TECHNICAL
RESEARCH
REPORT

S Y S T E M S
R E S E A R C H
C E N T E R

*Supported by the
National Science Foundation
Engineering Research Center
Program (NSFD CD 8803012),
Industry and the University*

ROBUST: A Hardware Solution to Real-Time Overload

by S. Baruah and J.R. Haritsa

TR 92-56

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

ROBUST: A Hardware Solution to Real-Time Overload

Sanjoy Baruah* Jayant R. Haritsa†

Abstract

It has been proven that no on-line scheduling algorithm can guarantee a processor utilization greater than 25% under conditions of overload. From this result, it follows that there is no satisfactory software solution (within the paradigm of deterministic computation) to the problem of constructing overload-tolerant real-time systems, and we are forced to consider alternative remedies. In this paper, we describe an attempt at a hardware solution. In particular, we quantify the advantages of using *faster* hardware in real-time systems. An obvious advantage is that more work can be done in a shorter amount of time and the system is therefore less likely to enter overload. More interestingly, we prove that the performance of systems using faster hardware improves dramatically over systems using slower hardware if overload *does* occur. We present here a new scheduler, **ROBUST**, which efficiently uses faster hardware to prevent performance degradation under overload conditions.

*Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712-1188. (Supported in part by a research grant from the Office of Naval Research under contract number N00014-89-J-1472.)

†Systems Research Center, University of Maryland, College Park, Maryland 20742. (Supported in part by a Systems Research Center Post-Doctoral Fellowship.)

1 Introduction.

Real-time computer systems are systems where the correctness of a computation depends upon both the result of the computation and the *time* at which this result was produced. These systems have become increasingly important over the past few years. We all come across numerous real-time application systems in our day-to-day lives; these applications span the spectrum from air-traffic-control systems to power-plant monitoring to laser delivery systems for eye surgery.

We consider here real-time application systems that are extremely safety critical, e.g., the monitoring and control of nuclear power plants. Typically, when such systems are designed, the system designers attempt to anticipate every eventuality and incorporate it into the design of the system. Under ideal circumstances, such a system would never become overloaded, and the behavior of the system would be as expected by the system designers. Unfortunately, real-time systems need to interact with the real world rather than an ideal one. The best-laid plans of system designers do sometimes go awry: unanticipated situations arise, the unexpected occurs, and it may happen that the required amount of processor time exceeds the system capacity. The system is then said to be in *overload*. If this happens, it is extremely 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.

One 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. (We make the reasonable assumption that our scheduling algorithms use no inserted idle times; i.e., the processor is never idle while there are tasks that can be executed on it. We require that any interval used to compute the *EPU* include no idle time, and that it encompass the deadlines of all tasks that have received non-zero service in the interval. The *EPU* of a *system* is the lowest *EPU*, measured over any interval of time, of the system.) Consider, as an example, 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 2 for 8 units of processor time by a deadline of 10. A scheduler that schedules T_1 to completion has an *EPU* of 0.3 over $[0, 10)$, while one that schedules T_2 to completion has an *EPU* of 0.8 over $[0, 10)$. Clearly, no scheduler can schedule both T_1 and T_2 to completion.

Certain uniprocessor on-line scheduling algorithms such as the Deadline algorithm [3], for example, are known to be optimal under non-overloaded conditions. These algorithms can therefore guarantee an *EPU* of 100% under conditions of non-overload. Furthermore, 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. 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. This result, along with the optimality of the Deadline algorithm, implies that the onset of an emergency may force a deterioration in system performance by a factor of four. In this paper, we attempt to overcome this fall in performance by using faster hardware. The major question that we address is: Is it necessary to obtain hardware that is be at least four times as fast as the original hardware in order to compensate for the four-fold loss in performance? The answer, fortunately, is "no".

Our task model is as follows: 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 $T.a + T.d$. 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. For any task T, we define its **slack factor** 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.

Now, suppose that we were provided with hardware that is *twice* as fast as the hardware for which the specifications of some system had been written, and that we wanted to transfer the system onto this new hardware. Let task T' in the new system correspond to some task T in the original one. It is easy to see that the new task T' will have $T'.a = T.a$, $T'.e = T.e/2$, and $T'.d = T.d$. All tasks T' in the new system will therefore have slack factor $= T'.d/T'.e = T.d/(T.e/2) = 2 \times T.d/T.e \geq 2$.

In this paper, we present **ROBUST** (Resistance to Overload By Using Slack Time), an on-line scheduling algorithm that is guaranteed to achieve an effective processor utilization of at least one half under conditions of overload, provided all incoming tasks have a slack factor of at least 2. Figure 1 shows how the ROBUST scheduler can be used in conjunction with faster hardware to develop a satisfactory solution to the overload problem. In this figure, we have plotted the "work done" by the system against the incoming load. Both axes are labeled to percentages of the capacity of the original system. The beaded line profiles the behavior of the original system, while the full

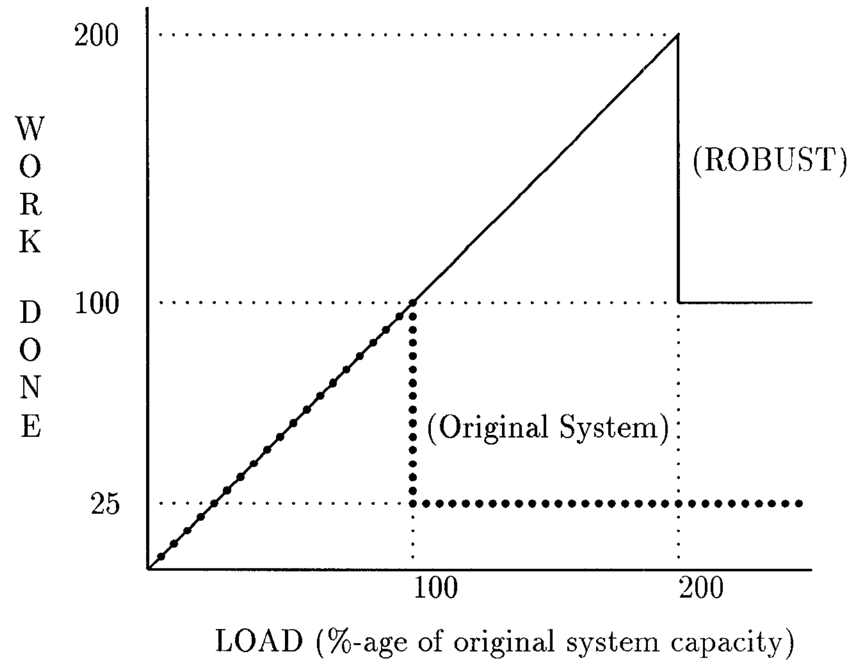

Figure 1: The effect of load on system behavior

line shows the performance of the ROBUST scheduler. Focusing our attention on the original system, it is clear that for loads no larger than its capacity, the existence of optimal algorithms under conditions of non-overload ensures that the work done by the system increases linearly with the load. For loads greater than the capacity of the system, however, it has been shown [2, 1] that a performance degradation by a factor of four is unavoidable. Turning our attention to the faster system, where the hardware is twice as fast as that of the original system, we observe that the performance behavioral profile changes in two ways: First, loads between 100% and 200% of the original system's capacity no longer correspond to overload; the work done by the system therefore increases linearly with the load in this region. Second, if the load does increase to beyond 200% of the original system capacity, the ROBUST scheduler ensures that the performance does not degrade by more than a factor of two, to the 100% line. The new system is thus more "overload-tolerant" in that it can absorb an overload of up to twice the capacity of the original system, and, even under more extreme overload, the performance never degrades below the original system's capacity. While doubling processor speeds to handle overload may not be considered cost-effective in conventional computer systems, it is quite reasonable, however, in the real-time domain. This directly relates to the application domain of real-time systems where functionality, rather than cost, is usually the driving consideration.

The remainder of this paper is organized in the following fashion: In Section 2, we prove that our new algorithm, ROBUST, guarantees a worst-case *EPU* of one-half under conditions of overload when all incoming tasks have a slack factor at least 2. In Section 3, 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 succeed in proving an upper bound of 5/8'ths on the performance of *any* on-line algorithm in the above situation. Then, in Section 4, we generalize the above analysis to the case where the processor speedup factor is an arbitrary number. Finally, Section 5 concludes with a summary of the results presented herein, and outlines future research directions.

2 The ROBUST On-Line 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[1] 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 new task T_{new} arrives during this phase. Since its slack factor must be at least 2, it is guaranteed that this new 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} after to the end of the phase and still meet its deadline. For every new task that arrives during Phase-1, therefore, it is the case that either

- its computation time 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

[1] A task T is *active* at time t if (i) $T.a \leq t$; i.e., the task arrives before t, (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) $T.e_r \leq (T.d - t)$, i.e., the task can still complete before its deadline — it has not yet become degenerate.

active task with the largest execution requirement is scheduled. For the duration of this phase, whenever a new task arrives, the scheduler compares the execution requirement of the newly arrived 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 task is scheduled. At the end of each even-numbered phase Phase-$(2j-2)$, therefore, the processor is executing the currently active task with the largest execution requirement. Let t' be the time when Phase-$(2j-2)$ ends, and 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 arrives during this phase, the condition on its slack factor ensures that either

- its computation time is less than that of task $T_{\max}^{(j)}$, or
- it can be successfully scheduled to completion <u>after</u> 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-$(2j-1)$.

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

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

□

3 An Upper Bound on EPU

In an environment where all incoming tasks are guaranteed to have a slack-factor no less than two, the ROBUST 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 <u>not</u> 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 2 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 2, no on-line scheduling algorithm can obtain an *EPU* greater than five-eighths.

$\square$

4 Processors With Arbitrary Speedups.

The ROBUST scheduling algorithm guarantees an *EPU* of one-half under conditions of overload if provided with hardware that is twice as fast as the hardware for which a particular real-time system has been designed. In the previous sections, we discussed how this algorithm can be used to construct a new system that is more overload-tolerant than the original one in that (i) it can absorb an overload of up to twice the capacity of the original system, and (ii) even under more extreme overload, its performance never degrades to below the original system's capacity.

In the above analysis, our focus on a processor speedup of two was primarily to highlight the performance potential of the ROBUST algorithm and also to demonstrate that this speedup was sufficient to maintain performance equal to the original system's capacity even under overload. We now go on, in this section, to demonstrate that the ROBUST algorithm is applicable over a wide range of processor speedup factors. Our analysis of the previous sections is extended to include arbitrary processor speedups, and we show how the hardware speedup may be used to minimize the performance degradation of overloaded real-time systems.

Lemma 1 *Consider a real-time system designed for some particular hardware. If the system is implemented on hardware that is f times as fast, $f > 1$, then all tasks in the implementation will have slack factor $\geq f$.*

Proof. Let task T' in the implementation correspond to some non-degenerate task T in the specification. It is easy to see that the new task T' will have $T'.a = T.a$, $T'.e = T.e/f$, and $T'.d = T.d$. All tasks T' in the new system will therefore have slack factor $= T'.d/T'.e = T.d/(T.e/f) = f \times T.d/T.e \geq f$.

□

4.1 The Generalized ROBUST Algorithm

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.*

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

4.2 Upper Bound on EPU

The following theorem establishes an upper bound on the EPU that is attainable for arbitrary hardware speedups.

Theorem 4 *No on-line scheduling algorithm can guarantee an EPU greater than* $\frac{f}{f+1}$ *under conditions of overload in an environment where all incoming tasks have slack-factor of at least* f.

Proof. Construct a set of $f+1$ tasks such that f of them have $a=0$, $e=1$, and $d=f$, while the remaining task is identical except that its $d=f-\epsilon$ where $0<\epsilon<f$. It is simple to see that while

it is straightforward to successfully schedule f tasks, no on-line scheduler can manage to schedule all $f+1$ tasks. This means that an EPU greater than $\frac{f}{f+1}$ cannot be obtained.

□

An important point to note here is that the above theorem establishes a "quick-and-dirty" upper bound since the bound is clearly not tight (compare the bounds established by Theorem 2 and the above Theorem for $f = 2$). However, the point we wish to make is that even if the bound *were* tight, the Generalized ROBUST algorithm provides a performance that is at most

$$1 - \frac{(f-1)/f}{f/(f+1)} = \frac{1}{f^2}$$

fractionally off from the optimal. Therefore, 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 speedup factors greater than 3.2. Further, depending on the looseness of the above EPU upper bound, the ROBUST algorithm may be within ten percent of the optimal at considerably lower slack factors. In fact, it is even possible that the algorithm may itself be optimal — we are currently researching this issue. In summary, the ROBUST scheduler in conjunction with faster hardware appears to provide a reasonably efficient solution to address the problem of performance degradation under overload. While this seems true in general, we note, however, that it is not the case for a small range of speedup 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 the use of faster hardware. Therefore, with hardware that is no more than $4/3$ times as fast as the original, it is not worthwhile to use the

ROBUST algorithm. When hardware with a speed-up of greater than 33% is available, however, the algorithm may be productively used to improve the performance guarantees of the system.

We now summarize the performance benefits with respect to overload of hardware that is f times as fast as the original hardware:

- First, the system is less likely to go into overload, since its "capacity" is greater. Any load that is less than f times the capacity of the original system will not push the system into overload.

- Second, when the load on the system is extreme, overload will occur. However, if the ROBUST algorithm is used to schedule tasks during the overloaded time periods, the resultant EPU is guaranteed to be always as much as $(f - 1)$ times the capacity of the original system.

5 Conclusions.

It has previously been shown [1] that no on-line scheduling algorithm can guarantee an EPU greater than 25% under conditions of overload. It immediately follows from this result that there can be no satisfactory software solution to the problem of constructing overload-tolerant real-time systems. In this paper, we have attempted to develop a hardware solution to handle real-time overload.

We characterized the inflationary effect of faster hardware on the *slack factor* of tasks (Lemma 1), and exploited this effect to design 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). 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 hardware speedup. We also showed that it is guaranteed to be within ten percent of the optimal for hardware that is slightly more than thrice as fast as the original hardware.

Using faster hardware is not the only source of obtaining a larger slack factor. An alternative situation where the same effect is obtained is where the designer is willing to relax the deadlines (or scale down the execution requirements) of all the tasks. 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.

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 major design goal for the developers of overload-tolerant real-time systems.

A number of problems remain open. First, neither the specific ROBUST algorithm presented in Section 2, nor its generalization discussed in Section 3, have been proven optimal. A more important open question concerns our assumption that the use of hardware f times as fast as the original results in a decrease in execution time of all tasks by a factor of f. In many practical systems, this is clearly not the case. For example, a task whose execution time is dependent upon factors external to the system would be unaffected by any speedup in system hardware. We plan to investigate a model where there are several types of tasks, and the effect of faster hardware on a task depends upon its type.

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 a real-time system "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 [4] for systems that are implemented on the hardware for which they were designed. The design of similar integrated algorithms for systems that are implemented on faster hardware appears to be a fruitful research area.

Acknowledgements.

The proof in Section 3 is similar to a proof that first appeared in [2], and subsequently in [1].

Appendix

A Lemma for Section 3.

Lemma 2 *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

$$
\begin{aligned}
\textbf{Property 1.}\ (x_j + \sum_{i=0}^{j+1} x_i)/(\sum_{i=0}^{j+1} 2x_i) &= 5/8 \textit{ for all } j \geq 0 \qquad (1) \\
\textbf{Property 2.}\ (x_m + \sum_{i=0}^{m} x_i)/(\sum_{i=0}^{m} 2x_i) &\leq 5/8 \textit{ for some } m > 0 \qquad (2)
\end{aligned}
$$

Proof.

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

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

We have thus shown that Property (1) above is equivalent to

$$x_{j+1} = 4x_j - \sum_{i=0}^{j} x_i. \tag{3}$$

The reader may verify by substitution in Equation (3) that the recurrence in the statement of the lemma satisfies Property (1) for $j = 0$.

From Equation (3), it follows that

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

Subtracting Equation (3) from Equation (4), we obtain

$$x_{j+2} - x_{j+1} = 4x_{j+1} - 4x_j - x_{j+1}$$

$$\equiv$$

$$x_{j+2} = 4(x_{j+1} - x_j).$$

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[2]:

$$\frac{x_m}{(\sum_{i=0}^{m} x_i)} \leq \frac{1}{4} \text{ for some } m > 0 \tag{5}$$

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

[2]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.

The LHS of Property (2) =

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

□

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] 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.