# Surge Handling as a Measure of Real-Time System Dependability

Zahava Koren, Israel Koren and C. M. Krishna

Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003

**Abstract.** Traditional reliability measures for computer systems can be classified into *Computer-Centric* or *Application-Centric* categories. The former concentrate on the hardware resources while ignoring the application's needs. The latter focus on the requirements of a specific application which is being executed, thus requiring the knowledge of all the details of the application; information which may not always be readily available. Also, the narrow view on the system's reliability through a single application is too restrictive and provides very limited information regarding the way the system will handle other applications.

In this paper we present new measures for real-time system reliability. These measures are application-*sensitive* rather than application-*centric*, and are especially suitable for systems executing various applications with different attributes, some of which may not be known in advance.

Our proposed measures capture the capability of a real-time system to respond successfully to unexpected surges in the workload. These surges may result from a phase change in the system's mission, an application-related emergency situation or the failure of some system resources. The ability of the system to handle such surges determines, to a large extent, its chances of survival and meeting its applications' deadlines.

## 1 Introduction

In this paper, we discuss the merits of using the surge-handling capability of a real-time embedded system as a measure of its reliability. Such systems are increasingly used to control life-critical processes such as fly-by-wire aircraft, nuclear reactors, etc. Reliability in a real-time system is determined by the ability of the system to meet such task deadlines as is necessary for the correct functioning of the controlled process.

The problem with conventional approaches to evaluating reliability is that they treat reliability as a static, not a dynamic, quantity. Conventional approaches assume that failure is caused exclusively by the failure of individual hardware or software components. This approach to reliability ignores the dynamic component; namely, that the computer system is only reliable if it does not cause failure for the application (irrespective of the actual number of processors which are still operational).

Central to any reliability evaluation is the definition of an appropriate set of reliability measures. Measures are yardsticks by which reliability is expressed. In other words, reliability measures act as filters, imposing a scale of values that determines which factors are important for "reliability" and which are not.

Currently-used reliability measures for fault-tolerant systems fall in one of two categories.

- *Computer-Centric:* All traditional reliability measures are computer-centric. They focus on the computer in isolation to anything else. The system is defined as being in one of several states, and Markov models are usually used to model the transitions from one state to another. A subset of the states is defined as the failed states, and unreliability is the probability that the system, over a given period of operation, enters one of these states. In traditional models, failed states are entered either after a large number of hardware failures have occurred, leaving the system with insufficient computational capacity, or when there is "coverage failure," i.e., a failure that goes uncaught and uncorrected, and which causes the overall failure of the system. Examples include traditional reliability, availability, and throughput, together with variations such as performance-related reliability measures [1].
- *Application-Centric:* An application-centric measure starts with the premise that performance and reliability can only be meaningfully defined within the context of the application. An application-centric measure would start by considering what the computer needs to do in order to meet the needs of the application. The ability of the computer to meet the application's needs is the application-centric measure of the computer. We will provide some examples in the next section.

Performance measures are used in two related ways. One is to allow a comparison of multiple designs or systems for a given application. The second is to provide an interface by which the designer of the controlled process in which the computer is to be embedded (e.g., aircraft, spacecraft, etc.) can communicate to the computer designer the needs of the application in a form that is intelligible to the latter.

The first use of performance measures is fairly obvious, and is indeed the standard one; the second needs some elaboration. Computer engineers are not trained in control system terminology, and require the needs of the application to be translated to them in terms that they can understand. A good measure of real-time performance or dependability should express the control-theoretic needs of the application in a way that is meaningful to the computer designer. As an example, imagine that there is some cost function by which the impact on the application of a given computer response time (for each task) can be quantified. We shall see an example of this in Section 2. In defining the cost function associated with each control task, the control engineer quantifies the relationship between the response time for each task and the consequent performance of the controlled process, The computer engineer, upon receipt of this information, does not have to be concerned about the control-theoretic foundations of the connection between the controlled process and the embedded computer; the cost functions (and associated hard deadlines) of the various tasks are all that are required.

While application-centric measures have obvious advantages, they have two related disadvantages. The first is that one requires very specific information about the application in order to compute these measures. At an early stage of design, this information may not be available. The second disadvantage is that, by their very nature, application-centric measures are very application-specific. They express the capabilities of the computer entirely with respect to a given application. One cannot directly infer from this the generic ability of a computer to perform in other real-time applications.

The need therefore exists for real-time measures that straddle the middle ground between the computer-centric and the application-centric measures. Such measures should be gracefully degrading with respect to information. That is, as more information is made available about the application, they should become more application-specific. In the absence of any information about the application, they should express the attributes of the computer that render it capable of running real-time applications. As the amount of information increases, they should become increasingly focused on expressing how

the capabilities of the computer meet the particular needs of that application.

In this paper we suggest such a measure, namely, the ability of a computer to handle load surges. We claim that this ability can be used as an indicator of the quality of the following:

- Task assignment and scheduling algorithms.
- Ability of the system to handle a load surge following either an emergency condition or a phase change in the application's mission.
- Procedure followed for the reconfiguration of the hardware upon a failure occurrence.
- Procedure for reassigning and rescheduling tasks upon the failure of one or more processors.

This paper is organized as follows. In Section 2, we provide a brief literature survey of application-centric performance measures. This is followed in Section 3 by a description of the proposed surge-handling measures. Illustrative examples are provided in Section 4, and Section 5 contains some discussion and conclusions regarding the new measures.

## 2 Prior Work

Not much work has been reported on measures specifically meant for real-time systems. For a survey, see [3, 4]. We describe here two efforts in that direction.

### 2.1 Performability

This measure was introduced by Meyer [7, 8]. The application is defined as having a set of *accomplishment levels*, which are levels of performance which can be distinguished from one another by the user. We list what the computer must do in order to allow the application to meet each of these accomplishment levels, $A_1, A_2, \cdots, A_n$. The computer can then be modeled to find the probability $P_i$ that it can perform at a level that allows the application to deliver accomplishment level $A_i$. The vector of probabilities, $P_1, P_2, \cdots, P_n$, is the performability of the computer. See [8] for a detailed example.

### 2.2 Cost Functions and Probability of Dynamic Failure

This measure was introduced by Krishna and Shin [2, 9, 10, 11], and was designed explicitly for use in embedded systems which handle industrial or other control processes. In such systems, the computer is in the feedback loop of the controlled process, and the computer response time is a component of the feedback loop delay. From elementary control theory, we know that such delays in the feedback loop have a detrimental effect on the quality of the control provided. This can be quantified and used to measure the level of performance provided by the computer. More precisely, we can identify hard deadlines for the tasks by calculating the response time delays that lead to a loss of stability of the controlled process. Even if deadlines are not missed, the cost of having a certain response time can be computed by quantifying the extent to which the performance of the controlled process has degraded. See [4] for further details.

## 3 The Measures – Definitions

We claim that the survival of a real-time system depends, to a large extent, on its ability to successfully handle unexpected surges in the workload. Such a surge can arise from any of the following causes:

- A change in mission phase, where one set of tasks is replaced by another set. During the period of transition, there can be a surge in the workload, with new tasks having to be run before all of the old tasks have been completed.
- An emergency situation that requires additional tasks to be run. For example, one may have the onset of instability being detected in a vehicle, following which certain additional work may have to be executed in order to restore stability.
- Failure of one or more processors. This affects the system in two ways: first, the workload which was previously run on the failed processor has now to be remapped onto functional processors. This remapping (which includes deciding which tasks to move to which processors, moving the tasks appropriately, and aligning the memories) imposes a transient load on the system. Secondly, the first iteration of such tasks following a move can have significantly reduced laxity, which appears to the surviving processors as a surge in the workload.

We introduce in this paper two new measures which express the ability of the system to respond to such a surge in the workload. For the purposes of this paper, a surge is defined as an additional workload that is suddenly imposed on the system. A surge may consist of one or more tasks, each with its own deadline. Individual tasks cannot be spread out among multiple processors. The size of the tasks determines the granularity of the surge, and the variation of the deadlines determines its homogeneity.

Our surge measures are the following.

- *Minimum Deadline Measure:* Consider a surge of a given magnitude $S$, consisting of one or more tasks, all with the same deadline. The Minimum Deadline Measure, *MD(S)*, expresses the minimum deadline necessary for the surge so that the system can handle it without missing any deadlines. The smaller the value of the minimum deadline measure for a given surge, the better the system's surge-handling capability.
- *Recovery Time Measure:* This measure, denoted by *RT(S)*, expresses how quickly the effect of a surge of magnitude $S$ on the system fades away. It measures how much time elapses between the arrival of the surge and the point in time when the task schedule is back to what it would have been if no surge had occurred, and no longer has any memory of the surge.

Note that both our measures are curves, functions of the surge size $S$, rather than single numbers. *MD(S)*, which measures the ability of the system to handle a surge within some deadline given a certain underlying ambient workload, represents the reserve capacity the system has. *RT(S)*, the recovery time measure, determines how soon the system recovers and is no longer vulnerable should a second surge follow the first one.

The surge-handling capacity of a system depends primarily on the task assignment and scheduling algorithms as well as on the system architecture. Task assignment and scheduling algorithms determine how effectively the available processing capacity can be harnessed by the workload. The task assignment algorithm can determine how the load is spread out among the processors. For example, some assignment algorithms attempt to balance the load. This can pose difficulties for surges of large granularity, since if the surge consists of a single indivisible task, no one processor may be available which can execute it on time. Other assignment algorithms attempt to utilize as few processors as possible, thus leaving other processors free to handle even large surges. The latter type

of assignment algorithm must be used with caution, however, since it leaves little room for handling variations in execution time demands of the ambient (non-surge) workload. An example comparing such two task assignment algorithms is presented in the next section.

The uniprocessor scheduling algorithm used to determine when to run tasks assigned to individual processors has a similarly large impact on the surge-handling capability. Such algorithms differ in their ability to effectively utilize available processing capacity, and thus differ in their ability to handle surges. In the next section, we will see examples comparing the Earliest Deadline and Rate Monotonic scheduling algorithms.

The system architecture is another important factor in determining surge-handling capacity. To begin with, the architecture governs the raw underlying computational capacity of the system. Also, the interconnection network topology and communication protocol determine the speed – and overhead – with which tasks can be moved from a failed processor to a new one, thus affecting the size of the surge.

The distinction between performability and cost functions on the one hand, and the newly introduced surge-handling measures on the other, should now be fairly obvious. The former are application-centric measures which require detailed – and precise – information about the application before they can be formulated. It would be very difficult to compute these measures either for a generic case, where the focus is not so much on a single application but on a set of possible applications, or when information about the application is incomplete. The new surge-handling measures are not so much application-*centric* as they are application-*sensitive*, in that while their interpretation can be from the point of view of the application (i.e., how much surge handling capability is required by a specific application), they can be computed without detailed information about the dynamics of the controlled process. Further discussion regarding this distinction appears in the final section of this paper.

## 4 The Measures – Illustrations

To illustrate the use of the two surge-handling measures defined in the previous section for assessing some system attributes, we selected a real-time embedded system comprised of $n$ processors connected through some interconnection network. The ambient workload consists of $m$ periodic tasks, where task $k$ has a period of $P_k$, an execution time of $E_k$, a deadline $D_k = P_k$, and an arrival time (of the first iteration of task $k$) equal to $T_k$. The load imposed on the system by task $k$ is measured by $U_k = \frac{E_k}{P_k}$ $(k = 1, ..., m)$.

Scheduling of tasks to processors can be done either by using bin-packing, i.e., balancing the load of the $n$ processors, or by the "first-fit" method, which fills each processor up to capacity before moving to the next one. After allocating the $m$ tasks to the $n$ processors, processor $i$ has $m^{(i)}$ tasks, with execution times $E_k^{(i)}$ and periods $P_k^{(i)}$ $(k = 1, ..., m^{(i)}$, $i = 1, ..., n)$.

At time $T_s$, the system experiences a surge of size $S$, with a deadline of $D_s$. Such a surge may arise either from the arrival of aperiodic tasks or from tasks that have been displaced because their processor has failed, and which must therefore be moved to other, functional, processors. The surge is divided among the $n$ processors by equalizing the loads of the processors as much as possible. $S$ is divided into $S^{(1)}, S^{(2)}, \cdots, S^{(n)}$ where $S^{(i)}$ is assigned to processor $i$ and $\sum_{i=1}^{n} S^{(i)} = S$.

The order of execution of tasks within a processor is either Rate Monotonic (RM), or follows the Earlier Deadline First (EDF) rule [4, 6]. In the RM algorithm, periodic tasks are assigned a static priority, which is proportional to the inverse of their periods. The EDF algorithm, as its name implies, executes the pending task with the earliest deadline. Both algorithms are preemptive.
Surge handling is successful if neither the surge, nor any ambient task, miss their deadlines. We will assume the worst case scenario, i.e., $T_1 = T_2 = \cdots = T_m = T_s = 0$. We will also assume, without loss of generality, that $P_1 \leq P_2 \leq \cdots \leq P_m$.

### 4.1 The Minimum Deadline Measure

The minimum deadline for a given surge clearly depends on the specific uniprocessor scheduling algorithm employed by each of the processors. We will demonstrate its calculation for both the Rate Monotonic (RM) and the Earliest Deadline First (EDF) scheduling protocols.
To calculate the minimum deadline measure for a given processor $i$, following the EDF scheduling protocol, define:
$\alpha_k^{(i)}(\tau) = \left\lfloor \frac{\tau}{P_k^{(i)}} \right\rfloor$ where $\lfloor x \rfloor$ the largest integer smaller than or equal to $x$.
$\alpha_k^{(i)}(\tau)$ is the number of iterations of task $k$ whose deadline occurs prior to or at time $\tau$.
Denote by $MD^{(i,EDF)}(S^{(i)})$ the minimum deadline necessary for a successful execution of a surge section of size $S^{(i)}$ arriving at processor $i$ at time 0. This is the shortest time $t$ in which the surge $S^{(i)}$ can be executed, all the tasks whose deadline is up to $t$ are executed on time, and no other task misses its deadline. Thus,

$$MD^{(i,EDF)}(S^{(i)}) = min \left\{ t \;\middle|\; \begin{array}{l} \sum_{k=1}^{m^{(i)}} \alpha_k^{(i)}(t) E_k^{(i)} + S^{(i)} = t \text{ and} \\ \sum_{k=1}^{m^{(i)}} \alpha_k^{(i)}(\tau) E_k^{(i)} + S^{(i)} \leq \tau, \tau = j P_l^{(i)} \, for\, any\, integers\, j, l \\ \text{where } t < j P_l^{(i)} \leq \prod_{k=1}^{m^{(i)}} P_k^{(i)} \end{array} \right\}$$

The minimum deadline for the whole system of $n$ processors, for a surge of size $S$ and following the $EDF$ scheduling protocol, is

$$MD^{(EDF)}(S) = \max_{1 \leq i \leq n} MD^{(i,EDF)}(S^{(i)})$$

To calculate $MD$ for the RM scheduling protocol, define for a given processor $i$,
$\beta_k^{(i)}(\tau) = \left\lceil \frac{\tau}{P_k^{(i)}} \right\rceil$ where $\lceil x \rceil$ is the smallest integer greater than or equal to $x$.
$\beta_k^{(i)}(\tau)$ is the number of iterations of task $k$ which arrived at processor $i$ prior to (but not at) time $\tau$.
In addition, denote: $h^{(i)}(\tau) = max \left\{ k \mid P_k^{(i)} \leq \tau \right\}$ ($h^{(i)}(\tau) = 0$ if $\tau < P_1^{(i)}$).
$h^{(i)}(\tau)$ is the index of the last task in processor $i$ whose period is not greater than $\tau$.

Thus,

$$MD^{(i,RM)}(S^{(i)}) = min \left\{ t \;\middle|\; \begin{array}{l} \sum_{k=1}^{h^{(i)}(t)} \beta_k^{(i)}(t) E_k^{(i)} + S^{(i)} = t \quad \text{and} \\ \sum_{k=1}^{h^{(i)}(P_l^{(i)})} \beta_k^{(i)}(\tau) E_k^{(i)} + S^{(i)} \leq \tau, \tau = jP_l^{(i)} \, for\ any\ integers\ j, l \\ \text{where } t < jP_l^{(i)} \leq \prod_{k=1}^{m^{(i)}} P_k^{(i)} \end{array} \right\}$$

and

$$MD^{(RM)}(S) = \max_{1 \leq i \leq n} MD^{(i,RM)}(S^{(i)})$$

We performed some numerical calculations to demonstrate the use of the $MD$ measure for comparing different system attributes. In all our numerical calculations, unless stated otherwise, we used the randomly selected set of values shown in Table 1. The number of processors is $n = 8$ and the number of tasks is $m = 24$. The arrival times are $T_i = T_s = 0$.

| Task No. | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Period | 10 | 12 | 12 | 13 | 14 | 15 | 16 | 16 | 17 | 17 | 18 | 18 |
| Execution Time | 3 | 4 | 2 | 4 | 4 | 1 | 5 | 3 | 1 | 1 | 4 | 4 |
| Load | .30 | .33 | .17 | .31 | .29 | .07 | .31 | .19 | .06 | .06 | .22 | .22 |

| Task No. | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Period | 18 | 19 | 19 | 19 | 20 | 20 | 20 | 20 | 20 | 20 | 21 | 24 |
| Execution Time | 3 | 5 | 5 | 4 | 6 | 3 | 2 | 5 | 5 | 6 | 7 | 8 |
| Load | .17 | .26 | .26 | .21 | .30 | .15 | .10 | .25 | .25 | .30 | .33 | .33 |

**Table 1.** The periods, execution times and loads of the 24 tasks.

Figure 1 depicts the minimum deadline for a given surge for the RM and the EDF task scheduling algorithms. We clearly see that EDF is superior, allowing the system to successfully handle surges with smaller deadlines.
In Figure 2 we illustrate the effect of increasing the number of processors in the system. The same 24 periodic tasks as in Figure 1 are assumed here and the EDF scheduling algorithm is employed. Clearly. the larger the number of processors the faster the surge handling, but the marginal advantage of increasing the number of processors decreases.
In the previous two figures we have assumed that the surge appears in the worst possible time instant when all 24 tasks are waiting to be executed, i.e., at t=0. In Figure 3 we examine the dependence of our measure on the exact time instant when the surge occurs. The same 24 periodic tasks are assumed here and the EDF scheduling algorithm is employed.
Figure 4 depicts the combined effect of varying both the scheduling algorithm and the number of processors on the surge handling capability of the system.

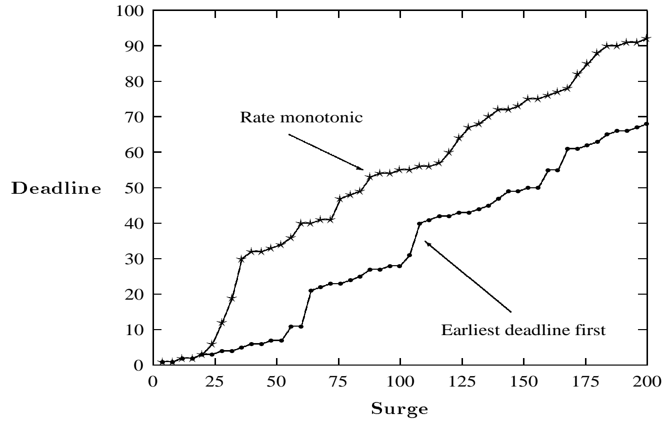


**Figure 1:** Minimum deadline for a given surge for different task scheduling algorithms (eight processors and 24 tasks).

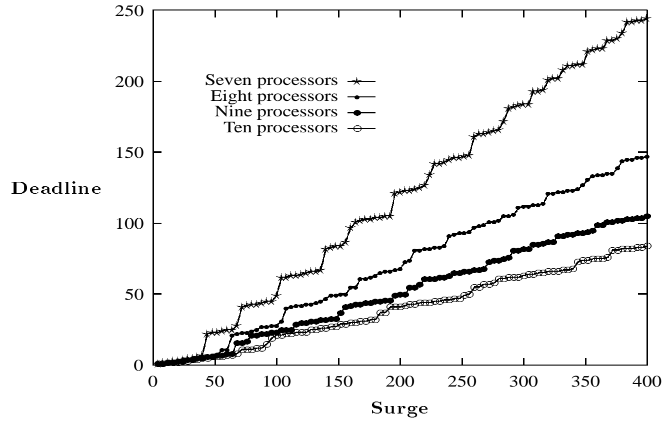


**Figure 2:** Minimum deadline for a given surge for different numbers of processors (24 tasks).

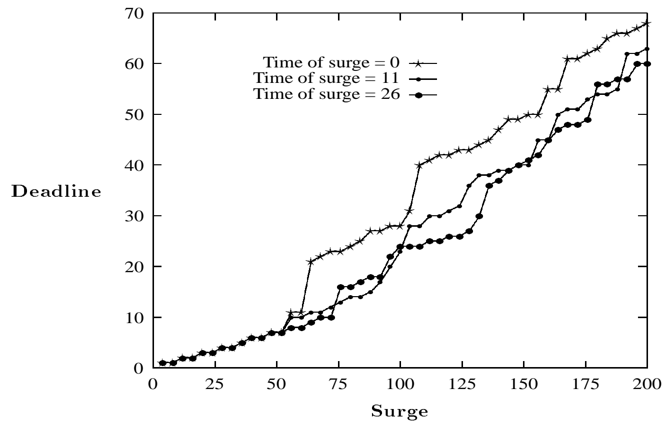


**Figure 3:** Minimum deadline for a given surge for different surge times (eight processors and 24 tasks).

### 4.2 The Recovery Time Measure

*RT(S)*, the recovery time for a given surge $S$, does not depend on the scheduling algorithm within the processor, as long as the procedure is "work conserving", i.e., the processor is never idle when there are tasks to be executed. *RT(S)* can be calculated as follows.

As before, define for a given processor $i$: $\beta_k^{(i)}(\tau) = \left\lceil \frac{\tau}{P_k^{(i)}} \right\rceil$.

The recovery time from a surge of size $S^{(i)}$ for processor $i$ is the shortest time $t$ in which the surge and all the ambient tasks which arrived up to $t$ can be executed. Therefore,

$$RT^{(i)}(S^{(i)}) = min \left\{ t \;\middle|\; \sum_{k=1}^{m^{(i)}} \beta_k^{(i)}(t) E_k^{(i)} + S^{(i)} \; = \; t \right\}$$

and the recovery time for the $n$ processor system is

$$RT(S) = \max_{1 \leq i \leq n} RT^{(i)}(S^{(i)})$$

Figure 5 depicts the recovery time as a function of the surge for an eight processor system with the same 24 periodic tasks as before. The scheduling algorithm employed is EDF and the surge is assumed to occur at t=0. We compare two task allocation algorithms, namely the first-fit and the bin-packing algorithms. We also consider two values of surge granularity, where the surge is divided into either three or five indivisible tasks. We can see that for the higher surge granularity (surge is divided into three tasks) bin-packing is preferred, while for the lower granularity first-fit is better.

Another important use for the recovery time measure is for comparing different fault-recovery procedures. The overhead involved in recovering from a fault can be considered a surge, and the better the recovery procedure, the shorter the surge recovery time. In this case, the *RT* measure can be used for comparing the different attributes of the recovery procedure. In Figure 6, the effects of the checkpointing interval and the checkpointing overhead are investigated. An intermittent fault is assumed to occur at processor 2 at time 0. Figure 6 depicts the recovery time as a function of the checkpointing interval for two values of the checkpointing overhead, namely 1 and 2 time units. Clearly, there is an optimal value of the checkpointing interval, and it is larger for the larger value of the overhead.

## 5 Discussion and Conclusions

Traditional performance measures have tended to be either totally computer-centric or totally application-centric. The problem with the former is that they do not take the needs of the application into account. While the latter type of measure is ideal when perfect information is available about the application, it is useless when the application is still under development, and full information about it is not available. By contrast, the surge-handling measures introduced here can be computed for a system using the current state of knowledge about the ambient workload. Another case when application-centric measures are useless is when we want to characterize the computer, not in the context of a specific application, but with respect to its general suitability for real-time applications. On the other hand, it is possible to characterize the surge-handling capability of the system for any ambient workload and any architecture. Recently, efforts

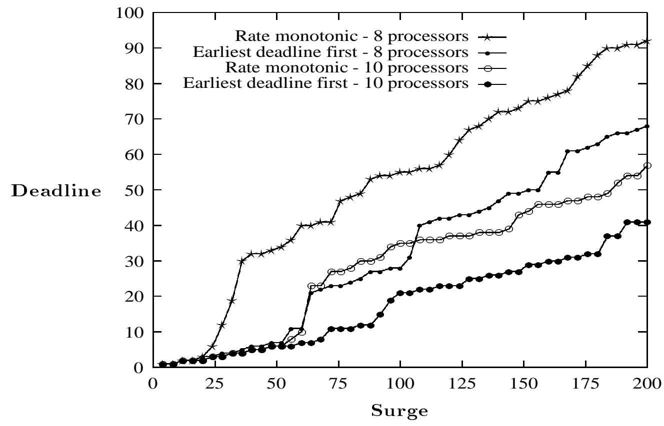


**Figure 4:** Minimum deadline for a given surge for different scheduling algorithms and numbers of processors (24 tasks).

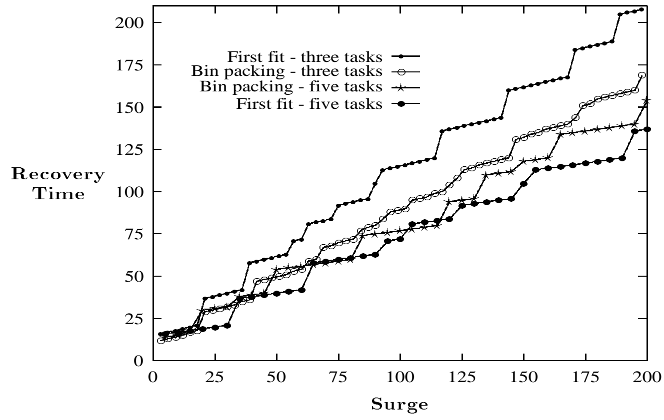


**Figure 5:** Recovery time for a given surge.

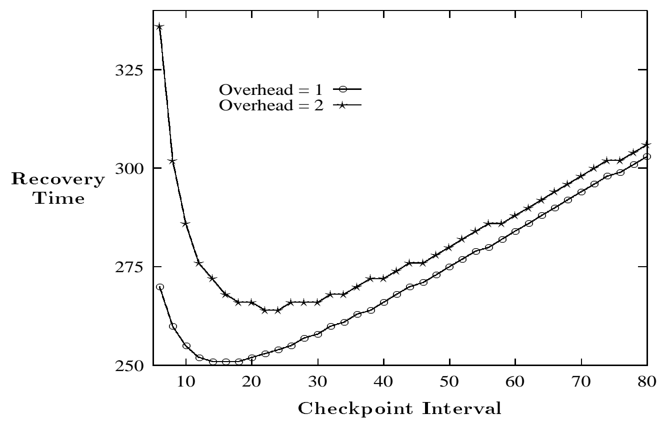


**Figure 6:** Recovery time for a given checkpoint interval.

have been made by several research teams to build standard real-time benchmarks, most notably by Mitre and Honeywell corporations. These benchmarks can be used to define the ambient workloads in terms of which the surge-handling measures can be evaluated. The surge-handling measures can also be used to evaluate the quality of real-time operating systems, especially their task assignment and scheduling algorithms. Other features that can be evaluated using these measures are the interconnection topology and the communication protocols (since they determine the costs associated with moving tasks) as well as the failure recovery procedures, including the checkpoint placement strategy [5].

**Acknowledgment:** This effort was supported in part by the Defense Advanced Research Projects Agency and the Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0341, order E349. The government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.
The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, or the Defense Advanced Projects Agency, Air Force Research Laboratory, or the U. S. Government.

## References

1. M. D. Beaudry, "Performance-Related Reliability Measures for Computing Systems," *IEEE Trans. Computers*, Vol. C-29, 1978.
2. C. M. Krishna and K. G. Shin, "Performance Measures for Multiprocessor Controllers," in A.K. Agrawala and S.K. Tripathi, *eds.*, *Performance '83*, 1983.
3. C. M. Krishna and K. G. Shin, "Performance Measures for Control Computers," *IEEE Trans Automatic Control*, Vol. AC-32, 1987.
4. C. M. Krishna and K. G. Shin, *Real-Time Systems*, New York: McGraw-Hill, 1997.
5. C. M. Krishna, K. G. Shin, and Y.-H. Lee, "Optimization Criteria for Checkpointing," *Communications of the ACM*, Vol. 27, No. 10, 1984.
6. C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment," *Journal of the ACM*, Vol. 20, 1973, pp. 46–61.
7. J. F. Meyer, "On Evaluating the Performability of Degradable Computing Systems," *IEEE Trans. Computers*, Vol. C-29, 1980.
8. J. F. Meyer, D. G. Furchtgott, and L. T. Wu, "Performability Evaluation of the SIFT Computer," *IEEE Trans. Computers*, Vol. C-29, 1980.
9. K. G. Shin and C. M. Krishna, "Characterization of Real-Time Computers," *NASA Contractor Report 3807*, August 1984.
10. K. G. Shin and C. M. Krishna, "New Performance Measures for Design and Analysis of Real–Time Multiprocessors," *Journal of Computer Science and Engineering Systems*, Vol. 1, pp. 179–192, October 1986.
11. K. G. Shin, C. M. Krishna, and Y.-H. Lee, "A Unified Method for Characterizing Real–Time Computer Controller and its Application," *IEEE Transactions on Automatic Control*, Vol. AC–30, No.4, April 1985, pp. 357–366.
12. K. Yu and I. Koren, "Reliability Enhancement of Real-Time Multiprocessor Systems through Dynamic Reconfiguration," *Fault-Tolerant Parallel and Distributed Systems,* D. Pradhan and D. Avresky (Editors), pp. 161-168, IEEE Computer Society Press, Los Alamitos, CA, 1995.

This article was processed using the LaTeX macro package with LLNCS style