NASA Technical Memorandum 85665

Time-Critical Multirate Scheduling Using Contemporary Real-Time Operating System Services

Dave E. Eckhardt, Jr.

SEPTEMBER 1983

25th Anniversary
1958-1983

NASA

NASA Technical Memorandum 85665

Time-Critical Multirate Scheduling Using Contemporary Real-Time Operating System Services

Dave E. Eckhardt, Jr.
Langley Research Center
Hampton, Virginia

National Aeronautics
and Space Administration

**Scientific and Technical
Information Branch**

1983

INTRODUCTION

A time-critical computer system is a real-time system in which the environment is said to be "hard real-time" (ref. 1). Because time-critical applications have hard deadlines, the system response time must be guaranteed to occur within a specified time frame. Unknown and untimely responses cannot be tolerated in time-critical applications.

For the most part, however, real time is taken to encompass a wide range of computer applications (ref. 2). Real time, for example, may imply a system response time which can vary from the millisecond response time necessary for automatic control of radar systems to the multisecond response time of airline reservation systems and point-of-sale systems. Some definitions include any response time that is necessary to control a given environment. Thus, response times of minutes and even hours could be included. Additionally, one could describe the response times of these systems in probabilistic terms. One might specify, for example, that 90 percent of the time a response to an input stimulus will occur within some specified time interval. Therefore, a fixed response is not guaranteed on each interaction with the system. Deadlines, then, are "soft" and sometimes will be relaxed during overloaded conditions.

There are many minicomputer manufacturers that provide real-time operating systems. (Ref. 3 provides a comprehensive list.) To various degrees, these operating systems provide the services necessary to process multirate time-critical applications. These services may include, for example, (1) provisions for designating tasks as real time in order to receive high priority service or special capabilities (such as locking tasks in memory), (2) provisions for dynamically controlling individual task priorities, (3) provisions for creating, suspending, resuming, and deleting tasks, (4) interprocess communications facilities such as semaphores and event flags, (5) means for entering a state of inactivity while waiting for signals from other tasks or waiting for the occurrence of a user-defined event, (6) means to share common memory areas among tasks, and (7) preemptive processor scheduling. However, even with this seemingly complete set of operating system services, it is still necessary to develop an executive-level (i.e., at a level above the operating system) scheduling mechanism in order to coordinate a set of synchronized, time-critical tasks which execute at different cyclic rates. This paper will examine time-critical scheduling algorithms and address executive-level scheduling requirements.

TIME-CRITICAL TASKS

A time-critical task is characterized by repetitions of C units of processing over a frame interval F within a deadline time $D \leq F$. The processor load factor L is equal to C/F. For asynchronous tasks, a variable frame time might be desirable; however, in practice a minimum frame time is specified in order to bound the processor load factor. The deadline, within which all processing must be completed, may result from a particular system implementation; that is, the output data must be available by D so that the system will have time to transfer the data out by the end of the frame. The deadline may also result from the requirements of the application itself. For example, a closed-loop control system will have time delay constraints to avoid inducing stability problems. Often, it is sufficient to let $D = F$.

For any set of time-critical tasks, it is necessary to determine the compatibility of the set, where compatibility refers to the ability to process all tasks of the set without having missed deadlines. Depending on the particular implementation of the time-critical system, this may simply require that the sum of all load factors plus some overhead factor is less than 1. On other implementations, a check of this necessary condition may not be sufficient. One brute force method of checking compatibility is to analyze each combination of possible frame occurrences. In effect, the entire time sequence of all time-critical processes operating together is simulated to ensure that each and every deadline can be met. Less complex algorithms (e.g., ref. 4) check only sufficient conditions for compatibility. This, of course, is a more conservative approach, since there may be compatible sets that do not meet the sufficiency conditions. Whatever approach is used, including a trial worst-case run on the actual system, it is useful for the system to detect and report missed deadlines.

TIME-CRITICAL SCHEDULING ALGORITHMS

The basic objective of a time-critical scheduling algorithm is to allocate processor time to a given number of time-critical tasks while at the same time maintaining the hard deadlines for these tasks. An efficient, time-critical scheduling algorithm, then, attempts to maximize the availability of processor time for time-critical tasks while minimizing the effects of constraints imposed by multiple and different frame intervals. Reducing the effects of these constraints, of course, increases the number of tasks that can exist together. Basically these algorithms can be classified according to the manner in which priorities are determined; that is, schedulers are either nonpriority, static priority, or dynamic priority.

The nonpriority algorithm is a time-slicing one that is somewhat analogous to a round-robin scheduler. Time is divided into quantum slices Q and each task i is given a proportion of the quantum based on the task load factor, i.e., L_iQ. If the system has interrupts, then the interrupts associated with the task are only allowed to be active during this time. If the deadline is equal to the frame time and if frames are restricted to be a multiple of the quantum, then over its frame time the ith task receives

$$(D_i/Q)L_iQ = L_iF_i = C_i$$

units of processor time within its deadline. This fulfills its requirements. If overhead is neglected, then the sum of all load factors being less than or equal to 1 is a necessary and sufficient condition to insure that all task deadlines can be met. Furthermore, 100 percent of the processor can be utilized by time-critical tasks. However, neither is true if $D_i < F_i$, as can be seen by the following example. Suppose there exist two tasks with the following parameters: $F_1 = F_2 = D_1 = 1$, $D_2 = 0.50$, $C_1 = 0.60$, and $C_2 = 0.25$. Observe that, even though over 1 unit of time only 0.85 processor units are required, task 2 has only received $D_2L_2 = 0.125$ within its deadline. This is less than the required C_2.

In the limit as Q approaches 0, this algorithm reduces to one in which each time-critical task appears to have its own processor. Task i has a processor with a processing rate L_iR, where R is the actual processing rate of the processor. Note that this is different from a processor-shared model of the batch round-robin scheduler, where for m tasks the processors are homogeneous and process at the same rate R/m. In addition to the problem with this scheduler when $D_i < F_i$, it is subject to a great deal of context switching, and so overhead may become excessive.

In the fixed priority algorithm, the static scheduler assigns each time-critical task a priority as it enters the system. This priority will remain constant for the task. These priorities will determine the order in which time-critical tasks receive processor service. Note, it would be possible to process the two tasks of the previous example if task 2 is given priority over task 1 so that task 2 always gets the first 0.25 processor units each frame. Liu (ref. 5) shows that for the case where all $D_i = F_i$, a priority assignment based on iteration rate (smallest frame time having highest priority) is optimum. It is optimum in the sense that any other set of fixed priority assignment rules which can schedule a set of time-critical tasks can also be scheduled by a priority assignment based on frame time.

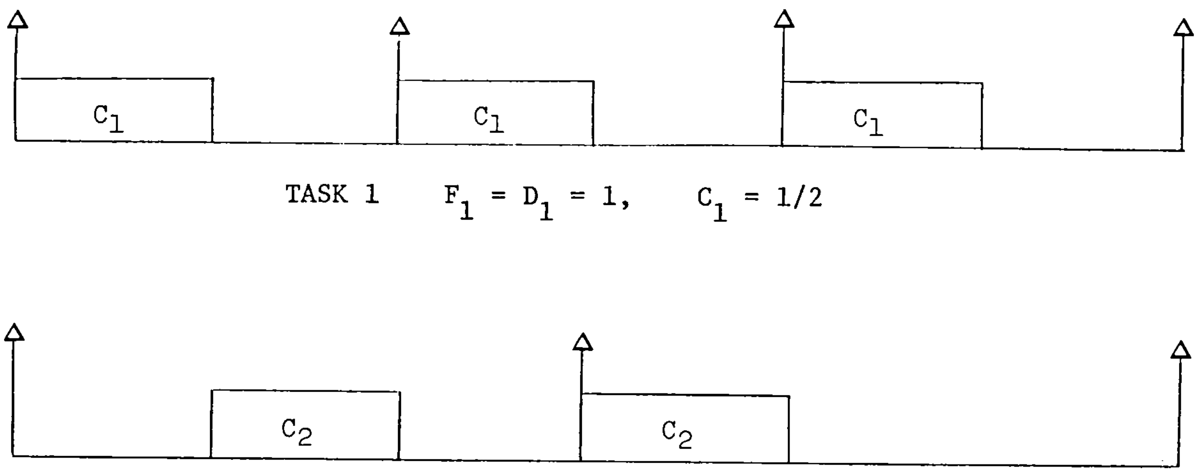
As previously mentioned, the performance of the time-critical schedule is determined by its ability to allocate a maximum amount of processor time to time-critical tasks while maintaining task deadlines. For a given set of tasks, there will be an upper bound of processor utilization U, which is the sum of all load factors. Varying the frame times for this set of tasks may decrease the maximum utilization that is feasible (i.e., where all deadlines are met). The worst-case processor utilization is the least upper bound of processor utilization. For m tasks, both Liu (ref. 5) and Serlin (ref. 6) show that for the fixed priority scheduling with all $D_i = F_i$, this least upper bound is

$$U = m(2^{1/m} - 1)$$

If the set of m tasks has a total processor utilization below this number, then there exists a fixed priority assignment which is feasible. Above this number, an assignment is feasible only if the frames are suitably related. The least upper bound U degrades to approximately 70-percent utilization as m approaches infinity. In the worst-case, then, for a large set of tasks, there could exist conditions for which only 70 percent of the processor time can be allocated to time-critical tasks.

Figure 1(a) shows an example of fixed priority scheduling with $F_1 = D_1 = 1$, $C_1 = 1/2$, and $F_2 = D_2 = 3/2$. As can be seen, if task 1 has a higher priority than task 2 (based on frame rate), then $C_2 = 1/2$ is the maximum amount of processor time that can be allotted to task 2 and results in $U = 5/6$. However, with an assignment as shown in figure 1(b), it is possible for C_2 to increase to 3/4 for 100-percent processor utilization. This improvement is brought about by the use of a dynamic priority algorithm, which is described next.

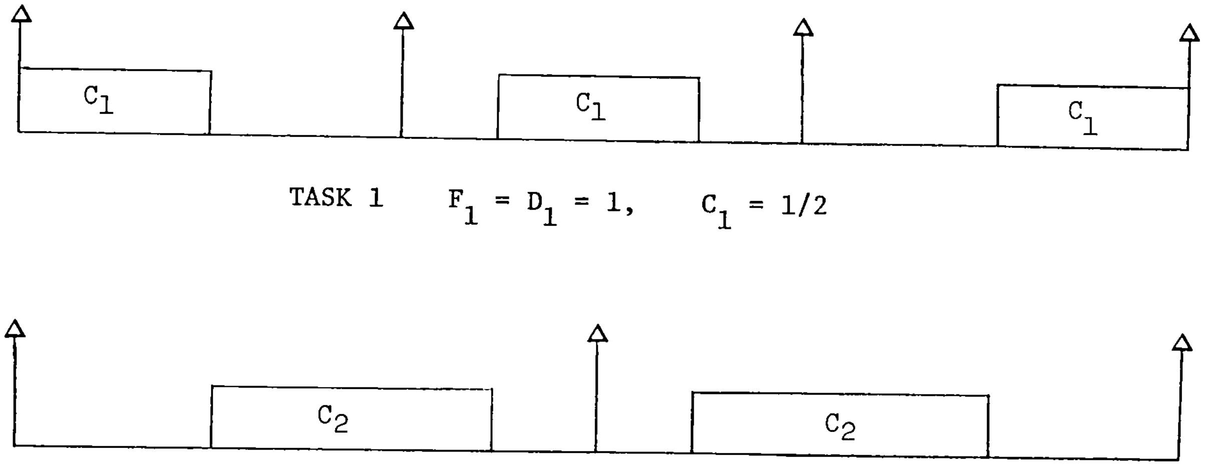
The dynamic priority algorithm will reevaluate the priorities of all tasks requiring the processor whenever an external interrupt, or clock, signals that a new time-critical task is queued up for the processor. The task whose deadline will occur next has the highest priority and is assigned the processor. For this reason, this algorithm is often called the "least-time-to-go" or the "relative urgency" algorithm (ref. 7).



TASK 1 $F_1 = D_1 = 1$, $C_1 = 1/2$

TASK 2 $F_2 = D_2 = 3/2$, $C_2 = 1/2$

(a) Fixed priority allocation.



TASK 1 $F_1 = D_1 = 1$, $C_1 = 1/2$

TASK 2 $F_2 = D_2 = 3/2$, $C_2 = 3/4$

(b) Dynamic priority allocation.

Figure 1.- Examples of fixed and dynamic priority scheduling.

As seen in figure 1(b), initially task 1 has the next deadline and, therefore, has priority over task 2. However, during the second frame of task 1 and the first frame of task 2, task 2 has priority over task 1. During the third frame of task 1 and the second frame of task 2, both tasks have equal priority, and task 2 is not interrupted.

Although not shown in this example, interruption of the processing of a task to allow for a higher priority task is sometimes necessary. However, as is the case with a fixed priority system, there is less task switching than with the time-slicing algorithm. Presumably, then, there is also less overhead. Another advantage of dynamic scheduling is that for the case where all $D_i = F_i$ and the overhead time is 0, 100-percent processor utilization is obtainable, and the sum of the load factors being less than or equal to 1 is a necessary and sufficient condition for all tasks to meet deadlines (ref. 5).

SCHEDULING REQUIREMENTS

A time-critical scheduling facility would ideally provide the following capabilities:

(1) Means to synchronize a set of tasks with different iteration rates. This is a desirable feature, since the iteration rates of segments of the application can be tailored to application dynamics rather than executing the entire application at the highest iteration rates required.

(2) Means to determine if the processes are maintaining synchronization while at the same time meeting the processing requirements.

(3) Precedence ordering within a frame interval; that is, the physical system will dictate that outputs from some tasks be inputs to other tasks, and thus a fixed ordering is required.

(4) Efficient task control services, such as task waking, context switching, and setting and clearing event flags, must be time efficient, since large system overheads are detrimental to tasks with iteration rates on the order of 20 to 50 iterations per second.

(5) Maximum utilization of the processor by the time-critical tasks. In addition to the system overhead, the scheduling algorithm, as previously shown, also dictates the availability of the processor.

EXECUTIVE-LEVEL SCHEDULING

Next we shall consider executive-level scheduling algorithms for a time-critical application which is comprised of a set of multirate tasks. We will assume a system in which task processing can be preempted by higher priority tasks. The executive routine will execute at the highest priority with a frame size F. The time-critical tasks can execute at a frame which is any integral multiple of F. Task synchronization is accomplished by the executive, which controls the initiation of frame processing for all tasks. An operating system service is required to activate the executive at regular frame intervals.

The executive also determines if tasks are maintaining time synchronization. A lost-time-synchronization indicator, lts, is maintained for each time-critical task. These indicators are set by the executive at the beginning of a task frame interval and are cleared by the task itself when it has completed processing for that frame. The most effective means of implementing the lost-time-synchronization indicator is the use of a table in a shared memory area. Each task, as it is created, is given its own index into the table.

Precedence ordering of tasks is handled external to the executive. An effective means for implementing ordering is to group all tasks that have the same frame interval into a single task, each with a front-end miniexecutive. The executive invokes the miniexecutives through operating system services, and each miniexecutive invokes its ordered set of tasks through direct subroutine calls. This approach reduces the number of interactions with the operating system.

A procedure for implementing a static scheduling algorithm is given below

```
PROCEDURE static
LOOP:
     wait (event = clock_interrupt);
     FOR i := 1 TO numtasks DO
          IF count [i] < multiple [i] THEN
             count [i] := count [i] + 1;
          ELSE
               IF lts[i] = false THEN
                    BEGIN
                         count [i] := 1;
                         lts [i] := true;
                         wake (task (i));
                    END
               ELSE
                    lostsync (task (i));
ENDLOOP
```

The basic executive loop begins when the executive is activated by the system at the beginning of a frame. For each task, the executive will increment a running frame count, count [i], to a maximum value, multiple [i], which represents the number of integral executive frame intervals per frame of the ith task. When a running frame count reaches the frame multiple, the associated task should have completed frame processing, cleared the lost-time-synchronization indicator, and be in a state awaiting activation for the next frame. If not, then a lost-time-synchronization indication is reported to a procedure for handling that condition. Initially, each task is assigned a priority that is proportional to its cyclic frame rate, so that once the executive initiates tasks, the operating system will provide the appropriate preemption and task scheduling. Also count [i] should be initialized to multiple [i] and lts [i] to false so that all tasks will begin synchronized processing on the first frame.

As previously mentioned, a dynamic scheduling algorithm generally provides more processor time for synchronized time-critical applications. However, more system overhead is induced, since the priorities of the time-critical tasks must be regularly updated with operating system services. A procedure for the dynamic scheduling algorithm is given below

```
PROCEDURE dynamic
LOOP:
     wait (event - clock_interrupt);
     FOR i: = 1 TO numtasks DO
         IF count [i] < multiple [i] THEN
           BEGIN
            count [i] := count [i] + 1;
            priority [i] := max - multiple [i] + count [i];
            setpriority (task (i), priority (i));
           END
         ELSE
            IF lts[i] = false THEN   (*TASK DONE*)
                BEGIN
                      count [i] := 1;
                      lts [i] := true;     (*CLEARED BY TASK AT TERMINATION*)
                      priority [i] := max - multiple [i] + count [i];
                      setpriority (task (i), priority (i));
                      wake (task (i), priority (i));
                END
            ELSE
                lostsync (task (i));
ENDLOOP
```

A priority index is maintained for each task. This priority will increase to a maximum priority, max, when the task reaches the last executive frame which makes up the individual task frame. The maximum task priority must be less than the priority of the executive. Count [i] and lts [i] are initialized as in the preceding algorithm.

These algorithms were implemented on a Digital Equipment Corporation VAX-11/780 computer running the VAX/VMS real-time operating system so that their effectiveness could be investigated. The single performance measure used was the amount of processor time that was available for time-critical tasks. This measure is a function of the number of tasks, the number and variations of task frame sizes, and the amount of processing time each individual task requires. Thus, there are many combinations of these factors. The results of one such combination are shown in figure 2. Each run consists of N tasks (from 1 to 10). The ith task has a frame size iF (F = 50 ms), and the available processor time is distributed such that all task load factors are equal. The results demonstrate, for this particular system, that when there is a requirement for frame sizes which can be any multiple of the executive frame interval, then a dynamic algorithm is more effective. For the dynamic algorithm, the unavailability of the processor is due entirely to system overhead. For the static algorithm, overhead is lower, and for $i < 3$, this algorithm is more effective. However, for $i \geqslant 3$, this particular combination of frame sizes reduces the available processor time for time-critical tasks (although it is available for asynchronous background tasks). Thus, the requirements (e.g., combination of frame sizes) of a particular application will dictate the more appropriate scheduling algorithm.

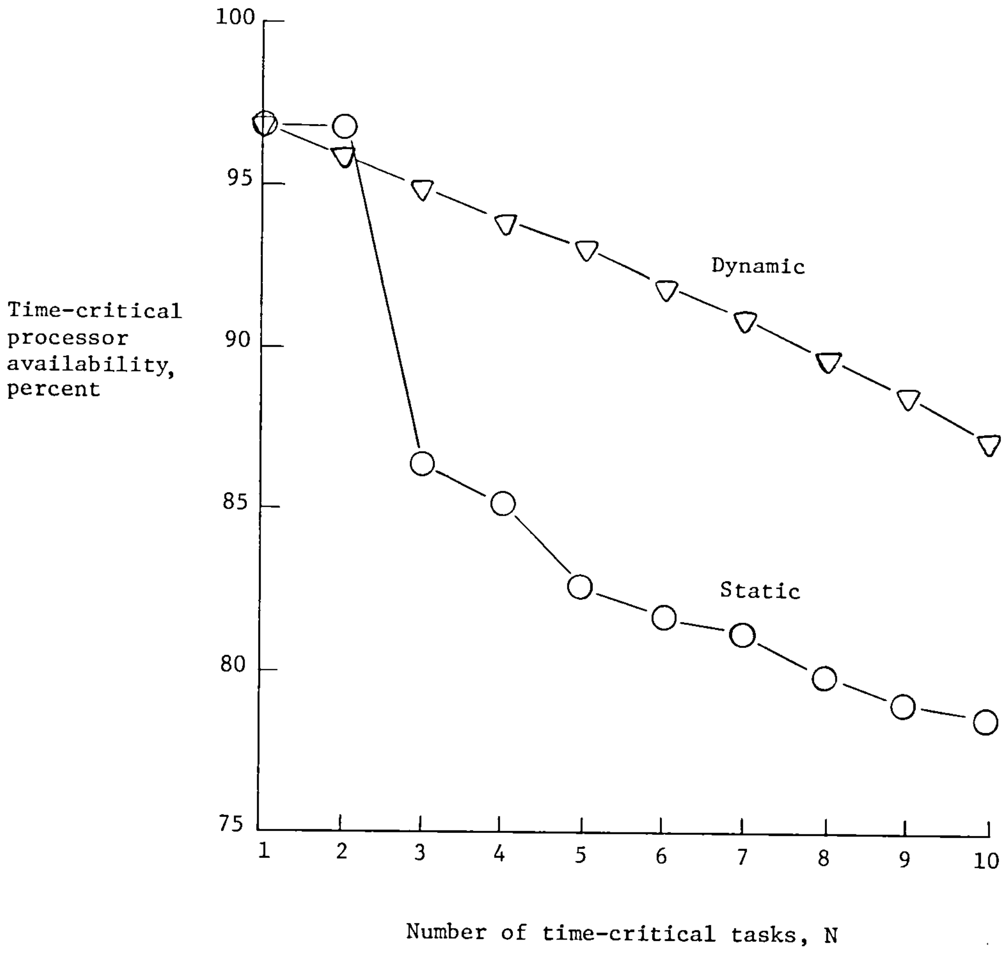


Figure 2.- Comparison of dynamic and static scheduling algorithms for $F_i = 50i$ ms, $i = 1$ to N, and $L_i = L_j$ for $i, j = 1$ to N.

CONCLUDING REMARKS

Time-critical applications differ from so-called real-time applications in that the former have hard deadlines which must be met, whereas the latter have soft deadlines which will sometimes be relaxed during system-overloaded conditions. It is not surprising that real-time systems do not always provide time-critical scheduling algorithms. It is shown here that these algorithms can often be provided within the constraints of services provided by contemporary real-time operating systems. The cost that is incurred is the overhead of implementing these algorithms at a level above the operating system.

Langley Research Center
National Aeronautics and Space Administration
Hampton, VA 23665
August 17, 1983

REFERENCES

1. Manacher, G. K.: Production and Stabilization of Real-Time Task Schedules. J. Assoc. Comput. Mach., vol. 14, no. 3, July 1967, pp. 439-465.

2. Martin, James: Design of Real-Time Computer Systems. Prentice-Hall, Inc., c.1967.

3. Datapro Reports on Minicomputers. Datapro Research Corp., c.1979.

4. Steinbach, Gary R.: and Gracon, Thomas J.: Pre-Determination of Schedulability for Least-Time-To-Go Interrupt Scheduling Schemes. Proceedings of the 1974 Summer Computer Simulation Conference, Simulation Councils, Inc., 1974, pp. 115-119.

5. Liu, C. L.; and Layland, James W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. J. Assoc. Comput. Mach., vol. 20, no. 1, Jan. 1973, pp. 46-61.

6. Serlin, Omri: Scheduling of Time Critical Processes. AFIPS Conference Proceedings, Volume 40 - 1972 Spring Joint Computer Conference, AFIPS Press, pp. 925-932.

7. Fineburg, Mark S.; and Serlin, Omri: Multiprogramming for Hybrid Computation. AFIPS Conference Proceedings, Volume 31 - 1967 Fall Joint Computer Conference, Thompson Book Co., c.1967, pp. 1-13.

1. Report No. NASA TM-85665	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle TIME-CRITICAL MULTIRATE SCHEDULING USING CONTEMPORARY REAL-TIME OPERATING SYSTEM SERVICES		5. Report Date September 1983
		6. Performing Organization Code 505-35-33-01
7. Author(s) Dave E. Eckhardt, Jr.		8. Performing Organization Report No. L-15644
9. Performing Organization Name and Address NASA Langley Research Center Hampton, VA 23665		10. Work Unit No.
		11. Contract or Grant No.
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546		13. Type of Report and Period Covered Technical Memorandum
		14. Sponsoring Agency Code
15. Supplementary Notes		

16. Abstract

Although real-time operating systems provide many of the task control services necessary to process time-critical applications (i.e., applications with fixed, invariant deadlines), it may still be necessary to provide a scheduling algorithm at a level above the operating system in order to coordinate a set of synchronized, time-critical tasks executing at different cyclic rates. This paper addresses the scheduling requirements for such applications and develops scheduling algorithms using services provided by contemporary real-time operating systems.

17. Key Words (Suggested by Author(s)) Scheduling algorithms Time-critical scheduling Multirate scheduling	18. Distribution Statement Unclassified - Unlimited Subject Category 59

19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 11	22. Price A02