Scheduling for Overload in Real-Time Systems

Sanjoy K. Baruah, *Member*, *IEEE*,
and Jayant R. Haritsa, *Member*, *IEEE*

Abstract—No on-line scheduling algorithm operating in an uniprocessor environment can guarantee to obtain a useful processor utilization greater than 0.25 under conditions of overload. This result holds in the general case, where the deadlines of the input tasks can be arbitrarily "tight." We address here the issue of improving overload performance in environments where there is a *limit* on the tightness of task deadlines. In particular, we present a new scheduling algorithm, ROBUST, that efficiently takes advantage of these limits to provide improved overload performance and is asymptotically optimal. We also introduce the concept of *overload tolerance*, wherein a system's overload performance never falls below its design capacity, and describe how ROBUST may be used to construct overload tolerant systems.

Index Terms—Real-time systems, uniprocessor scheduling, overload tolerance, performance evaluation, processor utilization.

1 INTRODUCTION

BROADLY defined, a real-time system is a processing system that is designed to handle workloads whose tasks have completion deadlines. The objective of the real-time system is to meet these deadlines; that is, to process tasks before their deadlines expire. Therefore, in contrast to conventional computer systems where the goal usually is to minimize task response times, the emphasis here is on satisfying the timing constraints of tasks.

In order to achieve the goal of meeting all task deadlines, the designers of safety-critical real-time systems typically attempt to anticipate every eventuality and incorporate it into the design of the system. Such a system would, under ideal circumstances, never miss deadlines and its behavior would be as expected by the system designers. In reality, however, unanticipated emergency conditions may occur wherein the processing required to handle the emergency exceeds the system capacity, thereby resulting in missed deadlines. The system is then said to be in *overload*. If this happens, it is important that the performance of the system degrade gracefully (if at all). A system that panics and suffers a drastic fall in performance in an emergency is more likely to contribute to the emergency, rather than help solve it.

A practical example of the above situation is the classic Earliest Deadline First scheduling algorithm [14], which is used extensively in uniprocessor real-time systems. This algorithm, which processes tasks in deadline order, is optimal under normal (nonoverload) conditions in the sense that it meets all deadlines whenever it is feasible to do so [7]; however, under overload, it has been observed to perform more poorly than even *random* scheduling [12], [11]. In fact, continuing to use the Earliest Deadline First algorithm in an emergency is probably the worst thing that a system can do! In this paper, we address the issue of designing real-time scheduling algorithms that are *resistant* to the effects of system overload.

- *S.K. Baruah is with the Department of Computer Science, Votey Building, University of Vermont, Burlington, VT 05405. E-mail: sanjoy@cs.uvm.edu.*
- *J.R. Haritsa is with the Supercomputer Education and Research Centre, Indian Institute of Science.*

Manuscript received 22 Sept. 1993; revised 2 Oct. 1995.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 105219.

0018-9340/97/$10.00 © 1997 IEEE

1.1 Overload Performance Metrics

A general real-time model adopted in many studies is the "firm-deadline" model [10]. In this model, only tasks that *fully* complete execution before their deadlines are considered to be successful, whereas tasks that miss their deadlines are considered worthless and are immediately discarded without being executed to completion. For the firm-deadline model, two contending measures of the "goodness" of a scheduling algorithm under conditions of overload are **effective processor utilization (EPU)** and **completion count (CC)**. Informally, EPU measures the fraction of time during overload that the processor spends on executing tasks that complete by their deadlines, while CC measures the number of tasks executed to completion during the overload interval. Which measure is appropriate in a given situation depends, of course, upon the application. For example, EPU may be a reasonable measure in situations where tasks ("customers") pay at a uniform rate for the use of the processor, but are billed only if they manage to complete, and the aim is to maximize the value obtained. By contrast, CC makes more sense when a missed deadline corresponds to a disgruntled customer, and the aim is to keep as many customers satisfied as possible.

The research described in this paper is focused on studying the impact of overload in uniprocessor real-time systems when **EPU** is the measure of scheduler performance.[1] The EPU metric has been widely used in the analysis of real-time scheduling algorithms under conditions of overload (e.g., [15], [4], [3], [13], [17]). A detailed discussion of the applicability of this metric to real-time systems is provided in [15]. In particular, our goal is to compare the EPU performance of *on-line* scheduling algorithms against that of an optimal off-line (or clairvoyant) algorithm. On-line schedulers make scheduling decisions at run-time and typically do not possess prior knowledge about the occurrence of future events—such schedulers are used in many real-time systems. In our performance evaluation, we wish to compare the *worst-case* EPU performance of on-line schedulers with respect to the optimal off-line scheduler; that is, we are interested in **performance guarantees**.

EXAMPLE 1. To illustrate the notion of EPU, consider a uniprocessor system where a task T_1 makes a request at time 0 for three units of processor time by a deadline of four, and task T_2 makes a request at time 1 for eight units of processor time by a deadline of 10 (as shown in Fig. 1). Clearly, no scheduler can schedule both T_1 and T_2 to completion. In this situation, a scheduler that schedules T_1 first to completion and then executes T_2 has an EPU of 0.3 over [0, 10) since T_2 has to be discarded at time 10. On the other hand, a scheduler that executes task T_1 during [0, 2), and then switches to executing T_2 to completion during [2, 10) has an EPU of 0.8 over [0, 10).

This example shows that, when forced to make a choice as to which tasks to complete and which tasks to discard, it is better to selectively complete "larger" tasks, that is, tasks with larger execution times, since they contribute more to the EPU.

1.2 The ROBUST Scheduler

The optimality results in [7] ensure that *Earliest Deadline First* schedulers, which execute tasks in deadline order, guarantee an EPU of 1.0 under normal (nonoverload) conditions. Furthermore, it has been shown [4], [3] that *no* uniprocessor on-line scheduling algorithm can guarantee an EPU greater than 0.25 under overload (this bound is tight). Taken in conjunction, these results imply that the onset of an emergency may, in general, force a deterioration in system performance by as much as a factor of *four*.

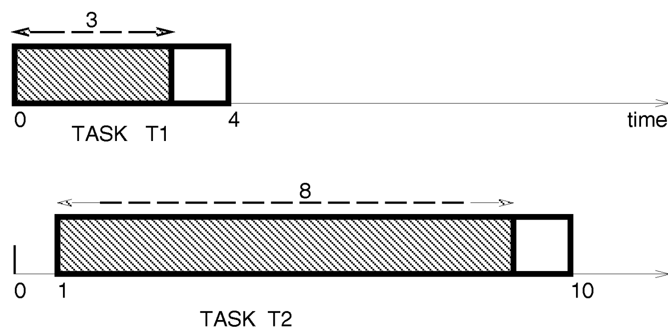

Fig. 1. Effective processor utilization (EPU).

The above results are applicable for the general case where the deadlines of individual tasks can be arbitrarily "tight" or stringent. As Stankovic et al. have pointed out in a recent survey article [17, p. 21], these are "very restrictive assumptions [and have] only theoretical validity. More work is needed to derive other bounds based on more knowledge of the task set." Our research is directed at taking advantage of prior knowledge on the tightness of deadlines in the task set.

A quantitative measure of the tightness of a task is given by its *slack factor*, which is defined to be the ratio between the task's deadline and its execution requirement. In the research described in this paper, we investigate the extent to which degradation in overload performance can be reduced in environments where all tasks are guaranteed to have a *minimum* slack factor. In particular, we study the effect of slack factor on the EPU performance guarantee of scheduling algorithms under overload. We present ROBUST (Resistance to Overload By Using Slack Time), a new on-line uniprocessor scheduling algorithm that performs efficiently during overload for a large range of slack factors, and is in fact **asymptotically optimal** with increasing slack factor.

1.3 Overload Tolerance

Ideally, one would like a safety-critical system to enhance its performance upon the onset of an emergency in order to better deal with the emergency. If this is not possible, we would, at the very least, like the system to continue to provide the same level of performance as was provided before the emergency occurred. Based on this observation, we introduce here the notion of **overload tolerance**: We define a safety-critical system to be overload tolerant if the performance of the system during overload never degrades to below its *maximal* normal performance.

The 0.25 bound on EPU [4], [3] implies that overload tolerance cannot, in general, be guaranteed by any on-line scheduling algorithm. We explore, in this paper, the possibility of using *faster hardware* to compensate for this fall in performance during overload. At first glance, one may expect that hardware four times as fast as the original is necessary in order to compensate for the four-fold performance degradation; however, we show here that there exist situations wherein using the ROBUST scheduling algorithm in conjunction with hardware merely *twice* as fast, suffices to guarantee overload tolerance.

1.4 Related Work

While there exists a large body of literature devoted to real-time scheduling (see [6] for a survey), much of this work has dealt with situations where the task arrival process is a priori sufficiently characterized. On-line scheduling algorithms, however, often have to contend with situations where prior knowledge of the future is not available. For this type of environment, schedulers such as

1. Recently, we have also begun investigating the impact of overload with respect to CC, and an initial set of results for this metric are presented in [2].

Best Effort [12] and AED (Adaptive Earliest Deadline) [11] have been proposed. Results of simulation-based experiments suggest that these algorithms perform well under a variety of overload situations. However, since these algorithms are based on heuristics, pathological conditions exist where their performance can be arbitrarily poor; they cannot, therefore, provide any worst-case *guarantee* on EPU performance.

Of late, there has been considerable activity in theoretical studies of overload scheduling algorithms [3], [4], [13], [8]. A common feature of these studies is that they all consider workloads where tasks can have arbitrary slack factors. In contrast, our study investigates the performance improvement that can be realized for workloads where all tasks are guaranteed to have a prespecified minimum slack factor.

1.5 Organization

The remainder of this paper is organized as follows: In Section 2, we precisely define our model and the notions of EPU and overload. We present the new algorithm, ROBUST, in Section 3 and characterize its EPU performance. In Section 4, ROBUST is proved to be asymptotically optimal. We then discuss methods of achieving overload tolerance, using ROBUST, in Section 5. Finally, in Section 6, we conclude with a summary of the results presented here.

2 Model and Definitions

In this section, we describe the workload and system models employed in our study. In addition, various definitions used in the subsequent proofs are detailed here.

2.1 Task Model

In our task model, each input task T is independent of all other tasks and is completely characterized by three attributes: $T.a$ (the **request time**), $T.e$ (the **execution requirement**), and $T.d$ (the **relative deadline**, often simply called the **deadline**). The interpretation of these parameters is that task T, for successful completion, needs to be allocated the processor for $T.e$ units of time during the interval $[T.a, T.a + T.d)$. We assume that the system learns of a task's parameters only at the instant when it makes the service request ($T.a$) and that there is no a priori bound on the number of tasks that may be generated by the real-time application executing on the system. Finally, only tasks that fully complete execution by their deadlines are of value to the user application; that is, all deadlines are *firm* [10].

The **slack factor** of task T is defined to be the ratio $T.d/T.e$ and is denoted by $T.s$; it is a quantitative indicator of the tightness or slackness of the task deadline. It is trivial to see that it is necessary that $T.s$ be ≥ 1 for it to be at all possible to complete a task before its deadline. In this study, we consider task sets where it is known a priori that all tasks in the task set will have a slack factor of at least f, where $f \geq 1$ is a prespecified constant.

A task T is said to be **active** at time-instant t if

1) it has requested service by time t (i.e., $T.a \leq t$),
2) its service is not complete (i.e., $T.e_r > 0$, where $T.e_r$ is the task's remaining service requirement), and
3) its deadline has not expired (i.e., $t < T.a + T.d$).

An active task T is **feasible** at time t if $T.e_r \leq (T.a + T.d - t)$; that is, it is still possible to meet the task's deadline. On the other hand, an active task is **degenerate** if the remaining service requirement exceeds the time remaining until its deadline. Active tasks remain in the system until they either complete or their deadline expires, whichever occurs earlier.

2.2 System Model

We focus our attention in this paper on the study of **uniprocessor** real-time systems. Our scheduling model is **preemptive**; that is, a task executing on the processor may be interrupted at any instant in time and its execution resumed later. There is no cost associated with such preemption.

The system is said to be **idle** at time-instant t_0 if there are no active tasks at time t_0, or if all active tasks in the system at time t_0 have their request-times equal to t_0. (That is, we do not consider tasks that arrive *at* time t_0 in determining whether the system is idle at t_0; this is a technical detail that facilitates the definitions of the start and finish of overload, described below.)

The system is said to be in **overload** if *no* scheduling algorithm, either off-line or on-line, can meet the deadlines of all the tasks that are submitted to the system. As mentioned in the Introduction, the Earliest Deadline First algorithm is optimal in the sense that it will successfully schedule any set of task requests if it is at all possible to do so. Given this optimality of the Earliest Deadline First algorithm, it follows that a system is in overload if the Earliest Deadline First algorithm fails to meet the deadline of one or more tasks in the system. The time period for which the system is in overload is called the **overload interval**. The **start time** of this overload interval is the latest time instant t_s, $t_s \leq t$, at which the system would be idle if the Earliest Deadline First algorithm were executed on the system. The **finish time** of this overload interval is the earliest time instant t_f, $t_f \geq t$, at which the system is idle.

While the start time of an overload interval is independent of the scheduling algorithm actually used in the system, the finish time is *necessarily* dependent upon the scheduling decisions made during the overload period. Consider, as an example, the situation in Example 1. If a scheduler executes task T_1 to completion, then the overload interval terminates at time-instant 10; if, on the other hand, it executes T_2 to completion over the interval [1, 9), then the overload interval terminates at time 9, since task T_1 is not active at this time (in fact, T_1 ceases to be active at time 4).

In the Introduction, we presented an intuitive description of EPU. Based on the above terminology, we now provide a more precise definition: Given an overloaded time interval that starts at time t_s and finishes at time t_f, the **EPU** over this time interval is computed by

$$EPU = \frac{\sum_{i \in C} x_i\left[t_s, t_f\right)}{t_f - t_s},$$

where C denotes the set of tasks that successfully complete (i.e., meet their deadlines) during $[t_s, t_f)$, and $x_i[t_s, t_f)$ represents the service received by task i during $[t_s, t_f)$.

The **EPU of a system** is the *lowest* EPU performance measured over any overloaded interval and over any task sequence; that is, it is the worst-case performance guarantee. In the remainder of this paper, we will use this metric as the performance measure.

3 The ROBUST Algorithm

In this section, we present a new on-line scheduling algorithm called ROBUST (Resistance to Overload By Using Slack Time) and analyze its EPU performance. For ease of understanding, we first discuss the specific case where the slack factor of all tasks is at least 2.0, and then present the general case where the minimum slack factor may be an arbitrary value.

3.1 Minimum Slack Factor = 2

For this case, we will show that ROBUST guarantees an EPU of 0.5 during overload. We first provide a high-level overview of the operation of ROBUST; a more detailed description then follows.

3.1.1 Overview of ROBUST

1) The ROBUST algorithm partitions an overloaded interval into an even number of contiguous *phases*—Phase-1, Phase-2, …, Phase-2n.

2) The length of each even (numbered) phase is equal to that of the preceding odd (numbered) phase. That is, the length of Phase-$2i$ is equal to the length of Phase-$(2i - 1)$ for all i, $1 \leq i \leq n$.
3) The algorithm does "useful" work—i.e., executes tasks to completion—during each odd phase. It accomplishes this by *nonpreemptively* executing one task to completion during each odd phase, thus ensuring that the time spent on this task contributes to the EPU.
4) After each odd phase, the following even phase is used to *set up* the execution of the "most valuable" task for the *next* odd phase. By most valuable task, we mean the task that can immediately contribute the most to the EPU; as discussed in Example 1, this task is the largest task, that is, the task with the maximum execution time among all the currently feasible tasks.

   Note that it is not important that any task execute to completion in the even phases—any such completion is a bonus, and contributes to increasing the EPU to above the guaranteed value of one-half.

3.1.2 Detailed Description of ROBUST

Suppose that the overloaded interval begins at time t. Then, let tasks $T_1^{(1)}, T_2^{(1)}, \ldots, T_{n_1}^{(1)}$ be the set of tasks that are feasible at this time. Let task $T_{\max}^{(1)}$ be the currently most valuable task in this set from an EPU perspective; that is, $T_{\max}^{(1)}.e \geq T_i^{(1)}.e$ for all i, $1 \leq i \leq n_1$. Also, let $e_r^{(1)}$ represent the *remaining* amount of processor time that is required by task $T_{\max}^{(1)}$ at time t. Then, Phase-1 is defined to be the interval $[t, t + e_r^{(1)})$, and Phase-2 the interval $[t + e_r^{(1)}, t + 2e_r^{(1)})$.

At the start of Phase-1, the scheduler commits itself to *nonpreemptively* executing task $T_{\max}^{(1)}$ to completion; that is, for the entire duration of the phase. Suppose now that a new task request, T_{new}, arrives at some time during this phase. Since the processor must execute $T_{\max}^{(1)}$ during the entire phase, it cannot begin executing task T_{new} until after Phase-1 has terminated. We consider two cases:

$T_{\text{new}}.e \geq T_{\max}^{(1)}.e$: In this case, the new task is more valuable than the currently executing task. However, since the slack factor of task T_{new} is at least two, it follows that

$$T_{\text{new}}.d \geq 2 \cdot T_{\text{new}}.e \geq T_{\text{new}}.e + T_{\max}^{(1)}.e \geq T_{\text{new}}.e + e_r^{(1)}.$$

This means that although task T_{new} is given no service in Phase-1, it is *guaranteed* to still be feasible at the end of this phase.

$T_{\text{new}}.e < T_{\max}^{(1)}.e$: In this case, the new task is less valuable than the currently executing task, and it is possible that task T_{new} may become degenerate by the end of the phase.

The important point to note in the above cases is that committing to the nonpreemptive execution of task $T_{\max}^{(1)}$ in Phase-1 *does not* incur the danger of discarding a task that could potentially contribute more to the EPU than task $T_{\max}^{(1)}$.

Upon the termination of Phase-1, Algorithm ROBUST has executed task $T_{\max}^{(1)}$ to completion. It now switches to a *preemptive* mode of execution for Phase-2. At the start of Phase-2, the currently most valuable task is scheduled. For the duration of this phase, whenever a new task makes a request, ROBUST compares the execution requirement of the new task and the execution requirement of the currently executing task: If the execution requirement of the new task is greater, ROBUST preempts the current task and begins executing the new one, otherwise, the current task continues execution. If the currently executing task completes execution, the most valuable remaining feasible task is scheduled.

We have described above how ROBUST behaves in the first pair of phases. The remaining odd and even phases are executed in an identical manner to that just described for the first odd phase and the first even phase, respectively. That is, during each odd phase Phase-$(2j - 1)$, ROBUST nonpreemptively executes the task $T_{\max}^{(j)}$, which has the largest execution requirement of all the feasible tasks that are active at the start of the phase, and, during each even phase Phase-$2j$, ROBUST switches to a preemptive mode, such that, throughout the phase, the processor is at each instant executing the currently active task with the largest execution requirement. The process ends if, at the termination of a phase, the system is found to have become idle, signaling the termination of overload—it is easy to prove that this can only happen at the end of an even phase, thereby guaranteeing that the total number of phases in the overload interval is even.

We now prove that the ROBUST algorithm described above guarantees an EPU of 0.5.

THEOREM 1. *The ROBUST algorithm achieves an EPU of at least 0.5 during the overload interval when all tasks have a slack factor of at least two.*

PROOF. Suppose that the ROBUST algorithm divides the overload interval into $2n$ phases numbered 1 through $2n$. Observe that the processor is guaranteed to be "useful," (i.e., executing tasks that do complete by their deadlines) during all the odd phases. Furthermore, the length of each odd phase is exactly equal to the length of the succeeding even phase. The EPU over the entire overloaded interval is, therefore,

$$\geq \frac{\sum_{i=1}^{n}\left[\text{length of Phase-}(2i-1)\right]}{\sum_{j=1}^{2n}\left[\text{length of Phase-}j\right]} = \frac{1}{2}.$$

□

3.2 Arbitrary Minimum Slack Factors

Theorem 1 shows that when we move from general task sets which have no slack constraints (i.e., minimum slack factor of one) to task sets with minimum slack factor of two, the EPU guarantee goes up from 0.25 to 0.5. In this subsection, we extend our analysis to profile the improvement in EPU performance when the minimum slack factor is an arbitrary value, f. We present a generalized version of the ROBUST algorithm and show that it provides an EPU of $(f - 1)/f$ during the overload interval.

3.2.1 The Generalized ROBUST Algorithm

Consider an environment where all tasks are guaranteed to have a slack factor of at least f, $f > 1$. The Generalized ROBUST algorithm behaves exactly like the ROBUST algorithm described in Section 2, except that the length of every even phase Phase-$2i$ is set to $1/(f - 1)$ times the length of the preceding odd phase Phase-$(2i - 1)$. Using proof techniques similar to those of Theorem 1, it is straightforward to prove that the processor is "useful" during all the odd phases, yielding the following theorem:

THEOREM 2. *The Generalized* ROBUST *algorithm achieves an EPU of at least $(f - 1)/f$ during the overload interval.*

When we refer to the ROBUST algorithm for the remainder of this paper, we will mean the generalized algorithm described here.

4 Optimality of ROBUST

In the previous section, we characterized the EPU performance of ROBUST. We now investigate as to whether this is the *best* performance that can be attained; that is, is ROBUST optimal?

The following theorem establishes an upper bound on the EPU that is attainable for task sets with arbitrary minimum slack factor.

THEOREM 3. *No on-line scheduling algorithm can guarantee an EPU greater than* $\lceil f \rceil / (\lceil f \rceil + 1)$ *during overload in an environment where all incoming tasks have a slack-factor of at least f.*

PROOF. Construct a set of $2\lceil f \rceil$ tasks which all have an execution requirement of 1.0 and a relative deadline of f. Then, assign a request time of 0.0 to $\lceil f \rceil$ tasks and a request time of $1 - \epsilon$ ($0 < \epsilon < 1$), to the remaining $\lceil f \rceil$ tasks. It is simple to see that while it is straightforward to schedule $\lceil f \rceil$ tasks to completion, no on-line scheduler can successfully schedule $\lceil f \rceil + 1$ tasks. For $\epsilon \to 0$, therefore, an EPU greater than $\lceil f \rceil / (\lceil f \rceil + 1)$ cannot be obtained on the overloaded interval $[0, \lceil f \rceil + 1 - \epsilon)$. □

An important point to note here is that the bound established by the above theorem is *not tight*. For example, for $f = 2$, it gives an EPU bound of 0.67, whereas we have been able to separately prove a bound of 0.625 for this case [1]. However, the theorem does establish that the Generalized ROBUST algorithm provides a performance that is at most

$$1 - \frac{(f-1)/f}{\lceil f \rceil / (\lceil f \rceil + 1)} \leq 1 - \frac{(f-1)/f}{(f+1)/(f+2)} = \frac{2}{f(f+1)}$$

fractionally off from the optimal. Thus, with increasing slack factor, the ROBUST algorithm is **asymptotically optimal**. As a practical matter, the ROBUST algorithm is guaranteed to be *within 10 percent* of the optimal for slack factors of 4.0 or greater (i.e., $2/(f(f+1)) \leq 0.1$ for all $f \geq 4$). Furthermore, depending on the looseness of the bound of Theorem 3, the ROBUST algorithm may turn out to be within 10 percent of the optimal at even lower slack factors. In fact, it is even possible that the algorithm may *itself* be optimal—we are currently studying this issue. In summary, the ROBUST scheduler appears to provide, by using task slack times, a reasonably efficient solution to the problem of performance degradation during overload. In the next section, we move on to discussing how ROBUST can be used to achieve overload tolerance.

5 Overload Tolerance

As mentioned in the Introduction, we define a safety-critical system to be **overload tolerant** if the performance of the system under conditions of overload never degrades to below its maximal normal performance. The 1/4 bound on EPU in overload conditions [4] implies that no on-line scheduling *algorithm* can in itself guarantee overload tolerance.

One method of achieving overload tolerance in uniprocessor systems, despite this inherent limitation, is to ensure that the processor is not permitted to become overloaded in the first place. This could be achieved, for example, by assigning *values* to all tasks in the system, and choosing for execution a maximal-valued subset of tasks (from among the set of all tasks making requests) which do not overload the processor. (The problem of determining such a maximum-valued subset is, in fact, related to the Knapsack Problem, which is known to be NP hard [9].) In any event, such an approach is necessarily application-specific, in that the assignment of values to individual tasks must be made based upon the unique characteristics of the particular application system that is being designed: e.g., the importance of the task to the system.

In this section, we propose a mechanism for achieving this goal which is based on using *faster hardware.*

Consider a workload with no slack restrictions except that all tasks be feasible upon arrival, that is, $T.s \geq 1$ for all tasks. If these tasks are executed on hardware that is f times as fast as the hardware for which they had been specified, the execution requirement of the tasks is reduced to $\frac{T.e}{f}$—their slack factor is therefore $\geq f$. The behavior of a system whose hardware is upgraded in this fashion changes in two ways from that exhibited by the original system:

- First, the system is less likely to go into overload, since its "capacity" is greater. Any load that is no more than f times the capacity of the original system will not push the system into overload.
- For larger loads, overload *will* occur. But, if the ROBUST algorithm is used to schedule tasks during the overloaded time periods, the EPU will always be at least $(f - 1)$ times the original system's capacity. This is because, by Theorem 2, the performance of the system will not degrade by more than a factor of $(f - 1)/f$ from its current performance level of f times the capacity of the original system, i.e., to $(f - 1)$ times the original system's maximum capacity.

This is illustrated in Fig. 2, where performance is plotted against system load, with both axes labeled to percentages of the capacity of the original system. The beaded line profiles the behavior of the original system. The solid line represents the behavior of the system when installed on hardware *twice* as fast as the initial hardware, and with the ROBUST algorithm used for scheduling during overload. Notice that *the performance of this new system never degrades to below the maximum performance of the original system, even under extreme overloads.* This implies that, in order to make a safety-critical system overload-tolerant, it is sufficient to double the speed of the processor and use the ROBUST scheduler.

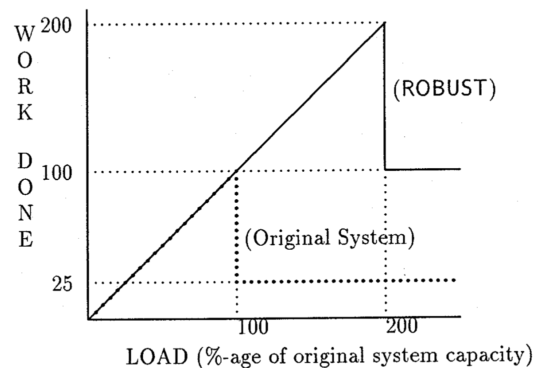

Fig. 2. The effect of load on system behavior .

5.1 Implementation Issues

We now discuss a few implementation issues associated with the above method of achieving overload tolerance.

In the above analysis, it was assumed that the use of hardware f times as fast as the original would result in a decrease in execution requirements of *all* tasks by a factor of f. This may not always be the case in actual systems: For example, if task execution times are dependent upon factors external to the system, then speedup in system hardware will not have a proportional effect on execution time reduction.

Another problem associated with designing a system to achieve the performance profile attributed to ROBUST in Fig. 2 is that the approach outlined above requires *online* identification of the onset of overload, and *switching* from an optimal scheduler (e.g., Earliest Deadline First) to ROBUST during overload. Identifying the exact beginning of an overloaded interval online is, in general, impossible. Consider, however, a set of task requests that does not over-

load the original system (i.e., a set of requests such that the load is ≤ 100% the capacity of the original system), and suppose that installing this set on hardware twice as fast as the original halves the execution requirements of all tasks in the set. In this case, it is easily verified that using ROBUST to make scheduling decisions on this set of tasks upon the new hardware results in all the tasks completing by their deadlines. When the intent is to construct overload tolerant systems, therefore, it suffices to use ROBUST under *all* circumstances, both overloaded and normal. The worst-case behavior of the system will now be as shown in Fig. 3.

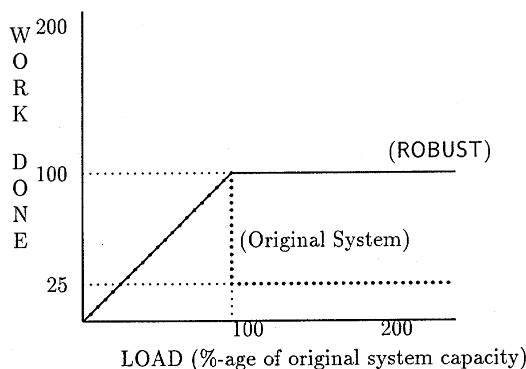

Fig. 3. Performance profile when ROBUST is used over the entire range of loads.

5.2 Optimality

We have seen that a hardware speedup by a factor of two can be sufficient for guaranteeing overload tolerance. We now show that it is absolutely *necessary* to have hardware at least twice as fast as the original in order to guarantee overload tolerance.

THEOREM 4. *For arbitrary task systems, a hardware speedup by a factor of two is necessary for overload tolerance.*

PROOF. Let ϵ be a constant, $0 < \epsilon < 1$. Let λ be a constant, $0 < \lambda < \epsilon/2$. Consider a task system T_1, T_2, with $T_1.a = T_2.a = 0$, $T_1.e = T_2.e = (1 - \lambda)$, $T_1.d = T_2.d = 1$. If this task system is now installed on hardware $(2 - \epsilon)$ times as fast as the original, the execution requirements may be reduced to $(1 - \lambda)/(2 - \epsilon)$. Since $2(1 - \lambda)/(2 - \epsilon) > 1$, only *one* of the two tasks will complete on the new hardware, yielding an EPU of $(1 - \lambda)/(2 - \epsilon)$ over the interval [0, 1). The performance of the new system is, therefore, $\text{EPU} \times \text{speedup} = \frac{1-\lambda}{2-\epsilon} \times (2 - \epsilon) = (1 - \lambda)$. Since $\lambda > 0$, $(1 - \lambda) < 1$; the performance during overload is consequently strictly less than the maximal performance under normal conditions. □

6 CONCLUSIONS

It has been shown [3], [4] that no on-line uniprocessor scheduling algorithm can guarantee an EPU greater than 0.25 under conditions of overload for arbitrary task sets. We have presented here Algorithm ROBUST (Resistance to Overload By Using Slack Time), an on-line scheduling algorithm that is not limited by the 0.25 bound for task sets that guarantee a minimum slack factor for every task. We have discussed how system designers could use the ROBUST scheduler to enhance the performance of their systems. In particular, we demonstrated that, with ROBUST, doubling the processor speed is sufficient to ensure that the system's EPU never falls below the original system's capacity.

We explored the optimality of the ROBUST algorithm and proved that it is asymptotically optimal with respect to task slack factor. Specifically, we showed that its performance is guaranteed to be within 10 percent of the optimal for slack factors greater than four.

The scheduling algorithms presented in this paper require the slack factor of all tasks to be greater than a certain minimum value in order for their performance guarantees to hold. In practice, the semantics of particular applications may permit a trade-off between slack factors of different tasks. We suggest that maximizing the minimum slack factor in a system of tasks be a design goal for the developers of safety-critical real-time systems.

ACKNOWLEDGMENTS

A preliminary version of this paper, entitled "ROBUST: A Hardware Solution to Real-Time Overhead," was presented at the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, May 1993, Santa Clara, California. Sanjoy K. Baruah was supported in part by U.S. National Science Foundation grants OSR-9350540 and CCR-9410752, and by University of Vermont grant PSCI94-3. J.R. Haritsa was supported in part by a grant from the Department of Science and Technology, Government of India.

REFERENCES

[1] S. Baruah and J. Haritsa, "ROBUST: A Hardware Solution to Real-Time Overload," *Proc. 13th ACM SIGMETRICS Conf.*, pp. 207–216, Santa Clara, Calif., May 1993.
[2] S. Baruah, J. Haritsa, and N. Sharma, "On-Line Scheduling to Maximize Task Completions," *Proc. 15th Real-Time Systems Symp.*, San Juan, Puerto Rico, Dec. 1994.
[3] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang, "On the Competitiveness of On-Line Real-Time Task Scheduling," *Proc. 12th Real-Time Systems Symp.*, San Antonio, Tex., Dec. 1991.
[4] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, and D. Shasha, "On-Line Scheduling in the Presence of Overload," *Proc. 32nd Ann. IEEE Symp. Foundations of Computer Science*, San Juan, Puerto Rico, Oct. 1991.
[5] S. Biyabani, J. Stankovic, and K. Ramamritham, "The Integration of Deadline and Criticalness in Hard Real-Time Scheduling," *Proc. Ninth Real-Time Systems Symp.*, Dec. 1988.
[6] S. Cheng, J. Stankovic, and K. Ramamritham, "Scheduling Survey," *Hard Real-Time Systems Tutorial*, Dec. 1988.
[7] M. Dertouzos, "Control Robotics: The Procedural Control of Physical Processors," *Proc. IFIP Congress*, pp. 807-813, 1974.
[8] J. Dey, J. Kurose, D. Towsley, C. Krishna, and M. Girkar, "Efficient On-Line Processor Scheduling for a Class of IRIS Real-Time Tasks," *Proc. 13th ACM SIGMETRICS Conf.*, pp. 217-228, Santa Clara, Calif., May 1993.
[9] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York: W.H. Freeman, 1979.
[10] J. Haritsa, M. Carey, and M. Livny, "On Being Optimistic about Real-Time Constraints," *Proc. 1990 ACM Principles of Database Systems Symp.*, Apr. 1990.
[11] J. Haritsa, M. Carey, and M. Livny, "Earliest-Deadline Scheduling for Real-Time Database Systems," *Proc. 12th IEEE Real-Time Systems Symp.*, San Antonio, Tex., Dec. 1991.
[12] E. Jensen, M. Carey, and M. Livny, "A Time-Driven Scheduling Model for Real-Time Operating Systems," *Proc. IEEE Real-Time Systems Symp.*, Dec. 1985.
[13] G. Koren and D. Shasha, "D^{over}: An Optimal On-Line Scheduling Algorithm for Overloaded Real-Time Systems," *Proc. 13th Real-Time Systems Symp.*, Phoenix, Ariz., Dec. 1992.
[14] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. ACM*, Jan. 1973.
[15] C. D. Locke, "Best-Effort Decision Making for Real-Time Scheduling," PhD thesis, Computer Science Dept., Carnegie Mellon Univ., 1986.
[16] A. Mok, "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment," PhD thesis, Laboratory for Computer Science, Massachusetts Inst. of Technology, May 1983.
[17] J. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo, "Implications of Classical Scheduling Results for Real-Time Systems," *Computer*, June 1995.