Pipelined Bursts in Real-Time Scheduling*

Donald W. Gillies†‡

November 1, 1998

Abstract

We consider the problem of performance guarantees for a hard real-time file system. We adopt an idea called *constant-throughput resource management* to manage the disk subsystem in a predictable way. This type of management leads us to the idea of static-priority periodic scheduling of bursty requests; a bursty request increases the task period, execution time, and deadline by a common multiplier of one or more. We derive critical instant tests and utilization formulae for this new type of scheduling workload. To achieve better performance, we pipeline the requests and generalize important earlier work on pipelined scheduling. We show that at least 50% of a burst-managed resource may be used for real-time requests, by setting the pipeline factor equal to the maximum burst factor. For the file system we conclude that at least 50% of the disk's time may be dedicated to data transfer, with modest buffering and start-up latencies. The other 50% of the disk's time could be dedicated to seeking. Our design allows the system administrator to choose this split (50-50, 90-10, or whatever) at disk format time, and trades fragmentation for improved performance. We also discuss other applications of bursty scheduling, i.e. managing jitter in real-time networks, performing MPEG transfer, sizing real-time caches, and managing transient overloads through explicit scheduling.

*This research was partially supported by a grant from NSERC

†Department of Electrical Engineering, University of British Columbia, Vancouver, British Columbia, Canada V6T 1Z4. email: gillies@cs.ubc.ca

‡Planetwide Software, Inc., 4448 Cather Ct, San Diego CA 92122

1 Introduction

Traditional real-time periodic scheduling is well-suited for low-variance execution requests when there is little or no preemption cost. For example, a sensor may be served by repeated periodic attention from the CPU. Dedicated byte-wide serial channels may also be serviced in this way. Such system resources require little or no start-up and shut-down time to access the device, and there is typically a steady and repeating workload stream for each device.

Many next-generation real-time applications cannot easily adapt to this workload model. In particular, some devices may have erratic preemption times (disks, packet networks) because the device can be left in a still-busy state by the previous user or the previous request. For example, the disk head may be far away from the next sector request, or a large FTP packet may still be under transmission when it is time to send a low-latency voice packet. Contemporary static-priority scheduling policies produce poor guarantees in these devices. Previous techniques to model preemption costs assume small fixed preemption costs [13]. Unlike this previous work, we reconfigure the application structure to match the worst-case preemption cost to the amount of work done for each device operation, to achieve *constant-throughput device management.*

The application that motivated this work originally was the design of a hard real-time file service. The problem arises in disk seeking, produced either by fragmentation or by context switching between different streams of periodic read or write requests. If the disk arm and file layout on the disk are not managed carefully, up to 98% of the disk throughput could be squandered by fragmentation and disk seeking [2]. By formatting the disk in a special way to reduce the preemption cost (in relation to the size of a data transfer), we will show how a system can sustain 50-80% of the disk's maximum transfer rate can with modest buffering.

The scheduling models of this paper are similar to the Liu-Layland models [6] with the following extensions. In a Liu-Layland task system, the task period is always a multiple of the task execution request (the multiple is the inverse of the task *utilization*). In a Liu-Layland model, the execution request never varies. In our model, the execution request may inflate by a variable factor in the range $[1, \beta]$. The value β would be $1.5 - 3$ in a typical real-time file system. In a given period, the inflation β' is called the *burst factor*. The value of β' is unpredictable and varies in every new period. When the task execution request inflates by β', the task period inflates by β' at the same time. Thus, over long periods of time the task system utilization is constant. We note also that Liu and Layland considered CPU scheduling, whereas we consider disk transfer (seek and read/write) scheduling. Our engineering analysis can be applied to other problems where preemption costs are variable, if the application can be structured to match preemption costs to the amount of useful work done in the device transaction.

We choose to study static-priority scheduling because it is simple and efficient, and it behaves predictably under transient overload. This type of scheduling achieves 85% CPU utilization on average [8], and nearly 100% CPU utilization when task deadlines exceed the periods [8]. In some practical applications such as the Iridium Satellite Payload Software, the workload may achieve 99+% utilization [4] with modest restructuring of the application. When applied to workloads with widely varying priods, 95% utilization can often be achieved, and a file system load with widely varying transfer rates (video, audio, animation) should achieve this performance. Nearly every CPU that supports real-time programming supports static-priority scheduling in its interrupt hardware. Static-priority scheduling is gaining wide acceptance; it has recently begun DoD trials for certification as an accepted technology for U.S. military projects.

1.1 Disk Arm Scheduling and Buffer Management

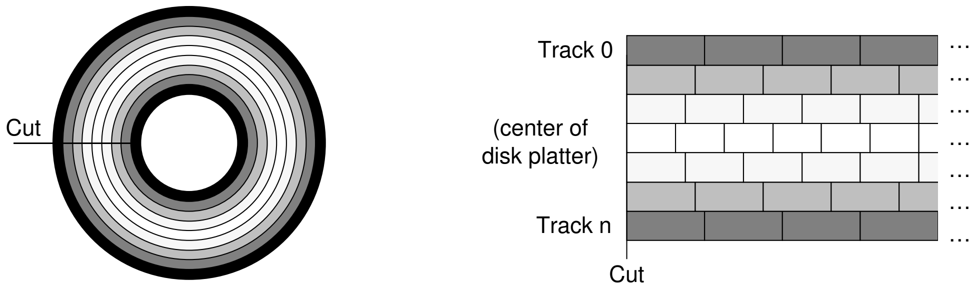
An important challenge in real-time file system design is in disk arm control. There are many disk scheduling algorithms in the literature [1] [3] [12], however, few designs support (a) continual real-time write capability, with no need for offline defragmentation, (b) arbitrary peak playback speed, up to the maximum transfer rate of the disk, set at format time, and (c) complete asynchrony, i.e. clients can read or write, stop and go, at any rate up to a guaranteed rate, chosen by the client when the file was opened, without troubling the file system software. Our design supports all these features at once, and provides precise analytic guarantees.

We assume that all disk transfers are non-preemptive due to the mechanical limits of disk hardware. Our fundamental premise is that every real-time disk access should produce data at a rate of R bytes per second (or more), no matter where the disk arm starts. To do this, we format the disk in a special way, so that e.g. some requests may be for R bytes and the worst-case seek+read takes 1 second, and other requests may be for $3R$ bytes, and the worst-case seek+read requires 3 seconds. In all cases, the worst-case seek and read will produce R bytes per second.

To format the disk we divide it into logical blocks, each one composed of many physical blocks. Unlike most file systems, our file system has a *variable sized logical blocks*. Logical blocks at the outer edge of the disk platter may at times require a long seek, because the disk head may start at the spindle. To achieve R bytes per second, the logical block there will have to be larger, based on the worst-case seek starting at the spindle. Logical blocks in the center tracks of the platter may be smaller, because the worst-case seek distance is only half the radius of the platter. This logical block might be half the size of a block at the edge of the disk. Figure 1(a) depicts the appearance of the disk. In both figures, larger logical blocks have a darker shading. Figure 1(b) depicts the logical block sizes in more detail. This figure is derived by cutting the disk surface and stretching it topologically into a rectangle. When formatting is completed, the burst size may be calculated as the maximum seek and transfer divided by the minimum seek and transfer. Recall that each disk transaction produces data at R bytes per second.

A client calls $open(p, r)$, where p is a path name, and r is a ceiling on the expected transfer rate (in bytes per second). The file system prepares to issue seeks and transfers. The periodic task request will be one seek and one transfer. The task period will be R/r times larger than the shortest seek and transfer (measured when the disk was originally formatted).

Let X be the time for the smallest seek and transfer created at disk format time. Let M be the time for the largest seek and transfer created at disk format time. Because of the formatting



(a) *Disk layout* (b) *Disk layout in fine detail*

Fig. 1. *Disk block layout for a bursty-scheduling real-time file system.*

algorithm, $M = \beta * X$. The client's execution time will be X, and the period will be R/r seconds. The client is given a memory (RAM) buffer of size $M+1$. Thus, the period, execution time, and task system-wide burst factor are konwn, and the schedulability tests in this paper may be used to determine if the aggregate workload will meet all the periodic deadlines. In a pipelined file system, with δ pipeline periods, that overlapping task period would be $\delta R/r$, and the client memory would be of size $\delta M + 1$.

1.2 Related Problems

In this section, we review other examples of burst processing in a computer system.

Cache Management. Modern cache memories are managed using burst processing principles. When a memory reference is not in the cache, extra words are fetched to make up for the time lost in the cache miss of the first word. The burst factor for this device is the ratio between the hit time and miss time. An optimized cache will have all useful fetches proceed at the same rate, where the useful fetch size is the typical portion of the cache line that is likely to be used.

Video Transmission. Consider a video router than is outputting MPEG frame bursts from several clients. Some MPEG frames of one stream may be larger than others in that stream. The time ratio between the largest and smallest MPEG frame is a burst factor. If different streams use different static frame priorities, the router's ability to carry real-time traffic as a function of router buffer size may be calculated using the results of this paper.

Transient Overloads. A transient overload occurs when one or more tasks exceed their typical worst-case execution requirements. With scheduler support, an overload may cause the offending task to wait longer for its next request. In this type of system, the results of this paper may be used to determine if a task could starve during overload, and also, what degree of pipelining is required to prevent task starvation during transient overload.

Jitter in a Real-time Network. Over a long transmission path, a smooth periodic sequence of frames produced by a leaky bucket traffic shaping algorithm may come to resemble a bursty periodic workload. The burst factor will depend on the maximum jitter of each frame stream. When several frames streams are superimposed, according to static service classes, and there is limited buffering, the results of this paper will determine the worst-case frame loss among the lower-priority service classes.

1.3 Outline

The rest of this paper is organized as follows. Section 2 introduces notation for the periodic scheduling problem and describes previous work in periodic scheduling. Section 3 considers the bursty scheduling problem and derives a utilization test for this problem. Section 4 extends the bursty scheduling problem to include overlapping pipelined requests, assuming that the deadline of a task is an integer multiple of the period. This section again derives a utilization test for this problem. Section 5 considers the problem of non-preemptive scheduling with periodic bursty pipelined tasks. Most of the resources in a computer system cannot be easily preempted. This section extends existing non-preemptive utilization tests to include bursts and pipelining. Section 6 summarizes the paper and proposes future work.

2 Notation and Previous Work

The periodic scheduling problem is defined as follows. We are given a set of tasks $\{\tau_1, \tau_2, \cdots, \tau_n\}$. Each task τ_i is an independent unit of computation and consists of an infinite sequence of requests, $\tau_i = \{\tau_{i,1}, \tau_{i,2}, \cdots\}$. The *request period* of τ_i is T_i, the *deadline* of τ_i is D_i, and the *computation time* of τ_i is C_i. A set of tasks and associated parameters $\mathbf{T} = \{\tau_i, C_i, D_i, T_i\}$ is known as a *task system.* Initially, we assume that $D_i = T_i$, i.e. the deadline is equal to the period. Later, this restriction will be removed. The task system is periodic in the following sense: if τ_i makes a computation request at time 0, it asks to execute for C_i time units in the time interval $[0, T_i]$, and asks again for C_i time units in each of the intervals $[T_i, 2T_i]$, $[2T_i, 3T_i]$, and so forth. The periodic scheduling problem is thus to produce a time schedule so that each task executes exactly once in each period.

A task *misses its deadline* in the k'th period if it receives less than C_i units of execution in the interval $[kT_i, kT_i + D_i]$. In our task systems the first request of each task occurs at time 0, and the task system is said to be in *in phase.* In this paper we only consider task systems that are in phase.

This problem is often solved using a *static-priority* heuristic. Task names are assigned according to importance, task τ_1 has the highest importance or priority and task τ_n has the lowest priority. The scheduling heuristic works as follows. Whenever the processor is free, the algorithm selects the highest priority unfinished request among all outstanding requests. Whenever a new request period begins, the heuristic may reevaluate its notion of the highest priority request, possibly preempting an existing task.

This real-time scheduling problem and heuristic are widely studied. Results have been derived by Liu & Layland [6], Lehoczky [8], Lehoczky, Sha, and Strosnider [10] [9], Shih, Liu, and Liu [15], Tindell [17], and many others. In particular, Liu & Layland showed that the rate-monotonic priority assignment was optimal among all priority assignments, and that all deadlines would be met if the task system utilization $\sum(C_i/T_i) \leq ln\ 2$.

In this paper we extend the work of Liu & Layland [6] and Lehoczky [8] by considering a new variation of this problem. We assume in this paper that each task may choose a computation request $\beta' C_i$, and request period $\beta' T_i$, $1 \leq \beta' \leq \beta$, at its own discretion. The choice may change in every new period, and different tasks may make difference choices in every period. We call β the *maximum burst factor* for the task system. The quantity β' is the *period burst* for task τ_i.

3 Simple Bursty Workloads

In this section we introduce some theorems that describe the scheduling of bursty workloads. In the normal case when there are no bursts, a task repeatedly requests execution at the times 0, T_i, $2T_i$, etc. The *response time* of a task is the time span between the time of an execution request and the completion of execution (i.e. the total time that elapses before the task has executed for C_i time units). A *critical instant* for a task is the instant at which a task request for execution will have the largest response time [6]. Consider a set of tasks in priority order $\tau_1, \tau_2, \cdots \tau_n$, where task τ_1 has the highest priority and τ_n has the lowest priority. The theorems that follow apply to the priority assignment of any static-priority scheduling algorithm, and are similar to the theorems found in [6] and [8].

Lemma 3.1. *If each task τ_j finishes before time T_j for all j, a critical instant for task τ_i occurs when τ_i requests execution simultaneously with all higher priority tasks.*

Proof. Assume an instance of a critical instant schedule, where τ_i is released at some time t_1 and completes its execution at time t_2, where $t_2 - t_1$ is as large as possible. We will adjust the release times of $\tau_1 \cdots \tau_i$ without decreasing the response time of τ_i.

Let t_0 be the latest time instant before t_1 when the executing task has priority i or lower, or the processor is idle. Obviously if τ_i were released at time t_0 it would still finish its execution at time t_2, since only higher priority tasks execute in the time interval $[t_0, t_1]$. Therefore, we may set $t_1 \leftarrow t_0$, i.e. we may assume without loss of generality that no higher priority task is available for execution at the time τ_i is released. Note that it might not be possible to reduce t_1 if the response time of the previous request of task τ_i exceeded T_i. In this case the previous request of τ_i might contribute to the response time of the present request of τ_i, and reducing t_1 might reduce the response time of the present request of τ_i.

Let τ_j be any higher priority task other than τ_i. Let the next release of τ_j after $t_1 = t_0$ occur at time t_j. We assert that the request by τ_j whose period ends at t_j must have completed its execution request by time t_0 (since all tasks at priority level i or higher have no outstanding execution requests at time t_0). Observe that τ_i is delayed by an integer number of requests of τ_j. If we change the phase of τ_j by setting $t_j \leftarrow t_0$, then τ_j makes either zero or one more request before τ_i completes execution. Hence the response time of task τ_i is either unchanged or increased. This argument is repeated for each higher priority task τ_j, as needed.

Hence we may rearrange the phases of all higher priority tasks to coincide with the release time of τ_i. Therefore, a critical instant for τ_i occurs when all higher priority tasks are released at the same time as τ_i. $\square$

This lemma allows us to characterize a critical instant for a task τ_i:

LEMMA 3.2. *A critical instant for τ_i occurs if*

(1) All tasks request execution simultaneously (e.g. at time 0);

(2) τ_i finishes at time t;

(3) Each higher priority task τ_j makes n_j requests in the time interval $[0, t - \beta C_j]$.

(4) Request n_j of task τ_j is a maximum burst, i.e. $\beta' = \beta$, $1 \le j < i$.

Proof. From Lemma 3.1, we may assume that a critical instant occurs when all tasks make requests simultaneously. Consider any schedule of the tasks $\{\tau_1, \cdots \tau_i\}$ exhibiting a critical instant for task τ_i. If task τ_i completes at time t, every higher-priority execution request that occurs in the time interval $[0, t]$ completes before τ_i completes. In fact, τ_i completes at exactly time t if and only if the act of removing τ_i from the task system leaves exactly C_i idle time units free in the time interval $[0, t]$ with t as small as possible. We will show that any critical instant schedule can be transformed into a schedule that fits the theorem, with no decrease in the response time of τ_i.

For each task τ_j, $1 \le j < n$, we assert that the last request must be of size βC_j. If it is not, then increasing the last request must delay the finish time of τ_i since the processor is 100%

utilized in the time period $[0,t]$, and this would contradict the critical instant property of the schedule. Hence, there is a class of critical-instant schedules that have this form. □

To derive a utilization test for the bursty workload model it is necessary to find the number of requests in a worst-case workload. A worst-case workload has minimal task-system utilization, and leads to a schedule where one task misses its deadline. We assume without loss of generality that τ_n misses its deadline, because tasks of lower priority may not execute before τ_n misses its deadline, so these lower priority tasks may be omitted from the task system.

LEMMA 3.3. *For the rate-monotonic scheduling algorithm there exists a class of worst-case workloads wherein each task $\tau_1 \cdots \tau_{n-2}$ makes two execution requests before a deadline is missed. In this class of workloads, τ_{n-1} makes either one or two requests before a deadline is missed.*

Proof. Consider a worst-case bursty workload with a task τ_i, $i < n$, that makes $k > 2$ requests before a deadline is missed at time T_n. Let the k bursty requests by τ_i before time T_n be of size β_1, β_2, $\cdots$ β_k. Let τ_i' denote a new task with $T_i' = T_i \sum_{j=1}^{k-1} \beta_j$, and let $C_i' = C_i \sum_{j=1}^{k-1} \beta_j$. We replace τ_i by τ_i' in the task system. It is evident that the demand for execution by task τ_i' in the time interval $[0, T_i \sum_{j=1}^{k-1} \beta_i]$ is either unchanged or increased, and occurs earlier than the demand from τ_i. The second request of τ_i' occurs at time T_i' and its computation increases from $\beta_k C_i$ to $\beta_k C_i \sum_{j=1}^{k-1} \beta_j$, and hence, task τ_n still misses its deadline.

We may apply this modification to every task $\tau_1 \cdots \tau_{n-1}$ to show that there is a class of worst-case schedules where each task makes at most 2 requests. In the many situations we may increase the response time of τ_n further by modifying some T_i to make single request. For rate-monotonic scheduling we must have $T_i \leq T_n$ so we set $C_i' = T_n \frac{C_i}{T_i}$ and set $T_i' = T_n$. By Lemma 3.2, this last request will be a maximum burst. This modification will increase or leave unchanged the response time of τ_n iff $\beta T_n \frac{C_i}{T_i} \geq C_i(1+\beta)$.

If we modify two different tasks τ_{i_1}, τ_{i_2} to make a single request in the time period $[0, T_n]$ then it is easy to see that both tasks can be merged into single task, and hence there is a class of worst-case schedules where a unique task makes a single request in the time period $[0, T_n]$.

Since this single-request task has period T_n, we may assume without loss of generality that this task is named τ_{n-1}. $\square$

LEMMA 3.4. *There exists a family of worst-case workloads such that* $T_{i+1} \geq T_i + C_i$ *for* $1 \leq i < n-1$ *and either* $T_n = T_{n-1}$ *or* $T_n \geq T_{n-1} + C_{n-1}$.

Proof. First, we apply Lemma 3.3 to the task system, so that $\tau_1 \cdots \tau_{n-1}$ each makes at most two execution requests before τ_n misses its deadline. If task τ_{n-1} makes only one request, then $T_n = T_{n-1}$ by Lemma 3.3 and a portion of the lemma is established. We then consider every pair of tasks τ_i and τ_{i+1}. If $T_i = T_{i+1}$, we may delete τ_{i+1} from the task system, and set $C_i \leftarrow C_i + C_{i+1}$ without changing the response time of task τ_n. If $T_{i+1} = T_i + \delta$, $0 < \delta < \beta C_i$, it should be clear that task τ_{i+1} cannot possibly execute in the time interval $[T_i, T_i + \beta C_i]$, assuming rate-monotonic priorities are used and a burst by T_i in its second period according to Lemma 3.2. Therefore, we may set $T_{i+1} \leftarrow T_i + \beta C_i$, for every i where this relationship holds. All these modifications can only decrease the utilization of the task system, and will not affect the response time of τ_n. $\square$

THEOREM 3.1. *A bursty periodic task system with* $\beta \geq 1$ *is schedulable according to the rate-monotonic priority assignment if* $U \leq min\{\frac{1}{\beta}((n-1)((\frac{1+\beta}{\beta})^{1/(n-1)}-1)+1-\frac{1}{\beta}), \frac{n}{\beta}((1+\beta)^{1/n}-1)\}$

Proof. The problem of determining the utilization of a worst-case workload may be formulated as a linear program with a non-linear objective function. We want to minimize the task system utilization with a constraint that one task misses a deadline. The utilization of a task system is expressed in (1) below. Each task must have an execution time that is less than its period, which is represented by (2) below.

Case 1. *Assume that task* τ_{n-1} *makes only one request in* $[0, T_n]$. Then the results of Lemma 3.4 yield the set of formulae given in (3) and (4) below. We represent the fact that T_n misses its deadline as follows. We assume that τ_n makes a bursty execution request of size β' with $1 \leq \beta' \leq \beta$, and calculate the processor demand by all the higher priority tasks that run in the time period $[0, \beta' T_n]$. After applying Theorem 3.1, this may be expressed by (5) below assuming one request by T_{n-1}. In this program all the variables are positive.

$$
\begin{array}{lrrcll}
(1) & \text{Minimize} & U & = & \displaystyle\sum_{i=1}^{n} \frac{C_i}{T_i} & \\
(2) & \text{Subject to} & C_i & \leq & T_i, & 1 \leq i \leq n \\
(3) & & T_{i+1} & \geq & T_i + \beta C_i, & 1 \leq i < n-1 \\
(4) & & T_{n-1} & = & T_n & \\
(5) & & \displaystyle\sum_{i=1}^{n-2} (\beta+1) C_i + \beta C_{n-1} + \beta' C_n & \geq & \beta' T_n &
\end{array}
$$

By moving $\beta' C_n$ to the right hand side of (5), we see that for any solution with $\beta' > 1$, there is also a solution with $\beta' = 1$. Thus we may assume without loss of generality that $\beta' = 1$.

We claim there is no solution where (3) is a strict inequality. For if there is we may set $T_i \leftarrow T_{i+1} - \beta C_i$, which decreases U and all the other constraints are still satisfied. Hence in any solution, equality must hold in (3). By a similar argument, in any solution equality must hold in (5). Since there is an obvious solution with $U = 1$, we may drop (2) from the program. We are left with the following simplified program:

$$
\begin{array}{lrrcll}
(6) & \text{Minimize} & U & = & \displaystyle\sum_{i=1}^{n} \frac{C_i}{T_i} & \\
(7) & \text{Subject to} & T_{i+1} & = & T_i + \beta C_i, & i < n-1 \\
(8) & & T_{n-1} & = & T_n & \\
(9) & & \displaystyle\sum_{i=1}^{n-2} (\beta+1) C_i + \beta C_{n-1} + C_n & = & T_n &
\end{array}
$$

The constraints (7) and (8) suggest that the T_i are independent variables in this program, hence, we may remove the C_i from the objective function using the identities $C_i = (T_{i+1} - T_i)/\beta$, $1 \leq i < n-1$, and from the constraints (8) and (9), we see that in any minimal solution, $C_n = 0$ and $C_{n-1} > 0$ (because if $C_n > 0$ the computation of C_n could be shifted into task C_{n-1}, reducing U).

Hence $U = (1/\beta)[\sum_{i=1}^{n-2}(T_{i+1}/T_i - 1)] + \frac{1}{\beta T_{n-1}}[T_n - \frac{1+\beta}{\beta}(T_{n-1} - T_1)]$. Taking partial derivatives with respect to T_i and setting them equal to zero, we conclude that $T_i = T_1 \alpha^{i-1} =$

$T_1(\frac{1+\beta}{\beta})^{(i-1)/(n-1)}$, $\beta C_i = T_{i+1} - T_i = T_1\alpha^{i-1}(\alpha - 1)$, and hence $C_i/T_i = \frac{1}{\beta}(\alpha - 1)$, $i < n-1$ is a general solution to this program. Checking the result we see that $C_i \geq 0$, $T_i \geq 0$ for all i, so we have a feasible solution [11]. Therefore,

$$U = \sum_{i=1}^{n} C_i/T_i \leq \frac{1}{\beta}((n-1)((\frac{1+\beta}{\beta})^{1/(n-1)} - 1) + 1 - \frac{1}{\beta}) \quad \textit{if} \ \ \beta \geq 1.$$

Case 2. *Assume that task τ_{n-1} makes two requests in the time period* $[0, T_n]$. Then the results of Lemma 3.4 and the equality arguments from Case 1 yield the set of formulae given in (10), (11), and (12) below.

$$\begin{array}{llrcl}
(10) & \text{Minimize} & U & = & \sum_{i=1}^{n} \frac{C_i}{T_i} \\
(11) & \text{Subject to} & T_{i+1} & = & T_i + \beta C_i, \quad i < n \\
(12) & & \sum_{i=1}^{n-1} (\beta+1)C_i + C_n & = & T_n
\end{array}$$

Solving this non-linear program as before, we obtain $U \leq \frac{n}{\beta}((1+\beta)^{1/n} - 1) \quad \Box$ if $\beta \geq 1$.

3.1 Discussion

The worst-case performance from the formulae is depicted in Figure 2. The first case predominates for nearly all β. For example if $\beta \geq 1.25$ then the first case predominates. For $n > 2$ we must have $\beta \approx 1$ for the second case to predominate. As can be seen from the figure, the performance is slightly less than $1/\beta$ for most values of β. This is expressed in the following corollary.

COROLLARY 3.1. *As $n \to \infty$ and for $\beta > 1.25$ the task system is schedulable if $U \leq \frac{1}{\beta}(\ln \frac{1+\beta}{\beta} + 1 - \frac{1}{\beta})$. In other words we must have $U \leq \frac{1}{\beta} - O(1/\beta^3)$.*

The non-linear programming formulation that we used to find the bounds has not been used in previous papers on worst-case scheduling analysis. The only similar work has been our previous work on linear programming models for aperiodic scheduling [5]. In previous work on static-priority scheduling, all steps in the proof used ad-hoc arguments. With this new approach,

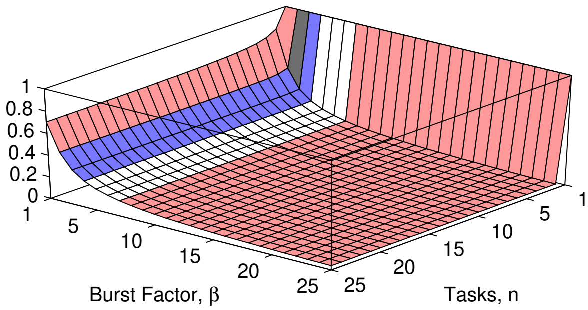


Fig. 2. *Worst-case utilization of bursty static-priority scheduling.*

it becomes clear that just 3 or 4 facts (constraints) are needed to establish the problem, and mechanical methods can be used to solve the problem thereafter.

4 Pipelined Bursty Workloads

The worst-case utilization bounds for bursty scheduling are discouraging. For example, if $\beta = 3$, the worst-case utilization approaches .3181 as $n \to \infty$. The standard way to get better utilization is to let the deadline exceed the period length. To do this we allow several outstanding requests for the same task to exist and the result is a *pipelined* real-time system. In this section we derive the worst-case utilization when the deadline is an integer multiple ($\Delta = 1, 2, 3, \cdots$) of the period length. In other words, $D_i = \Delta T_i$ for all tasks τ_i. We want to know how large Δ should be, in order to offset the scheduling loss due to bursty requests.

It is possible to derive a utilization formula for Δ an arbitrary rational multiple of the period length. However, this would involve solving and taking the minimum of several non-linear programs, which is beyond the scope of this paper. Our work follows Lehoczky's work [8] on pipelined workloads. The utilization theorem can be proved with some refinements to the non-linear program from the previous section.

Theorem 4.1. *If a critical instant occurs when all tasks request execution simultaneously, then a bursty pipelined task system is schedulable according to the rate-monotonic priority scheme if its utilization is less than the minimum of (16) and (21).*

Proof. As before, we divide the proof into two cases, one in which τ_{n-1} makes $\Delta+1$ requests, and one in which τ_{n-1} makes only Δ requests. Note that τ_{n-1} must make at least Δ requests or else it would have lower priority than τ_n.

Case 1. *Assume that $T_1 \cdots T_{n-1}$ all make $\Delta+1$ requests before T_n finishes.* By application of lemmas very similar to 3.2 and 3.3, we arrive at the following non-linear program

$$\text{(13)} \qquad \text{Minimize} \qquad U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

$$\text{(14)} \qquad \text{Subject to} \qquad \Delta T_{i+1} = \Delta T_i + \beta C_i, \quad 1 \le i \le n-1$$

$$\text{(15)} \qquad \sum_{i=1}^{n-1} (\beta+\Delta) C_i + C_n = \Delta T_n$$

Solving this program as we did before yields a formula $U = \frac{\Delta}{\beta}(n(\beta+\Delta)^{1/n} - n - \Delta + 1)$, but this formula is sometimes too pessimistic. The formula sometimes represents a task system with $C_n < 0$ for certain values of Δ, β, and n. In fact, if $\beta + \Delta \le \Delta^n$, then it can be shown that $C_n \le 0$ in the solution just given. Consequently, we must enforce the restriction $C_n \ge 0$.

When $\beta + \Delta \le \Delta^n$, then we set $C_n = 0$ and solve the nonlinear program. This program can be solved using the results of Strosnider's work [8] [16], which contains separate solutions for when $C_n < 0$ and for when $C_n \ge 0$. When combined with the previous solution for $\beta + \Delta > \Delta^n$, we arrive at the worst-case utilization when all the higher priority tasks make two requests:

$$\text{(16)} \qquad U_2(n,\beta,\Delta) = \begin{cases} 1 & \textit{if } \; n = 1, \\ \frac{\Delta}{\beta}(n-1)\left(\left(\frac{\beta+\Delta}{\Delta}\right)^{1/(n-1)} - 1\right) & \textit{else if } \; \beta+\Delta \le \Delta^n, \\ \frac{\Delta}{\beta}(n(\beta+\Delta)^{1/n} - n - \Delta + 1) & \textit{else.} \end{cases}$$

Case 2. *Assume that $\tau_1 \cdots \tau_{n-2}$ make $\Delta+1$ requests but τ_{n-1} makes only Δ requests before τ_n finishes.* Then the worst-case utilization is given by the non-linear program:

$$\text{(17)} \qquad \text{Minimize} \qquad U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

$$\text{(18)} \qquad \text{Subject to} \qquad \Delta T_{i+1} = \Delta T_i + \beta C_i, \quad 1 \le i \le n-2$$

$$\text{(19)} \qquad \Delta T_{n-1} = \Delta T_n$$

$$\text{(20)} \qquad \sum_{i=1}^{n-2} (\beta+\Delta) C_i + (\beta+\Delta-1) C_{n-1} + C_n = \Delta T_n$$

Solving this program, we obtain

$$(21)\qquad U_1(n,\beta,\Delta)=\begin{cases}1 & \textit{if } n=1,\\ \frac{\Delta}{\beta}\left((n-1)\left(\left(\frac{\beta+\Delta}{\beta+\Delta-1}\right)^{1/(n-1)}-1\right)+1-\frac{\Delta}{\beta+\Delta-1}\right) & \textit{else.}\end{cases}$$

The worst-case utilization is given by the minimum of each quantity, i.e. $U(n,\beta,\Delta) = \min\{U_2(n,\beta,\Delta), U_1(n,\beta,\Delta)\}$. □

4.1 Discussion

Unlike the formulae for pipelined scheduling in [8], the formula of Theorem 4.1 has a smooth transition point between the different cases. If $\beta+\Delta=\Delta^n$ then both cases reduce to the following result.

Corollary 4.1. *If $\beta+\Delta=\Delta^n$, then $U_1(n,\beta,\Delta)=U_2(n,\beta,\Delta)=(n-1)(\Delta-1)/(\Delta^{n-1}-1)$.*

Figure 3 plots the worst-case utilization formula in a typical system with $n = 25$ periodic tasks. This figure is somewhat misleading because the surface is known to have a stair-step behavior when Δ is not an integer [8]. If Δ is not an integer, then $\lfloor\Delta\rfloor$ may be used in a pessimistic test for task system schedulability.

In the figure, a line crosses horizontally across the middle of the surface plot, where $\Delta=\beta$, and U ranges from .7059 on the left corner of the figure to .5100 on the right corner. This area is depicted in light grey. The bursty scheduling bound when $\Delta = 1$ and β is arbitrary appears at the left forefront of the figure. It is easy to check that the curve at the left foreground of Figure 3 matches the curve at the left foreground edge of Figure 2 as it should. It is also easy to check that the curve at the left background edge matches Lehoczky's result [8].

We are interested in the performance of bursty scheduling of tasks that have compensating pipelined deadlines. We want to determine $\Delta = f(U,\beta)$, the degree of pipelining necessary to achieve a given utilization bound U for a given β. The shading in Figure 3 is chosen to depict the level curves of the function, and indicates the result. Topologically, the level curves radiate

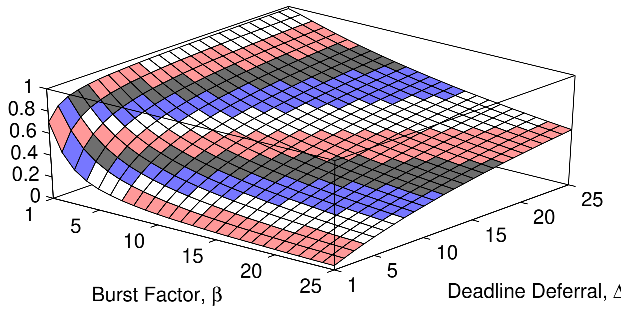


Fig. 3. *Worst-case utilization with 25 tasks, bursts, and pipelining.*

from the left corner of the surface, when $\Delta \in [1, 5]$ and $\beta \in [1, 5]$. The level curves appear to be linear, with a small aberration in crossing from $\Delta = 1$ to $\Delta = 2$. The following lemma interprets an important property of this result.

Corollary 4.2. *As $n \to \infty$ and for large β, Δ we have $U(n, \beta, \Delta) \to \frac{\Delta}{\beta+\Delta}$.*

Proof. The proof is immediate by observing the first term within the outer parenthesis of (21) converges to $\ln \frac{\beta+\Delta}{\beta+\Delta-1}$ which in turn goes to zero. □

The results of this corollary are depicted at the right corner of Figure 3. If $\beta = \Delta$, then 50% of the processor may be used to schedule periodic bursty real-time tasks.

The results of this section hinge on the assumption that the longest response time of a task occurs at the beginning of a level-n busy period [8]. Lehoczky gives examples to show that this is not true in general, however, he asserts that it is true for task systems with "low" utilization. To get around this problem, we can make use of our formula by using the following theorem and technique.

Theorem 4.2. *For each task system $\mathbf{T} = \{\tau_1, \cdots, \tau_n\}$ where the longest response time of τ_n does not occur on the first request in a level-n busy period, there is another task system* **T'** *$= \{\tau_1, \cdots \tau_{n+1}\}$ with $U' = U + \epsilon$ where the longest response time of task τ_{n+1} occurs on the first request in the level-$(n+1)$ busy period.*

Proof. The proof is immediate by adding a new task τ'_{n+1} with $T_{n+1} = T_n$ and $C_{n+1} = \epsilon\, T_{n+1}$.

Assume there are k periodic bursty pipelined requests to be scheduled. It may be the case that the longest response time does not occur at the beginning of the level-n busy period. However, if we were to add a task T_{n+1} to the task system according to the lemma, and the task system was still schedulable according to the lemma, then clearly task T_n would be schedulable because T_{n+1} met its deadline, T_n has the same deadline, and T_n has a higher priority.

Thus, we may plug in $n = k + 1$ into the utilization formulae and are assured that if the task system still passes the utilization test then it will be schedulable.

5 Non-preemptive Scheduling

Many of the applications discussed in the introduction require non-preemptive task scheduling. This can be achieved by using a utilization test similar to the test proposed for the priority ceiling protocol [14]. Under the priority-ceiling protocol a high priority task waits for at most one lower-priority critical section. Moreover, the lower-priority critical section has a ceiling at least as high as the high priority task.

In a periodic real-time resource management system, each resource will have its own work queue and schedule. Each resource request will typically be *entirely non-preemptive*, i.e. each request resembles a task that effectively locks one semaphore at its start and executes to completion before unlocking its semaphore. Examples of resource systems include packet networks (where a single packet cannot be preempted), disk drives (where a seek-read/write cycle can only be preempted by abandonment), and input-multiplexed A/D converters (where a read cycle cannot be preempted without resetting the converter entirely.)

In these situations each task conflicts with every other task, and hence, a task can wait for at most one lower priority task from among any of the lower priority tasks. Clearly, in the worst case the lower priority task could be a bursty request of maximum length.

Theorem 5.1. *A non-preemptive periodic bursty pipelined task system is schedulable under the rate-monotonic scheduling algorithm if for all* $1 \leq k \leq n$, $\sum_{i=1}^{k} \frac{C_i}{T_i} + \frac{\beta B_k}{\Delta T_k} \leq U(n, \beta, \Delta)$, *where*

B_k is the processing time of the longest critical section among all tasks at priority levels lower than k.

Proof. Consider a worst-case bursty-pipelined critical-instant schedule. The periods and computation times of this schedule can be deduced from the formulae in Theorem 4.1. In this schedule, Δ periods elapse before τ_n misses its deadline. In a non-preemptive schedule, τ_n may also be delayed by one lower priority task. Consider adding a new task $\tau_{n'}$ to this task system, with $T_{n'} = T_n - \epsilon$, $\epsilon \approx 0$, and $C_{n'} = \beta B_k/\Delta$. Assume that $T_{n'}$ never makes a bursty request, i.e. $\beta' = 1$ for this task. Then in a worst-case preemptive critical instant schedule of $\mathbf{T} \cup T_{n'}$ where T_n completes right before its deadline, task $\tau_{n'}$ will delay task τ_n by exactly $\Delta * (\beta B_k/\Delta) = \beta B_k$ time units in the interval $[0, \Delta T_n]$.

In a non-preemptive schedule according to the PCP Theorem [14] τ_n will be delayed for at most βB_k time units before it finishes its execution,

As $\epsilon \to 0$, $T_{n'} \to T_n$ and the work completed by task $\tau_{n'}$ in the time period $[0, T_n]$ approaches βB_k. Consequently, if $\sum_{i=1}^{n} C_i/T_i \leq U(n, \beta, \Delta) - C_{n'}/T_{n'}$, then there will be enough time in the interval $[0, T_n]$ for task $T_{n'}$ to complete Δ execution requests. Moreover, if there is enough time for $T_{n'}$ to complete Δ execution requests, then there is enough time for one request of length βB_k. $\square$

6 Conclusion

We have derived schedulability tests for bursty and for bursty-pipelined rate-monotonic scheduling algorithms. These tests indicate the minimum amount of pipelining necessary in order to offset the schedulability loss due to bursty requests. We have also produced a schedulability test for periodic non-preemptive requests. Resource requests are often of varying sizes and it is common for the resource manager in a computer system to support several outstanding requests.

For future work we are investigating algorithmic schedulability tests for individual bursty pipelined workloads. It is an open question as to whether an optimal schedulability test can run in pseudo-polynomial time. Also, we plan to use some recent disk models [2] to calculate the costs and benefits of amortizing disk seeks and disk reads when bursty pipelined scheduling is employed. Another interesting future extension would be the design and analysis of an ATM switch that can guarantee a substantial real-time workload. Such a switch may require a prohibitive amount of memory in order to service a wide range of bursty real-time workloads.

References

[1] R. K. Abbott and H. Garcina-Molina, *Scheduling I/O requests with deadlines: a performance evaluation*, Proceedings of the 11th IEEE Symposium on Real-Time Systems, Dec. 1990, pp. 113-124.

[2] R. Cheng, M. Greenstreet and D. W. Gillies, *Blueprint for a hard real-time file service*, submitted to IEEE Real-Time Systems Symposium, 1995.

[3] S. J. Daigle and J. K. Strosnider, *Disk scheduling for multimedia data streams*, Proceedings of the Conference on High-Speed Networking and Multimedia Computing, Feb. 1994, pp. 212-223.

[4] D. W. Gillies, *Iridium Payload Software Performance Analysis*, Motorola Satellite Communications working paper, August 1996.

[5] D. W. Gillies, J. W.-S. Liu, *Scheduling Tasks with AND/OR Precedence Constraints*, SIAM J. on Computing, August 1995.

[6] C. L. Liu and J. Layland, *Scheduling algorithms for multiprogramming in a hard real-time environment*, Journal of the ACM, vol. 10 no. 1, 1973, pp. 46-61.

[7] J. W.-S. Liu, K. J. Lin, W.-K. Shih, A. Yu, J.-Y. Chung and W. Zhao, *Algorithms for scheduling imprecise computations*, IEEE Computer Magazine, vol. 24 no. 5, May 1991, pp. 58-68.

[8] J. Lehoczky, *Fixed priority scheduling of periodic task sets with arbitrary deadlines*, IEEE Real-Time Systems Symposium, December 1990, pp. 201-213.

[9] J. Lehoczky, L. Sha and J. Strosnider, *Enhanced aperiodic responsiveness in hard real-time systems*, IEEE Real-Time Systems Symposium, December 1987, pp. 261-270.

[10] J. Lehoczky and L. Sha, *Performance of real-time bus scheduling algorithms*, ACM Performance Evaluation and Sigmetrics, May 1986, pp. 35-43.

[11] A. L. Peressini, F. E. Sullivan, J. J. Uhl, *The Mathematics of Nonlinear Programming*, New York: Springer Verlag, 1988.

[12] A. L. Reddy and J. Wyllie, *Disk scheduling in a multimedia I/O system*, Proceedings of the First ACM International Conference on Multimedia, Aug. 1993, pp. 405-420.

[13] L. Sha, J. Lehoczky, and R. Rajkumar, *Solutions for some practical problems in prioritized preemptive scheduling*, IEEE Real-Time Systems Symposium, December 1986, pp. 181-191,

[14] L. Sha, R. Rajkumar and J. P. Lehoczky, *Priority inheritance protocols: An approach to real-time synchronization*, IEEE Transactions on Computers, vol. 39 no. 9, September 1990, pp. 1175-1185.

[15] W. K. Shih, J. W. S. Liu and C. L. Liu, *Modified rate-monotonic algorithm for scheduling periodic jobs with deferred deadlines*, IEEE Transactions on Software Engineering, vol. 19 no. 12, January 1994, pp. 1171-1179.

[16] J. Strosnider, J. P. Lehoczky and L. Sha. *The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments*, Technical Report, Carnegie Mellon University, 1989.

[17] K. Tindell, *An extendible approach for analyzing fixed priority hard real-time tasks*, Real-Time Systems, vol. 6 no. 4, March, 1994, pp. 133-151.