idle Capacity within Arbitrary Intervals

The objective here is to measure the quantity of idle capacity $Z(r_a, d_a)$ as seen by aperiodic tasks J_a when they are released at r_a. We focus on firm aperiodic tasks and therefore need to provide guarantee to the task upon its release. The scheduler at r_a computes the amount of idle capacity of the periodic schedule in the interval $[r_a, d_a]$, where $d_a = r_a + D_a$. If the periodic tasks are currently serviced under the FPF and there sufficient capacity, $Z^{\mathrm{F}}(r_a, d_a) \geq C_a$, the scheduler verifies that the pending aperiodic tasks remain feasible, admits J_a and assigns it priority according to its rank in the queue of pending aperiodic tasks. If there is no sufficient capacity, the scheduler evaluates the idle capacity as if a schedule switch would take place at r_a. That is, the scheduler evaluates $Z(r_a, d_a)^{\mathrm{FL}} = D_a - dW^{\mathrm{FL}}(r_a, d_a)$, and if $Z(r_a, d_a^{\mathrm{FL}}) \geq C_a$, it verifies that the pending aperiodic tasks remain feasible, accepts J_a and assigns it a priority according to its ranking relative to the other pending aperiodic tasks. All aperiodic tasks receive priorities lower than the periodic task priorities. Finally, if there is no sufficient capacity, the scheduler rejects J_a.

In Fig. 13 through 20 we plot the experimental results on three synthetic and four actual hard real-time periodic sets, defined in Appendix B. In each experiment we let the workload state of the system evolve and

Computation of Exact FPF Workload Difference Between Two Arbitrary Points

Function $dW_n^{\mathrm{F}}(\boldsymbol{\tau}, \boldsymbol{\xi}(t_1), n, t_1, t_2)$
begin
$c_{t2} = \min\{[(t_2 - r_{1,n_1(t_2)})]^+, C_1\};$
$V_{t2} = n_1(t_2) \cdot C_1;\ V_{t1} = n_1(t_1) \cdot C_1;$
$W_{t2} = (V_{t2} - C_1) + c_{t2};\ W_{t1} = (V_{t1} - C_1) + \xi_1(t_1);$
$dW = [W_{t2} - W_{t1}]^+;$
for $(k = 2, 3, \ldots, n)$ **do** $\Theta(n)$
$c_{t2} = \min\left\{ \left[(t_2 - r_{k,n_k(t_2)}) - \left(W_{t2} - \boldsymbol{W}^F[k, r_{n_k(t_2)}] \right) \right]^+, C_k \right\};$
$v_{t2} = n_k(t_2) \cdot C_k;\ v_{t1} = n_k(t_1) \cdot C_k;$
$W_{t2} = W_{t2} + (v_{t2} - C_k) + c_{t2};\ W_{t1} = W_{t1} + (v_{t1} - C_k) + \xi_k(t_1);$
$V_{t2} = V_{t2} + v_{t2};\ V_{t1} = V_{t1} + v_{t1};$
$dW = dW + [W_{t2} - W_{t1}]^+;$
endfor
return (dW);
end

Figure 8: Computation of the Workload Difference $dW_n^{\mathrm{F}}(t_1, t_2)$ based on the Workload Matrix Method

firm aperiodic tasks arrive and request service. At the arrival times r_a we measure the available idle capacity in the interval $[r_a, d_a]$. As aperiodic tasks arrive the workload state evolves accordingly. The figures here present the amount of idle capacity within the request intervals of the firm tasks, $[t_i, t_{i+1}]$ when either we switch from FPF to LDL at time t_i or we remain in FPF until the end of the interval. Each figure plots two curves, marked dZ^{FF} and dZ^{FL}, respectively. Each jump point is the arrival of a firm task. The first curve plots at the point of a jump, the amount of idle capacity in $[r_a, d_a]$ if the schedule were to remain in the FPF mode. The second curve, marked Z^{FL}, shows the corresponding idle capacity, had the scheduler decided to switch to LDL at r_a.

These experiments demonstrate that in the interval that follows a switch from FPF to LDL we always maximize the amount of spare capacity within that same interval and we are able to guarantee more firm aperiodic tasks. The task sets we use have different utilization ratios and number of periodic tasks. In all cases when we switch from FPF to LDL we increase the amount of idle capacity in the interval that follows. It is worth noting that the increase in the amount of idle capacity in the interval following the switch depends on the parameters of the periodic task set and the current workload state of the system at the switch time t_1. The idle capacity increase in the experiments of Fig. 13, 14 and 15 vary from small to moderate. As we move to Fig. 16 and 17 the difference in idle capacity is relatively large. In Fig. 18 and 20 we notice that without switch to an idling schedule, the idle capacity is zero. This is the case where non-idling schedules service the unfinished workload in the periodic ready queue over lengthy busy periods. Although it is not shown on the figures, almost all firm aperiodic tasks have been rejected. However, by switching to LDL guarantees all the firm tasks. As a conclusion, schedule switching can relocate the idle capacity very effectively toward the intervals that aperiodic

Computation of Exact LDL Workload Difference Between Two Arbitrary Points

Function $dW^{\mathrm{L}}_n(\boldsymbol{\tau}, \boldsymbol{\xi}(t_1), n, t_1, t_2)$
begin
$\overline{c}_{t2} = \min\{[(d_{1,n_1(t_2)} - t_2)]^+, C_1\};$
$V_{t2} = n_1(t_2) \cdot C_1;\ V_{t1} = n_1(t_1) \cdot C_1;$
$W_{t2} = V_{t2} - \overline{c}_{t2};\ W_{t1} = (V_{t1} - C_1) + \xi_1(t_1);$
$dW = [W_{t2} - W_{t1}]^+;$
for $(k = 2, 3, \ldots, n)$ **do** $\Theta(n)$
$\overline{c}_{t2} = \min\left\{\left[(d_{k,n_k(t_2)} - t_2) - \left(\boldsymbol{W}^L[k, d_{n_k(t_2)}] - W_{t2}\right)\right]^+, C_k\right\};$
$v_{t2} = n_k(t_2) \cdot C_k;\ v_{t1} = n_k(t_1) \cdot C_k;$
$W_{t2} = W_{t2} + v_{t2} - \overline{c}_{t2};\ W_{t1} = W_{t1} + (v_{t1} - C_k) + \xi_k(t_1);$
$V_{t2} = V_{t2} + v_{t2};\ V_{t1} = V_{t1} + v_{t1};$
$dW = dW + [W_{t2} - W_{t1}]^+;$
endfor
return (dW);
end

Figure 9: Computation of the Workload Difference $dW^{\mathrm{L}}_n(t_1, t_2)$ based on the Workload Matrix Method

tasks require it, while not violating the deadlines of the hard periodic tasks, and thus maximizes the number of guaranteed firm aperiodic tasks. As we demonstrate in subsequent subsection in this paper, the admission control algorithms require insignificant amount of computation to make a decision. This allows large numbers of aperiodic tasks to be tested for feasibility and guaranteed by the scheduler.

5.2 Feasibility Testing Run-Time Overhead

In this set of experiments we perform $W^{\mathrm{F}}_n(t)$ and $W^{\mathrm{L}}_n(t)$ computations on several UNIX machines, and measure the actual execution times. The times are almost identical for $W^{\mathrm{F}}_n(t)$ and $W^{\mathrm{L}}_n(t)$ and we report the ones for $W^{\mathrm{F}}_n(t)$ only. These experiments were executed under the real-time and the time-sharing scheduling classes of UNIX, commonly found in SVR4 and POSIX compliant kernels [35], [36], [37], [38], [39], [40], [41], [42]. In Fig. 21 we evaluate the growth of the execution time as the number of periodic tasks increases, from $n = 10$ to $n = 160$. With 160 tasks the fastest machine requires 28μ-seconds and the slowest 85μ-seconds. We computed the $W^S_n(t)$ for a number of known periodic task sets, such as, the Inertial Navigation, the Submarine Sonar, the Surface Ship Radar, the Avionics and the Multimedia System. Information on these sets can be found in Section B of the Appendix. All calculations for Collection 2, ran at under 10μ-seconds, even for the slowest machine as Fig. 22 shows. The experiments of Fig. 21 and 22 were conducted under the real-time scheduling class. We performed the same set of experiments under the time-sharing class of the same systems as well. Fig. 23 plots the execution time measurements for Collection 1 and Fig. 24 for Collection 2. We notice that

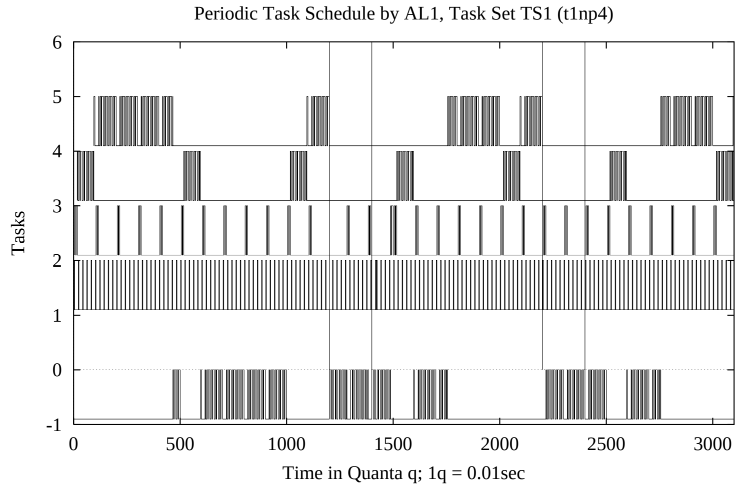


Figure 10: The Resulting Schedule Switchings for the Set TS1. Task τ_1 Is Not Shown for Clarity

under the real-time class, the computation is smoother, whereas under the time-sharing class the execution times experience higher variability. The computation of $W_n^S(t)$ is essentially the time the system needs to make the admission control decision for the incoming task J_{ak}. The extremely low actual overhead demonstrates the efficiency and the suitability of the method for scheduling firm real-time tasks in highly dynamic environments dominated by task arrivals at high rates.

The actual time to carry out scheduling decisions, in our methods, is very short. The potential of the scheduler can be realized in an environment were the underlying operating system does not incur prohibitive restrictions or overhead. Specifically, the kernel support for dispatching, timer management and process or thread management should have overhead and allow fine control at the level of our scheduler. Unfortunately, POSIX based systems do not offer this performance or control levels.

6 Discussion and Future Work

In this paper we have presented a framework for the analysis and development of efficient feasibility testing algorithms, utilized in the admission control of guaranteed, firm-aperiodic tasks for preemptive, fixed-priority systems. This work makes a number of basic contributions in the field of on-line schedulability testing algorithms and fixed-priority scheduling, in general. The first contribution is the development of a model for the characterization of the two unique, fixed priority, feasible schedules of a given task set $\boldsymbol{\tau}$. The second is the the development of linear time procedures to determine on-line the idle capacity, precisely, on both extreme schedules. The third is a formal proof that when in non-idling, fixed priority periodic schedules we apply idling dispatching policies after a certain point in time t, we manage to relocate the maximum possible idle capacity

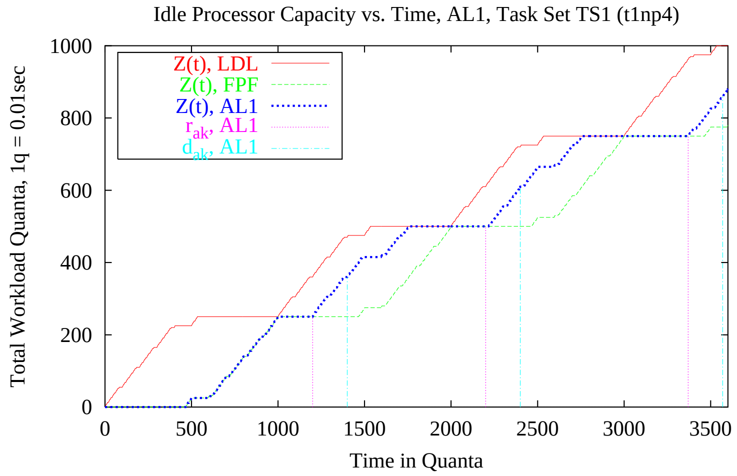


Figure 11: The Idle Processor Capacity $Z(t)$ for Task Set TS1 with Schedule Switching

within any interval starting at t.

This work considers the two extreme schedules for the periodic tasks in the system, the non-idling FPF and the idling LDL. The FPF reflects the schedule of the periodic tasks when they execute as soon as they are released and are dispatched by the highest-priority next (HPN) rule. The LDL reflects the latest points in time the periodic tasks can be assigned to the processor and still meet their deadlines. We provide a workload characterization of both schedules in terms of deterministic processes of time. The formal definition of the schedules gives rise to methods capable to compute on-line the time the system is busy serving periodic tasks in any arbitrary interval $[t_1, t_2)$. We have developed the Workload-Matrix scheme which calculates, precisely, the total time the system is busy within $[t_1, t_2)$ and performs this in time $\Theta(n)$. The determination of total busy period time within arbitrary intervals has been shown to be a computationally intensive task. Previous state-of-the-art methods require pseudo-polynomial time in the periodic task parameters to decide firm task feasibility on-line. To the best of our knowledge this is the first method reported in the literature which can accomplish this in linear time. This method is used directly in deciding the feasibility of incoming firm aperiodic tasks.

The Workload Dominance Theorem (1) shows that under FPF the system services the periodic jobs "faster" than under the LDL schedule. The practical effect of this is that FPF creates long busy periods clustered around the job release times. The system operating in under-load conditions dispatches the periodic jobs under the non-idling FPF schedule. The scheduler uses the workload-matrix method to compute precisely the total idle time in any given interval, and thus, determines the feasibility of incoming firm tasks in linear time. However, many firm tasks may have to be rejected. Under FPF, or any non-idling policy in general, firm tasks may have to wait for long times until the system serves the periodic jobs waiting in the ready queue and then provide

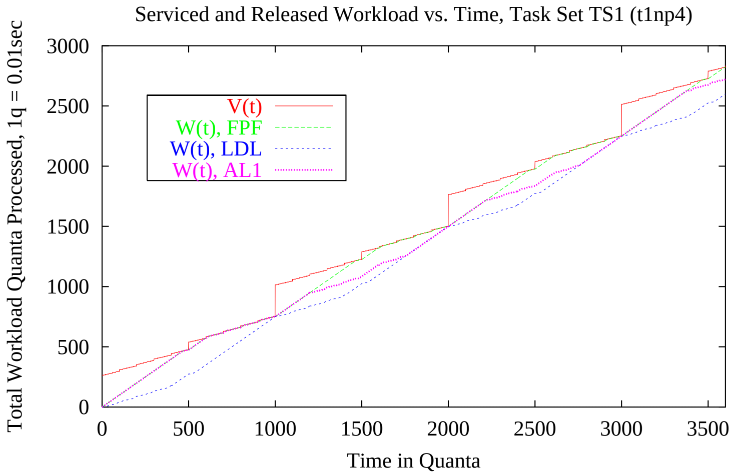


Figure 12: Serviced Workload $W(t)$ Processes for Task Set TS1 with Schedule Switching

service to them. As soon as a firm task cannot be guaranteed by the idle capacity in the current segment of the FPF schedule, by utilizing the results of Workload Dominance and Schedule Switch (2) Theorems, the scheduler determines the idle capacity had the system switched at that time into the idling LDL schedule. The Schedule Switch Theorem guarantees that the idle capacity can only increase at the interval following the switch point, and thus, the probability of the firm task becoming schedulable increases, as well. The LDL is the latest possible time periodic jobs can be dispatched, and thus, there is no other feasible schedule with this property and still being able guarantee the execution of the periodic jobs. Therefore, the decision to switch from a non-idling into an idling schedule maximizes the amount of idle capacity in the intervals starting at the switch point. The system may return back to the FPF schedule at the beginning of the first idle period following the completion of admitted firm tasks.

The Workload-Matrix method is capable to determine the idle capacity precisely, under the FPF, the LDL and the schedule following switchings from FPF to LDL and vice-versa, in linear time. In the absence of pending firm aperiodic tasks, the feasibility testing has a one-time linear complexity cost. In this work we do not provide details of the feasibility testing in the presence of several pending aperiodic tasks. In [43, 34] we show how we could integrate our method in firm aperiodic task scheduling and provide FIFO or EDF service to the firm tasks. The overall scheduling cost under FIFO is still $\Theta(n)$ and under EDF is $\Theta(n+k)$, where k is the number of pending aperiodic tasks.

Previous on-line feasibility methods required inordinately high computation time, rendering the methods very hard to scale. The value of high overhead admission control procedures is dubious, since they waste valuable processor time deciding instead of servicing dynamic requests. The approximate methods [15], [17], [44] which attempted to circumvent the heavy taxing of resources, still require time in the order of $\Theta(n^2)$. Performance

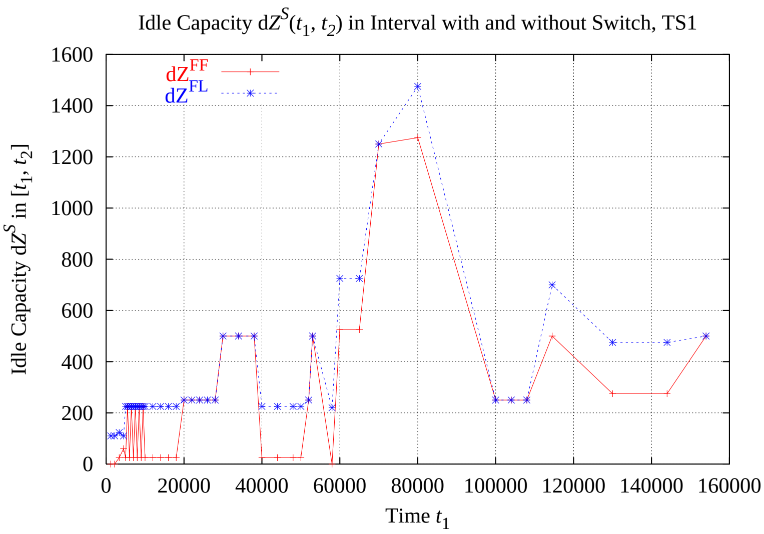


Figure 13: Idle Processor Capacity Difference for Task Set TS1 with dZ^{FL} and without dZ^{FF} Schedule Switching at t_1

studies [19], [44] also showed that in several circumstances they obtain very pessimistic lower bounds on the idle capacity, forcing many, otherwise schedulable, firm aperiodic tasks to be rejected.

The linear time complexity and extremely low actual computation overhead of the idle capacity management algorithms make them ideal for complex and highly dynamic environments. For instance, an ATM switch requires 2.6μ-seconds to transmit a single cell, at 155 Mbits/sec. Our idle capacity method, compiled with the regular compiler optimizations, executing on a common workstation without any hardware aid, requires 2.09μ-seconds when there up to 11 periodic tasks in the schedule. That is, the method can handle ATM link hard real-time scheduling at the cell level, as is. We expect that with carefully optimized code and/or the presence of hardware assist the method can handle schedules with large numbers of connections and at higher speeds.

We are currently incorporating the feasibility testing algorithms into a scheduler which is built upon POSIX-compliant systems. We are developing an approximate feasibility testing technique that need not store all $O(N)$ entries of the Workload-Matrix. This technique trades off accuracy for matrix storage. In the future we plan to develop schemes providing stochastic guarantees and systems characterized by stochastic workload parameters.

References

[1] R. I. Davis, K. W. Tindell, and A. Burns, "Scheduling slack-time in fixed priority preemptive systems," in *Proc. 14th IEEE Real Time Systems Symp.*, Raleigh-Durham, NC, Dec. 1993, pp. 222–231.

[2] John P. Lehoczky and Sandra Ramos-Thuel, "Scheduling periodic and aperiodic tasks using the slack stealing algorithm," in *Advances in Real-Time Systems*, Sang H. Son, Ed., chapter 8, pp. 175–198. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1995.

[3] Houssine Chetto and Maryline Chetto, "Some results of the earliest deadline scheduling algorithm," *IEEE Trans. on Software Engineering*, vol. 15, no. 10, pp. 1261–1269, Oct. 1989.

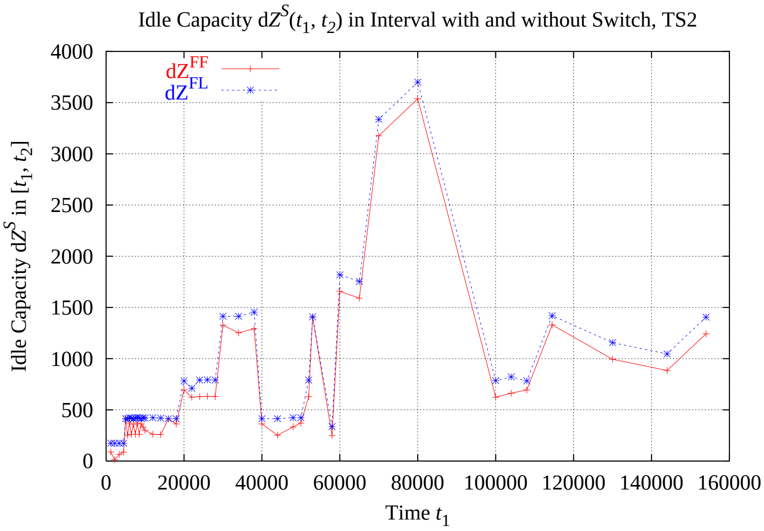


Figure 14: Idle Processor Capacity Difference for Task Set TS2 with $\mathrm{d}Z^{\mathrm{FL}}$ and without $\mathrm{d}Z^{\mathrm{FF}}$ Schedule Switching at t_1

[4] Maryline Silly, "The EDL server for scheduling periodic and soft aperiodic tasks with resource constraints," *Real-Time Systems*, vol. 17, no. 1, pp. 87–111, July 1999.

[5] R. I. Davis, "Approximate slack stealing algorithms for fixed priority pre-emptive systems," Tech. Rep. YCS–93–216, Comp. Science Dep't, Univ. of York, UK, 1993.

[6] R. I. Davis, K. W. Tindell, and A. Burns, "Mechanisms for enhancing the flexibility and utility of hard real-time systems," in *Proc. 15th IEEE Real Time Systems Symp.*, San Juan, Puerto Rico, Dec. 1994, pp. 12–21.

[7] Charlie McElhone, A. Burns, and R. Davis, "Hybrid algorithms for dynamic schedulability testing," in *Proc. 7th Euromicro Workshop on Real-Time Systems*, 14–16 June 1995, pp. 254–261.

[8] J. Lehoczky, Lui Sha, J. K. Strosnider, and Hide Tokuda, "Fixed priority scheduling theory for hard real-time systems," in *Foundations of Real-Time Computing: Scheduling and Resource Management*, André M. van Tilborg and Gary M. Koob, Eds., chapter 1, pp. 1–30. Kluwer Academic Publishers, Norwell, MA, 1991.

[9] Lui Sha, Mark H. Klein, and John B. Goodenough, "Rate monotonic analysis for real-time systems," in *Foundations of Real-Time Computing: Scheduling and Resource Management*, André M. van Tilborg and Gary M. Koob, Eds., chapter 5, pp. 129–155. Kluwer Academic Publishers, Norwell, MA, 1991.

[10] J. Lehoczky, L. Sha, and J. K. Strosnider, "Enhanced aperiodic responsiveness in hard real-time environments," in *Proc. 8th IEEE Real Time Systems Symp.*, Dec. 1987, pp. 261–270.

[11] B. Sprunt, J. Lehoczky, and L. Sha, "Exploiting unused periodic time for aperiodic service using the extended priority-exchange algorithm," in *Proc. 9th IEEE Real Time Systems Symp.*, Dec. 1988, pp. 251–258.

[12] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Trans. on Computers*, vol. 44, no. 1, pp. 73–91, Jan. 1995.

[13] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard real-time systems," *The Journal of Real-Time Systems*, vol. 1, pp. 27–60, 1989.

[14] Sandra Ramos-Thuel and John P. Lehoczky, "Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing," in *Proc. 15th IEEE Real Time Systems Symp.*, San Juan, Puerto Rico, Dec. 1994, pp. 22–33.

[15] R. I. Davis, "Scheduling slack time in fixed priority pre-emptive systems," Tech. Rep. YCS–93–217, Comp. Science Dep't, Univ. of York, UK, 1993.

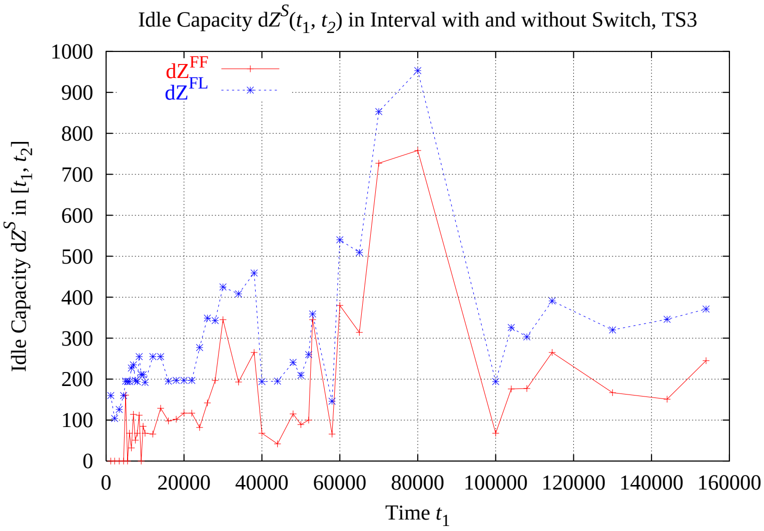


Figure 15: Idle Processor Capacity Difference for Task Set TS3 with $\mathrm{d}Z^{\mathrm{FL}}$ and without $\mathrm{d}Z^{\mathrm{FF}}$ Schedule Switching at t_1

[16] N. Audsley, A. Burns, R. I. Davis, and A. J. Wellings, "Integrating best effort and fixed-priority scheduling," in *Proc. of 1994 Workshop on Real-Time Programming*, June 1994, pp. 45–55.

[17] N. Audsley, R. I. Davis, A. Burns, and A. J. Wellings, "Appropriate mechanisms for the support of optional processign in hard real-time systems," in *Proc. 11th IEEE Real-Time Operating Systems and Software*, Seattle, WA, May 18–19 1994, pp. 23–27.

[18] Pamm Binns, "Incremental rate-monotonic scheduling for improved control system performance," in *Proc. Third IEEE Real-time Technology and Applications Symposium*, Montreal, Canada, June 9–11 1997, pp. 80–90.

[19] Charlie McElhone, "Adapting and evaluating algorithms for dynammic schedulability testing," Tech. Rep. YCS–94–225, Department of Computer Science, University of York, UK, 1994.

[20] Karsten Schwan and Hongyi Zhou, "Dynamic scheduling of hard real-time tasks and real-time threads," *IEEE Trans. on Software Engineering*, vol. 18, no. 8, pp. 736–748, Aug. 1992.

[21] Maryline Silly, Houssine Chetto, and Nadia Elyounsi, "An optimal algorithm for guaranteeing sporadic tasks in hard real-time systems," in *Second IEEE Symposium on Parallel and Distributed Processing*, 9–13 Dec. 1990, pp. 578–585.

[22] Housine Chetto, Maryline Silly, and T. Bouchentouf, "Dynamic scheduling of real-time tasks under precedence constraints," *The Journal of Real-Time Systems*, vol. 2, pp. 181–194, 1990.

[23] Too-Seng Tia, "Utilizing slack time for aperiodic and sporadic requests scheduling in real-time systems," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Urbana, IL, Apr. 1995, Available as Technical Report UIUCDCS-R-95-1906.

[24] Ismail Ripoll, Ana García-Formes, and Alfons Crespo, "Optimal aperiodic scheduling for dynamic-priority preemptive systems," *Proc. Third Int'l Workshop in Real-Time Computing Systems and Applications*, pp. 294–300, 1996.

[25] Ismail Ripoll, Alfons Crespo, and Ana García-Formes, "An optimal algorithm for scheduling soft aperiodic tasks in dynamic-priority preemptive systems," *IEEE Trans. on Software Engineering*, vol. 23, no. 6, pp. 388–400, June 1997.

[26] Giorgio C. Buttazzo and Fabrizio Sensini, "Optimal deadline assignment for scheduling soft aperiodic tasks in hard real-time environments," in *Proc. of the Third IEEE Int'l Conf. on Engineering of Complex Computing Systems*, Como, Italy, Dec. 1997, pp. 39–48.

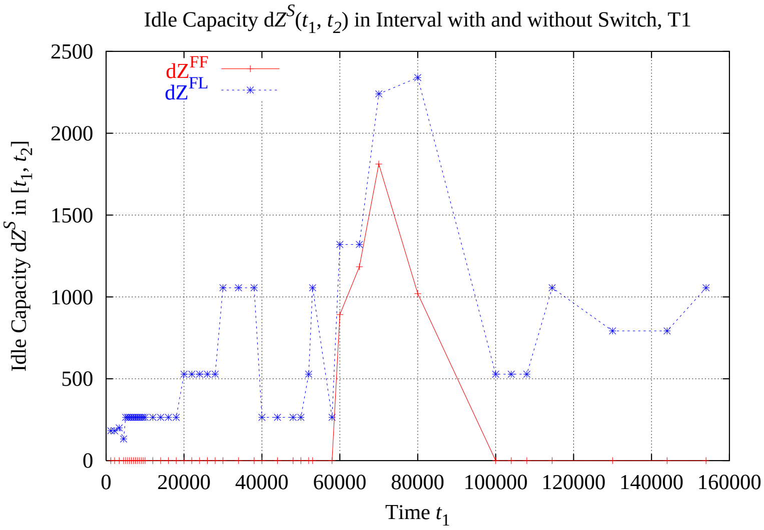


Figure 16: Idle Processor Capacity Difference for Task Set T1 with $\mathrm{d}Z^{\mathrm{FL}}$ and without $\mathrm{d}Z^{\mathrm{FF}}$ Schedule Switching at t_1

[27] Giorgio C. Buttazzo and Marco Caccamo, "Minimizing aperiodic response times in a firm real-time environment," *IEEE Trans. on Software Engineering*, vol. 25, no. 1, pp. 22–32, Jan./Feb. 1999.

[28] Giorgio C. Buttazzo and J. A. Stankovic, "RED: Robust earliest deadline scheduling," Tech. Rep. UM–CS–1993–025, Comp. Science Dep't, Univ. of Massachusetts, Amherst, July 1993.

[29] Giorgio C. Buttazzo and J. A. Stankovic, "Adding robustness in dynamic preemptive scheduling," in *Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems*, D. S. Fussell and M. Malek, Eds. Kluwer Academic Publishers, Boston, MA, 1995.

[30] C. M. Krishna and Kang G. Shin, *Real-Time Systems*, Mc Graw-Hill, New-York, NY, 1997.

[31] C. Liu and J. Layland, "Scheduling algorithms for multi-programming in a hard real-time environment," *Journal of the ACM*, vol. 20, pp. 46–61, 1973.

[32] Michael E. Thomadakis, "Algorithms and techniques for joint task scheduling in open computing systems," Ph.D. dissertation, Texas A&M University, College Station, TX 77843-3112, Dec. 1999.

[33] Joseph Y.-T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic real-time tasks," *Performance Evaluation*, vol. 2, pp. 237–250, 1982.

[34] Michael E. Thomadakis and Jyh-Charn S. Liu, "On the efficient scheduling of non-periodic tasks in hard real-time systems," Tech. Rep. CS-TR-99-012, Dept. of Computer Science, Texas A&M University, College Station, TX, May 10 1999.

[35] AT&T, *System V Interface Definition, 3rd Edition*, Addison-Wesley, Reading, MA, 1991.

[36] AT&T, UNIX *System V Release 4, Internals Student Guide*, Addison-Wesley, Reading, MA, 1990.

[37] Sandeep Khanna, Michael Sebree, and John Zolnowski, "Real-time scheduling in SunOS 5.0," in *Proc. of the Winter 1992 USENIX Conf.*, San Francisco, CA, 1992, pp. 375–390.

[38] IEEE Portable Applications Standards Committee, *IEEE Std 1003.1b-1993, Real Time Extensions*, IEEE, 1993.

[39] IEEE Portable Applications Standards Committee, *IEEE Std 1003.1c-1995, Threads Extensions*, IEEE, 1995.

[40] Silicon Graphics, "Topics in IRIX programming," Silicon Graphics Inc., Mountain View, CA, 1996.

[41] SunSoft, *SunOS Reference Manual, Section 2*, Sun Microsystems Inc., Mountain View, CA, 1994.

[42] U. Vahalia, UNIX *Internals: The New Frontiers*, Prentice-Hall, Upper Saddle River, NJ, 1996.

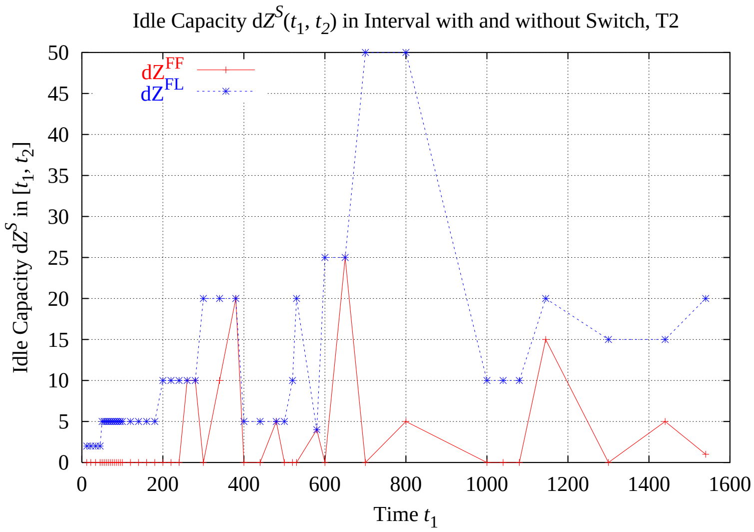


Figure 17: Idle Processor Capacity Difference for Task Set T2 with dZ^{FL} and without dZ^{FF} Schedule Switching at t_1

[43] Michael E. Thomadakis and Jyh-Charn S. Liu, "On the efficient scheduling of non-periodic tasks in hard real-time systems," in *Proc. 20th IEEE Real Time Systems Symp.*, Phoenix, AZ, Dec. 1–3 1999.

[44] Sandra Ramos-Thuel, "Enhancing fault tolerance of real-time systems through time redundancy," Ph.D. dissertation, Dep't of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, May 1993.

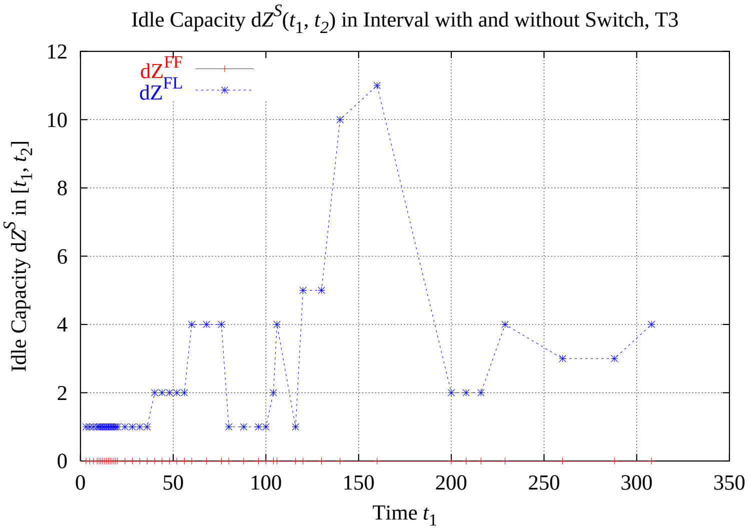


Figure 18: Idle Processor Capacity Difference for Task Set T3 with dZ^{FL} and without dZ^{FF} Schedule Switching at t_1

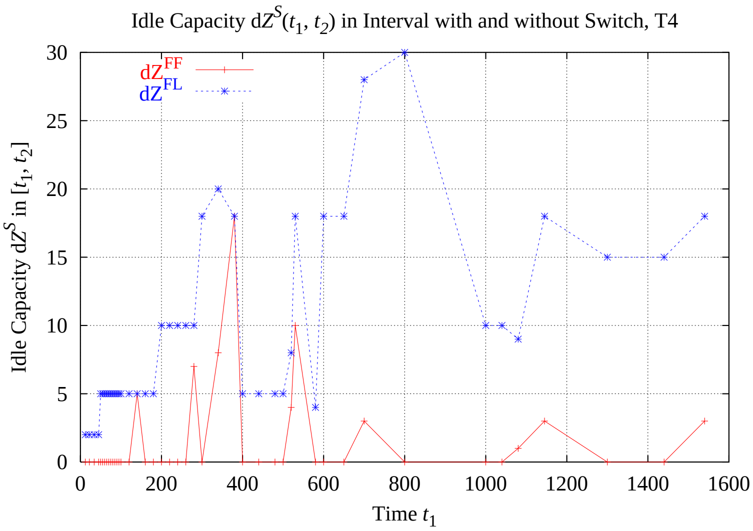


Figure 19: Idle Processor Capacity Difference for Task Set T4 with dZ^{FL} and without dZ^{FF} Schedule Switching at t_1

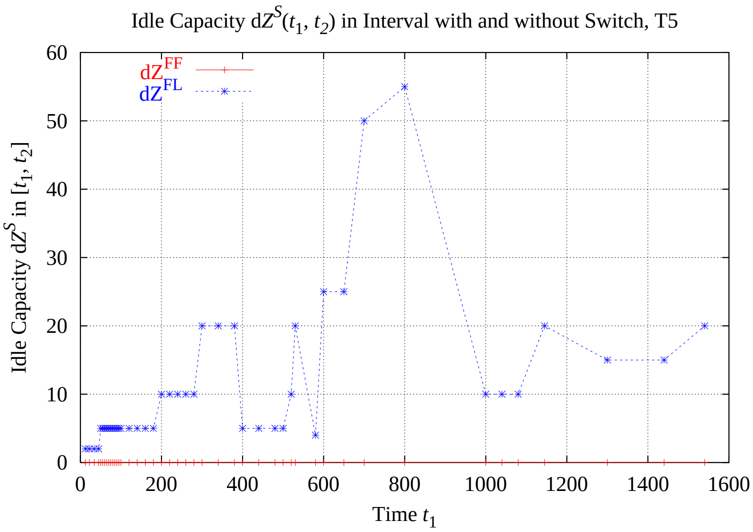


Figure 20: Idle Processor Capacity Difference for Task Set T4 with dZ^{FL} and without dZ^{FF} Schedule Switching at t_1

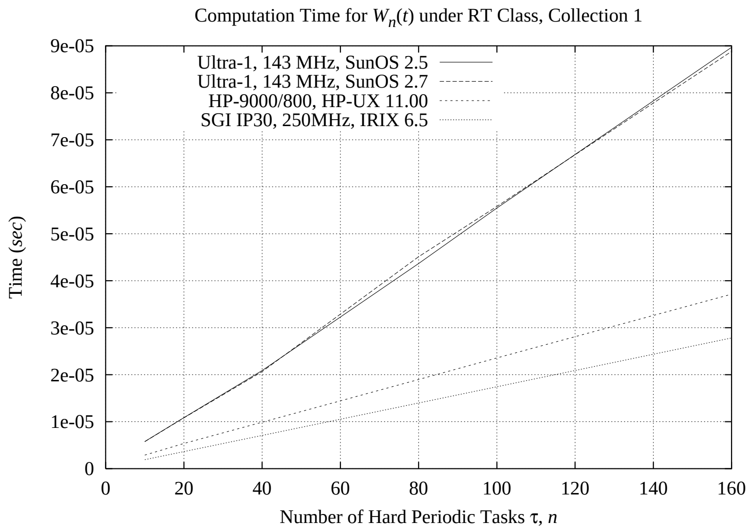


Figure 21: Computation Time of $W_n(t)$ for Collection 1 of Periodic Sets, under the Real-Time Scheduling Class in UNIX Platforms

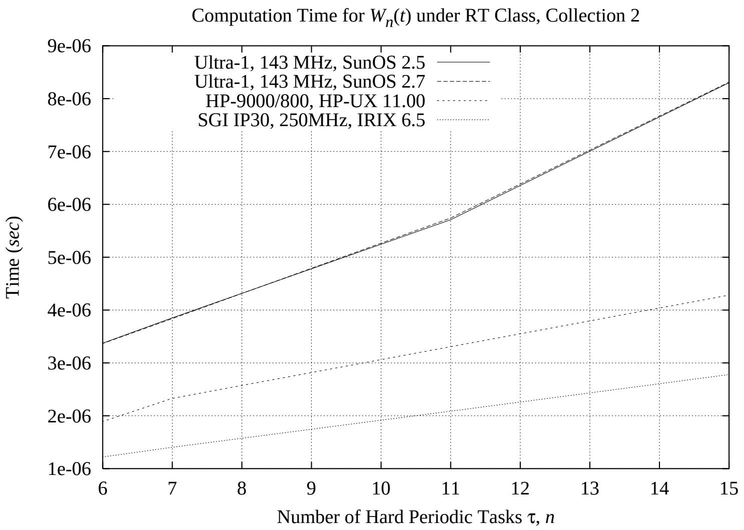


Figure 22: Computation Time of $W_n(t)$ for Collection 2 of Periodic Sets, under the Real-Time Scheduling Class in UNIX Platforms

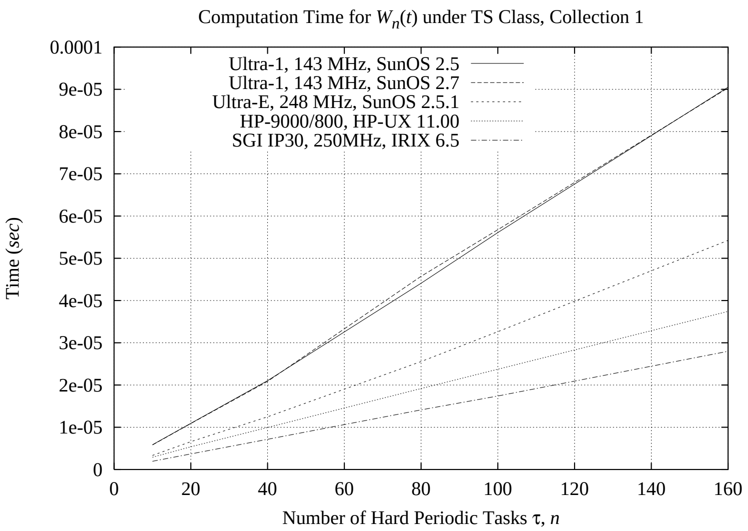


Figure 23: Computation Time of $W_n(t)$ for Collection 1, under the Time-Sharing Class in UNIX Platforms

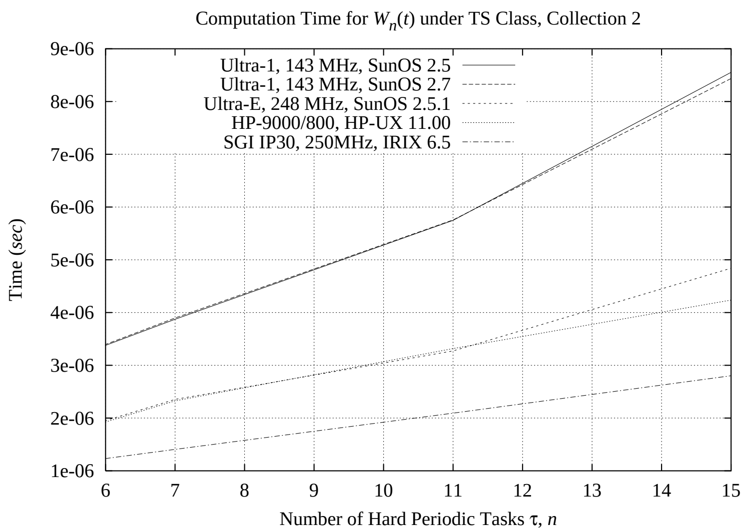


Figure 24: Computation Time of $W_n(t)$ for Collection 2, under the Time-Sharing Class in UNIX Platforms

APPENDIX

A Notation

Table 4: Brief Definition of Relevant Quantities

Notation	Definition
$\boldsymbol{\tau}$	Task system with n tasks $\tau_i(T_i, C_i, D_i, r_i)$
τ_{ij}	Job τ_{ij}, the jth instance of task τ_i
$\boldsymbol{J}_a$	Aperiodic job sequences
$v_i(t)$	Total workload for task τ_i in $[0, t]$
$V_i(t)$	Aggregate released workload for tasks $\tau_k,\ k = 1, 2, \ldots, i$ in $[0, t]$
$\mathcal{S}(t)$	Schedule Mapping $\mathcal{S} : \mathbb{R} \longrightarrow \{0, 1, \ldots, n\}$
$s_i(t)$	Indicator function for task τ_i in schedule $\mathcal{S}(t)$
$w_i(t)$	Total serviced workload for task τ_i in $[0, t)$
$W_i(t)$	Aggregate serviced workload for tasks $\tau_k,\ k = 1, 2, \ldots, i$ in $[0, t)$
$Z_i(t)$	Idle processor capacity in $[0, t)$, after tasks $\tau_k,\ k = 1, 2, \ldots, i$ have been serviced
s_{ij}, f_{ij}	Start and finish time for τ_{ij}, with $r_{ij} \leq s_{ij}$ and $f_{ij} \leq d_{ij}$
R_{ij}	Response time of τ_{ij}, with $R_{ij} := f_{ij} - r_{ij}$
$n_i(t)$	$\left\lfloor \frac{t - r_i}{T_i} \right\rfloor + 1$, current period number of τ_i at time t
$m_i(t)$	$(t - r_i) \pmod{T_i}$, "translated" remainder
$[x]^+$	$\max\{x, 0\}$.

B Experimental Taks Sets

In this part of the paper we give details on the periodic task sets we have used in the examples and for the experiments. Each table shows the number of periodic tasks n, the hyper-period H, the utilization ρ and the discretization unit q. Sets TS1, TS2 and TS3 are synthetic and T1, T2, T3, T4, T5, T6, T7 and T71 are actual real-time task sets.

Table 5: Synthetic Task Set TS1

n	H	ρ	q
5	$10sec$	0.75	0.01
C_i	T_i	D_i	r_i
0.020	0.100	0.100	0.000
0.030	0.200	0.200	0.000
0.100	1.000	1.000	0.000
0.500	5.000	5.000	0.000
2.000	10.000	10.000	0.000

Table 6: Synthetic Task Set TS2

n	H	ρ	q
5	$77sec$	0.356494	0.01
C_i	T_i	D_i	r_i
0.01	0.1	0.1	0.0
0.03	0.7	0.7	0.0
0.09	2.2	2.2	0.0
0.40	5.5	5.5	0.0
1.10	11.0	11.0	0.0

Table 7: Synthetic Task Set TS3

n	H	ρ	q
10	$231sec$	0.860823	0.01
C_i	T_i	D_i	r_i
0.01	0.1	0.1	0.0
0.01	0.2	0.2	0.0
0.02	0.7	0.7	0.0
0.03	0.7	0.7	0.0
0.12	2.1	2.0	0.0
0.14	2.1	2.0	0.0
0.22	4.2	4.1	0.0
1.22	4.2	4.1	0.0
0.40	5.5	5.5	0.0
1.10	11.0	11.0	0.0

Table 8: Inertial Navigation Task Set T1

n	H	ρ	q
6	$5000sec$	0.884040	0.01
C_i	T_i	D_i	r_i
1.18	2.500	2.500	0.000
4.28	40.000	40.000	0.000
10.28	62.500	62.500	0.000
20.28	1000.000	1000.000	0.000
100.28	1000.000	1000.000	0.000
25.00	1250.000	1250.000	0.000

Table 9: Submarine Sonar Task Set T2

n	H	ρ	q
6	$60000sec$	0.572500	1
C_i	T_i	D_i	r_i
50.0	100.0	100.0	0.0
9.0	500.0	500.0	0.0
41.0	1000.0	1000.0	0.0
5.0	2500.0	2500.0	0.0
33.0	3000.0	3000.0	0.0
2.0	4000.0	4000.0	0.0

Table 10: Surface Ship Radar Task Set T3

n	H	ρ	q
11	$20000sec$	0.837500	5
C_i	T_i	D_i	r_i
25.0	2000.0	2000.0	0.0
10.0	4000.0	4000.0	0.0
25.0	5000.0	5000.0	0.0
35.0	5000.0	5000.0	0.0
400.0	5000.0	5000.0	0.0
25.0	10000.0	10000.0	0.0
30.0	10000.0	10000.0	0.0
50.0	10000.0	10000.0	0.0
300.0	10000.0	10000.0	0.0
1125.0	10000.0	10000.0	0.0
5775.0	10000.0	10000.0	0.0

Table 11: Avionics System Task Set T4

n	H	ρ	q
11	$118000sec$	0.830093	1
C_i	T_i	D_i	r_i
2.0	25.0	25.0	0.0
5.0	25.0	25.0	0.0
1.0	40.0	40.0	0.0
3.0	50.0	50.0	0.0
5.0	50.0	50.0	0.0
8.0	59.0	59.0	0.0
9.0	80.0	80.0	0.0
2.0	80.0	80.0	0.0
5.0	100.0	100.0	0.0
3.0	200.0	200.0	0.0
1.0	200.0	200.0	0.0
1.0	200.0	200.0	0.0
3.0	200.0	200.0	0.0
1.0	1000.0	1000.0	0.0
1.0	1000.0	1000.0	0.0

Table 12: Multimedia System Task Set T5

n	H	ρ	q
11	$300000sec$	0.684450	1
C_i	T_i	D_i	r_i
28.0	125.0	125.0	0.0
19.0	250.0	250.0	0.0
1175.0	6000.0	6000.0	0.0
9.0	12000.0	12000.0	0.0
1880.0	25000.0	25000.0	0.0
1880.0	30000.0	30000.0	0.0
5000.0	100000.0	100000.0	0.0

Table 13: The Satellite Control System Task Set T6, without the Sporadic Tasks and with All $r_i = 0$

n	H	ρ	q
10	$1sec$	0.579220	0.000010
C_i	T_i	D_i	r_i
0.00028	0.050	0.009	0.0
0.00176	0.010	0.010	0.0
0.00213	0.200	0.014	0.0
0.00143	0.200	0.017	0.0
0.00143	0.200	0.022	0.0
0.00143	0.100	0.024	0.0
0.00821	0.100	0.050	0.0
0.05284	0.200	0.200	0.0
0.00516	1.000	0.400	0.0
0.00691	1.000	0.900	0.0

Table 14: The Satellite Control System Task Set T7, without the Sporadic Tasks

n	H	ρ	q
10	$1sec$	0.579220	0.000010
C_i	T_i	D_i	r_i
0.00028	0.050	0.009	0.00
0.00176	0.010	0.010	0.00
0.00213	0.200	0.014	0.05
0.00143	0.200	0.017	0.05
0.00143	0.200	0.022	0.00
0.00143	0.100	0.024	0.00
0.00821	0.100	0.050	0.05
0.05284	0.200	0.200	0.05
0.00516	1.000	0.400	0.20
0.00691	1.000	0.900	0.20

Table 15: The Satellite Control System Task Set T71, with the Sporadic Tasks

n	H	ρ	q
13	$3sec$	0.858560	0.000010
C_i	T_i	D_i	r_i
0.00018	0.00096	0.00063	0.00
0.00028	0.050	0.009	0.00
0.00176	0.010	0.010	0.00
0.00213	0.200	0.014	0.05
0.00143	0.200	0.017	0.05
0.00143	0.200	0.022	0.00
0.00143	0.100	0.024	0.00
0.00319	0.0625	0.030	0.00
0.00821	0.100	0.050	0.05
0.00408	0.100	0.050	0.05
0.05284	0.200	0.200	0.05
0.00516	1.000	0.400	0.20
0.00691	1.000	0.900	0.20