ROBUST: A Hardware Solution to Real-Time Overload

Sanjoy Baruah*
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

Jayant R. Haritsa†
Systems Research Center
University of Maryland
College Park, Maryland 20742

Abstract

No on-line scheduling algorithm operating in a uniprocessor environment can guarantee to obtain an effective processor utilization greater than 25% under conditions of overload. This result holds in the most general case, where incoming tasks may have arbitrary slack times. We address here the issue of improving overload performance in environments where the slack-time charactersitics of all incoming tasks satisfy certain constraints. In particular, we present a new scheduling algorithm, ROBUST, that efficiently takes advantage of these task slack constraints to provide improved overload performance and is asymptotically optimal.

1 Introduction

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 become overloaded, and its behavior would be as expected by the system designers. In reality, however, unanticipated emergency conditions may occur and it may so happen that the amount of required processor time exceeds the system capacity. 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 likely to contribute to the emergency, rather than help solve it.

*Supported in part by a research grant from the Office of Naval Research under contract number N00014-89-J-1472

†Supported in part by a Systems Research Center Post-Doctoral Fellowship

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
1993 ACM SIGMETRICS-5/93/CA,USA
© 1993 ACM 0-89791-581-X/93/0005/0207...$1.50

Under overload conditions in uniprocessor real-time systems, a natural measure of system performance is the **effective processor utilization (EPU)** of the system. Informally speaking, the *EPU* of a system over an interval of time measures the fraction of time within the interval that the processor spends on executing tasks that eventually *do* meet their deadlines. This notion is clearly illustrated in the following example.

Example 1. Consider a situation where task T_1 makes a request at time 0 for 3 units of processor time by a deadline of 4, and task T_2 makes a request at time 1 for 8 units of processor time by a deadline of 10. Clearly, no scheduler can schedule both T_1 and T_2 to completion. A scheduler that schedules T_1 first to completion and then schedules T_2 has an *EPU* of 0.3 over $[0, 10)$, while one that executes task T_1 during $[0, 2)$, and then schedules T_2 to completion by executing it during $[2, 10)$ has an *EPU* of 0.8 over $[0, 10)$.

□

It has recently been shown [1, 2] that no uniprocessor on-line scheduling algorithm can guarantee a "competitive ratio" larger than 1/4 under overload. (An on-line algorithm of *competitive ratio* r, $0 \leq r \leq 1$, is guaranteed to achieve a cumulative value at least r times the cumulative value achievable by any clairvoyant algorithm on any sequence of requests, where a task's value is equal to its execution time.) With minor modifications, the proof of this result can be extended to show that no uniprocessor on-line scheduling algorithm can

guarantee an *EPU* greater than 1/4 under overload. Taken in conjunction with the classical theorem which proves that the Earliest Deadline scheduling algorithm guarantees 100 percent *EPU* under *non*-overload conditions (see, for example, [3]), this result implies that the onset of an emergency may force a deterioration in system performance by a factor of four. This holds in the most general case where the tasks in the workload may have arbitrary *slack factors* (slack factor is the ratio between a task's deadline and its execution time, and is indicative of how "tightly" a task needs to be scheduled). The research described in this paper investigates to what extent 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 of scheduling algorithms under overload. We present **ROBUST** (Resistance to Overload By Using Slack Time), an on-line uniprocessor scheduling algorithm that performs efficiently under overload over a large range of slack factors, and is in fact asymptotically optimal with increasing slack factor.

The remainder of this paper is organized in the following fashion: In Section 2, we precisely define our model and the notions of *EPU* and overload. We present our new algorithm, ROBUST, in Section 3 and prove that it guarantees a worst-case *EPU* of one-half under conditions of overload when all incoming tasks have a slack factor of at least 2. In Section 4, we attempt to determine if this algorithm is also *optimal*. That is, are there on-line algorithms that can guarantee an *EPU* greater than one-half under such conditions? Although we do not yet have a conclusive answer to this question, we do succeed in proving an upper bound of 5/8'ths on the performance of *any* on-line algorithm in the above situation. In Section 5, we generalize the results of Sections 3 and 4 to the case where the minimum task slack factor is an arbitrary number and prove that ROBUST is asymptotically optimal. We explore methods by which desired slack factors can be achieved in Section 6, and discuss other related issues (including the rationale for the title of this paper). We conclude in Section 7 with a summary of the results presented here, and outline future research directions.

2 System Model and Definitions

We focus our attention in this paper on the study of **uniprocessor** systems. In our task model, each task T is completely characterized by three parameters: $T.a$ (the **request time**), $T.e$ (the **execution requirement**), and $T.d$ (the **relative deadline**, often simply called the **deadline**), where $T.a$ is the time at which task T makes a request for $T.e$ units of processor time by a deadline of $T.a + T.d$. Tasks must complete execution by their deadlines in order to be of any value to the system; that is, all deadlines are *hard* [6]. We assume that nothing is known about a task until it makes its request, at which time all three parameters become known. In addition, there is no *a priori* bound on the number of tasks that will be encountered. Our scheduling model is **preemptive**; i.e., a task executing on the processor may be interrupted at any instant in time, and its execution resumed later. There is no penalty associated with such preemption.

The **slack factor** of task T is defined to be the ratio $T.d/T.e$. Clearly, for a task T to complete by its deadline, it is necessary that $T.d$ be at least as large as $T.e$; the slack factor of any non-degenerate task (i.e., a task that has any chance at all of completing by its deadline) must therefore be at least one.

A system is said to be in **overload** if no scheduling algorithm can satisfy all task requests that are made on the system. As mentioned in the Introduction, the Earliest Deadline algorithm is optimal in the sense that it will successfully schedule any set of task requests which are in fact schedulable. Given this optimality of the Earliest Deadline algorithm, it follows that a system is in overload if the Earliest Deadline algorithm fails to meet the deadline of some task in the system.

A task T is said to be **active** at time-instant t if (i) $T.a \leq t$; i.e., the task arrives by time t, and (ii) $T.e_r > 0$, where $T.e_r$ is the remaining amount of processor time that needs to be allocated to task T before its deadline, and (iii) $T.d \geq t$; i.e. the deadline has not been reached. Active task T is **degenerate** at time t if $T.e_r > (T.a + T.d - t)$, i.e., its remaining execution requirement is strictly greater than the amount of time remaining until its deadline.

A processor is said to be **idle** at time-instant t_0 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 processor is idle at t_0; this is a technical detail that facilitates the definitions of the start and finish of overload, described below.

A system is said to be in an **overloaded state** at time t if the Earliest Deadline algorithm when executed on the system fails to meet the deadline of some task at time t. The **start time** of the overloaded interval is the latest time instant t_s, $t_s \leq t$, at which the processor would be idle if the Earliest Deadline algorithm were executed on the system. The **finish time** of the overloaded interval is the earliest time instant t_f, $t_f \geq t$, at which the processor is idle.

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

Notice that our definitions specify that a task which becomes degenerate remains active until its deadline expires; i.e., no task – degenerate or otherwise – is discarded before its deadline. In the scenario of Example 1, the scheduler that first executes T_1 to completion is *not* permitted to discard T_2 until time 10, even though it is clear after time $2+\epsilon$ that this task will fail to meet its deadline (where ϵ is any arbitrarily small positive number). This has implications upon the definition of an overloaded interval: in the above scenario, the overloaded interval terminates at time 10, and not $2+\epsilon$. We believe this to be quite reasonable, and a reflection of the fact that the "effect" of a task on a system remains until the task has either completed execution or its deadline has expired. A definition of overload that permits one to lessen the size of the overloaded interval by simply choosing to discard certain tasks would, in our opinion, not reflect the reality of very many actual applications.

In the introduction, we presented an intuitive description of EPU. 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

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

where C denotes the set of tasks that successfully complete 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 measured over any overloaded interval and it is this metric that we will be referring to in the remainder of this paper.

3 The ROBUST Algorithm.

In this section, we present ROBUST (Resistance to Overload By Using Slack Time), an on-line scheduling algorithm that guarantees a worst-case EPU of one-half under conditions of overload when all incoming tasks have a slack factor of at least 2. The ROBUST algorithm operates in the following manner during an overloaded interval: It divides the interval into an even number of phases, Phase-1, Phase-2, ..., Phase-$2n$, with the length of Phase-$(2i-1)$ equal to the length of Phase-$2i$ for all i, $1 \leq i \leq n$. (That is, the length of every even-numbered phase is equal to that of the preceding odd-numbered phase.) The length of each phase is determined as discussed below.

Suppose that the overloaded interval begins at time t. Let tasks $T_1^{(1)}, T_2^{(1)}, \ldots, T_{n_1}^{(1)}$ be the set of tasks that are active and non-degenerate at this time, and let task $T_{\max}^{(1)} \in \{T_1^{(1)}, T_2^{(1)}, \ldots, T_{n_1}^{(1)}\}$ be such that $T_{\max}^{(1)}.e \geq T_i^{(1)}.e$ for all i, $1 \leq i \leq n_1$; (i.e., $T_{\max}^{(1)}$ is the most "valuable" task in Phase-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 $[t + e_r^{(1)}, t + 2e_r^{(1)})$.

During Phase-1, the scheduler non-preemptively executes task $T_{\max}^{(1)}$ to completion. Suppose a task T_{new}

makes a request during this phase. Since its slack factor is at least 2, it is guaranteed that this task's deadline is at least twice its execution requirement, i.e., $T_{\text{new}}.d \geq 2T_{\text{new}}.e$. Let $T_{\text{new}}.e$ be greater than $T_{\max}^{(1)}.e$. Since the length of Phase-1 is $e_r^{(1)}$, the scheduler can delay the execution of task T_{new} to after the end of the phase and still meet its deadline. For every task that becomes makes a request during Phase-1, therefore, it is the case that either

- its execution requirement is less than that of task $T_{\max}^{(1)}$, (i.e., it is less "valuable" than $T_{\max}^{(1)}$), or
- it can be successfully scheduled to completion after task $T_{\max}^{(1)}$ has completed execution.

There is therefore no danger of discarding too "valuable" a task during Phase-1.

At the start of Phase-2 (and indeed, every subsequent even-numbered phase), the currently active non-degenerate task with the largest execution requirement is scheduled. For the duration of this phase, whenever a new task makes a request, the scheduler 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, the scheduler preempts the current task and begins executing the new one, otherwise the current task continues execution. If the currently executing task completes execution, the currently most valuable active non-degenerate task is scheduled. At the end of each even-numbered phase Phase-$(2j-2)$, therefore, the processor is executing the currently active non-degenerate task with the largest remaining execution requirement. Let t' be the time when Phase-$(2j-2)$ ends. If the processor is idle at this point in time, then we can conclude that the overloaded interval has terminated. Otherwise, let $T_{\max}^{(j)}$ be the task executing at this instant. Let $e_r^{(j)}$ represent the remaining amount of processor time that is required by task $T_{\max}^{(j)}$. Then, Phase-$(2j-1)$ is defined to be the interval $[t', t' + e_r^{(j)})$, and Phase-$2j$ to be the interval $[t' + e_r^{(j)}, t' + 2e_r^{(j)})$.

At the start of Phase-$(2j-1)$ for all j, $1 \leq j \leq n$, the scheduler commits to executing task $T_{\max}^{(j)}$ to completion, and proceeds to do so for the entire phase. If a new task makes a request during this phase, the condition on its slack factor ensures that either

- its execution requirement is less than that of task $T_{\max}^{(j)}$, or
- it can be successfully scheduled to completion after task $T_{\max}^{(j)}$ has completed execution.

Once again, therefore, there is no danger of discarding too "valuable" a task as a result of committing to non-preemptively execute task $T_{\max}^{(j)}$ during Phase-$(2j-1)$.

(A point to note here is that if at any time there are no more non-degenerate tasks available, the algorithm executes tasks at random from among the degenerate tasks until either a new active non-degenerate task arrives to the system or all the degenerate tasks have been discarded. We leave it to the reader to verify that degenerate tasks can be executed only in an even phase and, given the requirement on task slack factors, that the deadlines of all the existing degenerate tasks will expire within that phase.)

Theorem 1 *The ROBUST algorithm achieves an* EPU *of at least one-half during conditions of overload.*

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

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

□

4 An Upper Bound on EPU

In an environment where all incoming tasks are guaranteed to have a slack-factor no less than two, the RO-

BUST scheduling algorithm obtains an *EPU* of at least one-half even under overloaded conditions. We now address the issue of *optimality:* Is the ROBUST algorithm optimal? That is, is it the case that no on-line scheduling algorithm can obtain an *EPU* greater than one-half in such an environment? We do not yet have a conclusive answer to this question. However, we prove in this section that no on-line scheduling algorithm can guarantee an *EPU* greater than five-eighths under conditions of overload in the above framework. This means that even if the bound of 5/8 were to be tight, the ROBUST algorithm is at most 20 percent worse than the optimal, since $\frac{(1/2)}{(5/8)} = 0.8$.

Theorem 2 *No on-line scheduling algorithm can guarantee an* EPU *greater than five-eighths under conditions of overload in an environment where all incoming tasks have slack-factor of at least two.*

Proof. The proof is by means of an adversary argument that consists of pitting any on-line algorithm against a (hypothetical) malicious adversary that generates a sequence of tasks, observes the behavior of the on-line algorithm on these tasks, and then extends the sequence with the explicit purpose of minimizing the *EPU* of the on-line algorithm. At time $t = 0$, the adversary generates two identical tasks T_o and R_o with $T_o.e = R_o.e = x_o$, and $T_o.d = R_o.d = 2x_o$. Just before the deadline of these tasks (i.e., just before time $T_o.d$) the adversary generates identical tasks T_1 and R_1 with $T_1.e = R_1.e = x_1$ and $T_1.d = R_1.d = 2x_1$. The "best" situation for the on-line algorithm to be in is for it to have completed the execution of task T_o, and to be currently engaged in executing R_o. In general, the on-line algorithm will have executed task T_i, and be executing R_i; just before the deadline of R_i, the adversary generates two new identical tasks T_{i+1} and R_{i+1} with $T_{i+1}.e = R_{i+1}.e = x_{i+1}$, and $T_{i+1}.d = R_{i+1}.d = 2x_{i+1}$. The on-line algorithm now has a choice

- discard R_i and begin executing T_{i+1}, in which case the adversary again generates two tasks T_{i+2} and R_{i+2} just before the deadline of task R_{i+1} (by which time the on-line algorithm would have completed the execution of task T_{i+1}, and will be executing R_{i+1}), or
- continue the execution of task R_i, in which case no further tasks are generated by the adversary; the on-line algorithm will get to complete the execution of R_i and exactly one of T_{i+1} or R_{i+1}.

Now this process could go on for ever if the on-line algorithm always chooses to discard R_i in favor of T_{i+1} every time such a choice is offered. However, recall that in our model, overloaded conditions correspond to emergencies, and emergencies are assumed to be finite. There is, therefore, a fixed integer m such that, if the on-line algorithm is executing task R_{m-1} and the adversary generates tasks T_m and R_m, then irrespective of the scheduling decision made by the on-line algorithm, the adversary will not generate any further tasks.

We leave it to the reader to verify that when the interaction between the on-line algorithm and the adversary has ceased, the on-line algorithm has completed the execution of all of the T_i tasks that were generated, and exactly one of the R_j tasks.

- If the task R_j that was executed to completion is in fact R_m, then the length of the overloaded interval is $\sum_{i=0}^{m}(T_i.d)$, which is equal to $\sum_{i=0}^{m} 2x_i$. The *EPU* over this interval is therefore $(x_m + \sum_{i=0}^{m} x_i)/(\sum_{i=0}^{m} 2x_i)$.
- If the task R_j that was executed to completion is not T_m, then the length of the overloaded interval is $\sum_{i=0}^{j+1}(T_i.d)$, which is equal to $\sum_{i=0}^{j+1} 2x_i$. The *EPU* over this interval is therefore $(x_j + \sum_{i=0}^{j+1} x_i)/(\sum_{i=0}^{j+1} 2x_i)$.

To complete our proof, we need to demonstrate the existence of a series of numbers $x_o, x_1, \ldots, x_i, \ldots, x_m$ such that the *EPU* in both cases above is at most 5/8. This is done in Lemma 1 in Appendix A. We have thus shown that, against an adversary that behaves as described here, and generates tasks with computation requirements and deadlines as dictated by the sequence defined in Lemma 1, no on-line scheduling algorithm can obtain an *EPU* greater than five-eighths.

□

5 Task Sets With Arbitrary Minimum Slack Factors

Thus far, our attention was restricted to the study of environments where all tasks were guaranteed to have a slack factor of at least two. We now extend our analysis to include arbitrary minimal slack factor guarantees, and show how such guarantees may be used to reduce the performance degradation under overload conditions.

5.1 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-numbered phase Phase-$2i$ is set to $1/(f-1)$ times the length of the preceding odd-numbered phase Phase-$(2i-1)$. We leave it to the reader to verify that, as before, the processor is "useful", (i.e., executing tasks that do complete by their deadlines) during all the odd-numbered phases, yielding the following theorem:

Theorem 3 *The Generalized ROBUST algorithm achieves an* EPU *of at least* $\frac{f-1}{f}$ *during conditions of overload.*

Henceforth, when we refer to the ROBUST algorithm, we will mean the generalized algorithm described here.

5.2 Upper Bound on EPU

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

Theorem 4 *No on-line scheduling algorithm can guarantee an* EPU *greater than* $\frac{\lceil f \rceil}{\lceil f \rceil + 1}$ *under conditions of 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 such that $\lceil f \rceil$ of them have $a = 0$, $e = 1$, and $d = f$, and the remaining $\lceil f \rceil$ tasks are identical except that they have their request times $a = 1 - \epsilon$ where $0 < \epsilon < 1$. 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 $\frac{\lceil f \rceil}{\lceil f \rceil + 1}$ cannot be obtained on the overloaded interval $[0, \lceil f \rceil + 1 - \epsilon)$

□

An important point to note here is that the above theorem establishes a "quick-and-dirty" upper bound on the best *EPU* obtainable. This bound is clearly not tight — compare the bounds established by Theorem 2 and the above Theorem for $f = 2$. However, it does establish that the Generalized ROBUST algorithm provides a performance that is at most

$$\begin{aligned} & 1 - \frac{(f-1)/f}{\lceil f \rceil/(\lceil f \rceil + 1)} \\ \geq \; & 1 - \frac{(f-1)/f}{(f+1)/(f+2)} \\ = \; & \frac{2}{f(f+2)} \end{aligned}$$

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 ten percent of the optimal for all slack factors of at least 4 (i.e., $\frac{2}{f(f+2)} \leq 0.1$ for all $f \geq 4$). Furthermore, depending on the looseness of the above *EPU* upper bound, the ROBUST algorithm may turn out to be within ten 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 a reasonably efficient solution to address the problem of performance degradation under overload by using task slack times. While this seems true in general, we note, however, that it is not the case for a small range of slack factors, as discussed below.

For $f = 4/3$, the *EPU* achieved by the ROBUST algorithm is 1/4. However, the same *EPU* is guaranteed by the algorithm described in [1] without making assumptions about task slack time. Therefore, with task

slack factor that is no more than 4/3 times as fast as the original, there is no benefit in using the ROBUST algorithm. When tasks have a slack factor of greater than 4/3, however, the algorithm may be productively used to improve the performance guarantees of the system.

6 Overload Tolerance

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 performance when not overloaded. Overload tolerance seems a reasonable property to require safety-critical systems to satisfy. Recall that, in our model, overloaded conditions are brought about by the onset of emergencies. Ideally, one would like a system to enhance its performance upon the onset of an emergency in order to better deal with the emergency. If this is not possible, however, we would nevertheless like the system to continue to provide the level of performance that was provided before the emergency occurred.

The 1/4 bound on *EPU* under conditions of overload mentioned in the Introduction implies that no online 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 [4].) 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 contrast, we propose a method here that abstracts away from individual applications and provides a mechanism for achieving overload tolerance irrespective of the characteristics of any particular application. Such an approach has the advantage of being widely applicable over a large number of systems, and, since the property of overload tolerance is guaranteed by the *mechanism,* requires no additional design effort for each application system once the mechanism has been designed and proven correct, as described below.

Recall that a task in our model is characterized by an arrival time $T.a$, an execution requirement $T.e$, and a (relative) deadline $T.d$. A task that has any chance at all of completing by its deadline has its relative deadline $T.d$ at least as large as its execution requirement $T.e$; i.e., it has slack factor $T.d/T.e \geq 1$. If this task is executed on hardware that is f times as fast as the hardware for which it had been specified, the execution requirement of the task is reduced to $\frac{T.e}{f}$ – its 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. However, 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 3, 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 Figure 1, where performance is plotted against system load, with both axes labelled to percentages of the capacity of the original system. The beaded line profiles the behavior of the orignal system, and the solid line the behavior of the system installed on hardware twice as fast as the initial hardware, with the ROBUST algorithm used for scheduling during overload. Notice that *the performance of the new (doubly fast) 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

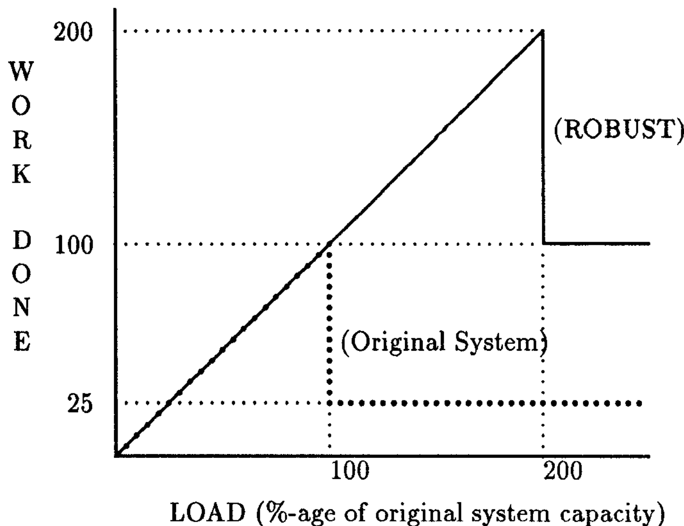

Figure 1: The effect of load on system behavior

to double the speed of the processor and use the ROBUST scheduler.

A word of caution, however: In the above analysis, it was essential that the use of hardware f times as fast as the original result in a decrease in execution requirements of *all* tasks by a factor of f. This is clearly not the case for many actual systems. For example, a task whose execution time is dependent upon factors external to the system would be unaffected by any speedup in system hardware. And in certain circumstances, even if the use of hardware f times as fast does result in slack factors $\geq f$ for all tasks, it may be that hardware twice as fast as the original is simply not available. In either of these cases, this approach to guaranteeing overload tolerance would, of course, fail, and the other, more application specific, method must be attempted.

Using faster hardware is not the only source for obtaining a larger slack factor. An alternative situation where the same effect is obtained is when the designer is willing to relax the deadlines (or scale down the execution requirements) of all the tasks. While overload tolerance can no longer be guaranteed, the ROBUST algorithm is equally applicable in such scenarios. If, for example, the user is willing to double all the task deadlines, then the ROBUST algorithm guarantees that the performance of the system will not degrade by more than a factor of two during overload.

7 Conclusions.

It has previously been shown [1] that no on-line uniprocessor scheduling algorithm can guarantee an EPU greater than 25% under conditions of overload for arbitrary task sets. We designed ROBUST (Resistance to Overload By Using Slack Time), an on-line scheduling algorithm that is not limited by the 25% bound of [1] (Theorems 1 and 3) for task sets that guarantee a minimum slack factor for every task. We described 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. We also showed that it is guaranteed to be within ten percent of the optimal for slack factors greater than 4.

The scheduling algorithms presented in this paper require the slack factor of <u>all</u> 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 major design goal for the developers of safety-critical real-time systems.

A number of problems remain open. First, neither the specific ROBUST algorithm presented in Section 3, nor its generalization discussed in Section 5, have been proven optimal. In addition, the systems we considered here operate with two sets of on-line schedulers – one for use under normal circumstances, and the other for use during emergencies. We assumed that an application "knows" when an emergency occurs, and switches schedulers accordingly upon the onset of overload. It is possible, however, that a system may be unaware that an emergency is occurring until it is actually well into the emergency. Therefore, we need one integrated algorithm that combines the optimal behaviors of the two separate algorithms. Such an integrated algorithm is presented in [5] for systems that are implemented on the hardware for which they were designed. The design of similar integrated algorithms for systems that are im-

plemented on faster hardware appears to be a fruitful research area.

Acknowledgements.

We are grateful to the anonymous referees for their very useful suggestions — an attempt has been made to incorporate most of these suggestions into this document.

The proof in Section 4 is similar to a proof that first appeared in [2], and subsequently in [1].

Appendix

A Lemma for Section 4.

Lemma 1 *The series of numbers*

$$\begin{aligned} x_o &= 1 \\ x_1 &= 3 \\ x_i &= 4(x_{i-1} - x_{i-2}), \quad i \geq 2 \end{aligned}$$

satisfies the following two properties:

Property 1. *For all* $j \geq 0$,

$$\frac{x_j + \sum_{i=0}^{j+1} x_i}{\sum_{i=0}^{j+1} 2x_i} = 5/8$$

Property 2. *For some* $m > 0$,

$$\frac{x_m + \sum_{i=0}^{m} x_i}{\sum_{i=0}^{m} 2x_i} \leq 5/8$$

Proof.
Property 1. Using standard techniques of algebraic manipulation, we first reduce Property 1 to a simpler form:

$$\begin{aligned} [\frac{x_j + \sum_{i=0}^{j+1} x_i}{2(\sum_{i=0}^{j+1} x_i)} &= \frac{5}{8}] \\ &\equiv [\frac{8}{5}x_j + \frac{8}{5}(\sum_{i=0}^{j} x_i) + \frac{8}{5}x_{j+1} \\ &\qquad = 2(\sum_{i=0}^{j} x_i) + 2x_{j+1}] \\ &\equiv [\frac{8}{5}x_j - (2 - \frac{8}{5})(\sum_{i=0}^{j} x_i) \\ &\qquad = (2 - \frac{8}{5})x_{j+1}] \\ &\equiv [x_{j+1} = \frac{8/5}{2 - 8/5}x_j - \sum_{i=0}^{j} x_i] \\ &\equiv [x_{j+1} = 4x_j - \sum_{i=0}^{j} x_i] \end{aligned}$$

We have thus shown that Property 1 above is equivalent to

$$x_{j+1} = 4x_j - \sum_{i=0}^{j} x_i. \tag{1}$$

The reader may verify by substitution in Equation (1) that the recurrence in the statement of the lemma satisfies Property (1) for $j = 0$.

From Equation (1), it follows that

$$x_{j+2} = 4x_{j+1} - \sum_{i=0}^{j+1} x_i. \tag{2}$$

Subtracting Equation (1) from Equation (2), we obtain

$$\begin{aligned} & x_{j+2} - x_{j+1} = 4x_{j+1} - 4x_j - x_{j+1} \\ \equiv \quad & x_{j+2} = 4(x_{j+1} - x_j). \end{aligned}$$

Notice that this is exactly the form of the recurrence relation. We have thus proved that, for $j > 0$, Property (1) is merely a re-statement of the recurrence relation. The recurrence therefore satisfies Property (1) for $j > 0$ as well.

Property (2). It has been proven elsewhere (see [1]) that the recurrence in the statement of this lemma satisfies the following property[1]:

$$\frac{x_m}{(\sum_{i=0}^{m} x_i)} \leq \frac{1}{4} \text{ for some } m > 0 \tag{3}$$

[1]Strictly speaking, this is not true. The RHS of this inequality should be $\frac{1}{4} + \epsilon$ for ϵ an arbitrarily small positive real number. As a result, the quantity 5/8 in the RHS of Properties (1) and (2) should be replaced with $\frac{5}{8} + \epsilon$. Since the Lemma is true for ϵ arbitrarily small, we have chosen to keep things simple and gloss over this minor mathematical detail.

We will use this property to prove that the LHS of Property (2) is at most 5/8 whenever the LHS of (3) is no larger than 1/4, therefore proving that the recurrence satisfies Property (2).
The LHS of Property (2) =

$$\begin{aligned}
& \frac{x_m + \sum_{i=0}^{m} x_i}{2\sum_{i=0}^{m} x_i} \\
\leq & \frac{\frac{1}{4}\sum_{i=0}^{m} x_i + \sum_{i=0}^{m} x_i}{2\sum_{i=0}^{m} x_i} \quad \text{(Using Property 3)} \\
= & \frac{1/4 + 1}{2} \\
= & \frac{5}{8}
\end{aligned}$$

□

References

[1] 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. In *Proceedings of the 12th Real-Time Systems Symposium*, San Antonio, Texas, December 1991. IEEE Computer Society Press.

[2] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, and D. Shasha. On-line scheduling in the presence of overload. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, San Juan, Puerto Rico, October 1991. IEEE Computer Society Press.

[3] M. Dertouzos. Control robotics : the procedural control of physical processors. In *Proceedings of the IFIP Congress*, pages 807–813, 1974.

[4] M. Garey and D. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*, W. H. Freeman and Company, New York, 1979.

[5] G. Koren and D. Shasha. D^{over}: An optimal on-line scheduling algorithm for overloaded real-time systems. Technical Report TR 594, Computer Science Department, New York University, 1992.

[6] A. Mok. Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment. Ph.D. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, May 1983.