A Time-Driven Scheduling Model for Real-Time Operating Systems

E. Douglas Jensen, C. Douglass Locke, Hideyuki Tokuda

Computer Science Department
Carnegie-Mellon University, Pittsburgh, PA 15213

Abstract

Process scheduling in real-time systems has almost invariably used one or more of three algorithms: fixed priority, FIFO, or round robin. The reasons for these choices are simplicity and speed in the operating system, but the cost to the system in terms of reliability and maintainability have not generally been assessed.

This paper originates from the notion that the primary distinguishing characteristic of a real-time system is the concept that completion of a process or a set of processes has a value to the system which can be expressed as a function of time. This notion is described in terms of a time-driven scheduling model for real-time operating systems and provides a tool for measuring the effectiveness of most of the currently used process schedulers in real-time systems. Applying this model, we have constructed a multiprocessor real-time system simulator with which we measure a number of well-known scheduling algorithms such as Shortest Process Time (SPT), Deadline, Shortest Slack Time, FIFO, and a fixed priority scheduler, with respect to the resulting total system values. This approach to measuring the process scheduling effectiveness is a first step in our longer term effort to produce a scheduler which will explicitly schedule real-time processes in such a way that their execution times maximize their collective value to the system, either in a shared memory multiprocessing environment or in multiple nodes of a distributed processing environment.

1. Introduction

Process scheduling in a computer operating system is merely an instance of an extensively studied problem from Operations Research (OR), in which it takes the form of producing a sequence of jobs which must utilize a common resource (e.g., a lathe in a machine shop) such that some metric (e.g., number of parts made in a day) is optimized (maximized or minimized). In the real-time operating system as well, the choice of process scheduling algorithms to allocate cpu resources can have a very important impact on the system performance as well as on the reliability and maintainability of the system.

This work was supported in part by the USAF Rome Air Development Center under contract number F30602-81-C-0297, the U.S. Naval Ocean Systems Center under contract number N66001-81-C-0484, the U.S. Navy Office of Naval Research under contract number N51160-84-C-0484, and the IBM Corporation, Federal Systems Division. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of RADC, NOSC, ONR, IBM, or the U.S. Government.

This paper is written from the context of the Archons project, currently under way at Carnegie-Mellon University. Archons is undertaking to create new resource management paradigms which are as intrinsically decentralized as possible, then applying them as the foundation of an experimental decentralized real-time operating system which will then be used in the design and construction of a "decentralized computer"[1, 2, 3]. The process scheduler for this system constitutes one of the critical research interests of the Archons project, which will be expected to manage time resources to ensure that the most important processes meet their time constraints, even in the event of an overload condition[4].

In the remainder of this section, we discuss real-time systems and the current state of real-time process scheduling, while in Section 2 we define our time-driven scheduling computational model. In Section 3 we describe our experimental simulation results, summarizing our results in Section 5.

1.1. The Scheduling Problem

Scheduling a set of processes consists of sequencing them on one or more processors such that the utilization of resources optimizes some scheduling criterion. Criteria which have historically been used to generate process schedules include maximizing process flow (i.e., minimizing the elapsed time for the entire sequence), or minimizing the maximum lateness (lateness is defined to be the difference between the time a process is completed and its deadline). It has long been known[5] that there are simple algorithms which will optimize certain such criteria under certain conditions, but algorithms optimizing most of the interesting scheduling criteria are known to be NP-complete[6], indicating that there is no known efficient algorithm which can produce an optimum sequence. Clearly, the choice of metric is crucial to the generation of a processing sequence which will meet the goals of the system for which the schedule is being prepared.

There are a number of significant differences between scheduling as it is practiced in most applications of OR and process scheduling in an operating system. In OR, scheduling is traditionally performed statically, off-line, while in an operating system, the information from which scheduling decisions must be made is dynamic, suggesting that the best scheduling decisions should be made dynamically. In addition, classic OR scheduling problems assume (for the sake of computational simplicity, not particularly for realism) that all the jobs to be scheduled and their processing time requirements, are available at the beginning of the sequence ($t = 0$), and that new jobs will not suddenly appear during processing. If a change in any data involved in the scheduling decision (such as the arrival of a new job) should occur, the previously computed sequence is invalidated, and scheduling must be started over if optimality is to be maintained. On the other hand, in the real-time operating system environment, processes frequently arrive and depart (i.e., are completed or terminated) at irregular intervals and have stochastic processing times.

Real-time processes can be partitioned into two categories: periodic and aperiodic. Periodic processes arrive at regular intervals, while aperiodic processes arrive irregularly, but even periodic processes can be terminated or undergo changes in period due to application dependencies, resulting in a constantly changing set of schedulable processes.

1.2. Real-Time Systems

Although many real-time systems have been constructed, the fundamental differences between a real-time system and other computer systems have not always been clearly understood. It is our thesis that the most important difference between the real-time operating system and other computer systems is that in a real-time system, the completion of a process *has a value to the system which varies with time*[7]. Although the time to complete a process is of some importance in all computer systems, in a real-time system the response time is viewed as a crucial part of the *correctness of the application software*; a computation which is late is very frequently no better, or perhaps even worse, than one producing an incorrect result.

This time value is usually described in terms of deadlines by which computations must be completed, and the deadlines are generally traceable to the physical environment with which the system must interact; for example, a reaction temperature measurement must be completed in time to apply a correction in a manufacturing system. Stated another way, the value to the system for completing a process with a deadline is some high constant value which abruptly drops to zero after the deadline.

Real-time systems have been divided into two classes: *hard* real-time systems and *soft* real-time systems[8]. Hard real-time systems are those whose deadlines must absolutely be met or the system will be considered to have failed (and whose failure might be catastrophic), while soft real-time systems allow for some deadlines, at least occasionally, to be missed with only a degradation in performance but not a complete failure.

Many existing real-time systems are being used in a number of such diverse environments as space or airborne platform management, factory process control, and robotics. These involve several levels of real-time, characterized by the tightness of their deadlines, and range from closed-loop sensor/actuator systems (frequently implemented on microprocessors dedicated to a small number of devices) to supervisory systems involving human interaction to manage a complex command and control environment.

1.3. Scheduling in Existing Real-Time Systems

In observing a number of existing real-time systems, we note the characteristics found in the operating systems used managing their manage resources in support of their real-time operation (such operating systems are usually called *executives* since a full operating system is not usually implemented in a real-time system). Two primary characteristics of these operating systems can be described:

- These operating systems are kept simple, with minimal overhead, but also with minimal function. Virtual storage is almost never provided; file systems are usually either extremely limited or non-existent. I/O support is kept to an absolute minimum. Scheduling is almost always provided by some combination of FIFO (for message handling), fixed priority ordering, or round-robin, with the choice made by the operating system designer.

- Simple support for management of a hardware real-time clock is provided, with facilities for periodic process scheduling based on the clock, and timed delay primitives.

Conspicuously missing from these systems at the operating system level is any support for explicitly managing user-defined deadlines, even though meeting such deadlines is the primary characteristic of the real-time application requirements. Instead, these systems are designed to meet their deadlines by attempting to ensure that the available resources exceed the expected worst-case user requirements, and their implementation is followed by an extensive testing period in an attempt to verify that these requirements can be met under these expected loads.

In fixed priority scheduler systems, deadline management is attempted by assigning a high fixed priority to processes with "important" deadlines, disregarding the resulting impact to less "important" deadlines. During the testing period, these priorities are (usually manually) adjusted until the system implementer is convinced that the system "works". This approach can work only for relatively simple systems, since the fixed priorities do not reflect any time-varying value of the computations with respect to the problem being solved, nor do they reflect the fact that there are many schedulable sets of process deadlines which cannot be met with fixed priorities. In addition, implementers of such systems find that it is extremely difficult to determine reasonable priorities, since, typically, each individual subsystem implementer feels that his or her module is of high importance to the system. This problem is usually "solved" by deferring final priority determination to the system test phase of implementation, so the resulting performance problems remain hidden until it is too late to consider the most effective design solutions. This approach results not only in real-time systems with frequently marginal performance, but also in extremely fragile systems in the presence of changing requirements.

A scheduling algorithm which is seldom used in practice has some interesting properties, but also presents a serious pitfall. A deadline scheduler (i.e., a scheduler which schedules the process with the closest deadline first[9]) solves the problem of missing otherwise schedulable deadlines due to the imposition of fixed priorities, but leaves other problems, most notably transient overloads. Transient overloads occur not only as a result of application and operating system design decisions, but also as a consequence of normal system activity, including interrupt processing, DMA cycle stealing, or blocking on semaphores[10]. The deadline scheduler provides no reasonable control over the choice of which deadlines must be delayed in an overload, leading to unpredictable failures and resulting in an impact on reliability and maintainability.

The value of a real process completion is well modeled by a step function only in those few cases in which in which there is no longer any value in completing the process after its deadline, such as computing a control parameter for a manufacturing step which has already been completed. In many actual systems, the value of completing a computation after its deadline may rapidly decay but remain positive (e.g., being late on an aircraft navigation update may result in a loss of positional accuracy, while missing it altogether would merely exacerbate the loss unless it became so late that it impacted making the next update), or completing a computation before its deadline may be more or less desirable than completing it exactly at its deadline (e.g., a satellite orbital insertion burn computation, which must occur within a small time "window" around an optimal time). As we show experimentally, process scheduling in the presence of a (possibly transient) overload shows particularly well the difficulty in producing a reasonable schedule with respect to different process' value functions. Determining an appropriate value for a given process consists not only of simply assigning a priority, but rather of defining the system value of completing it at any time. This value is normally defined by the physical application environment which the system must serve.

1.4. Evaluating a Real-Time Scheduler

Given that the primary characteristic of a real-time system is that correctness is determined not only by *what* is done, but *when* it is done, we propose to use a representation of a process completion value to measure the algorithms commonly used or considered for use in a real-time system. In particular, we will simulate a real-time system in which each process has a value expressed as a function of time which defines the value to the system of completing that process at any time, and we will "reward" the system with the value determined by that function when the process terminates. A set of processes which arrive at stochastically determined intervals (regular intervals for periodic processes and Poisson arrivals for aperiodic processes) and compete for a processing resource in a symmetric multiprocessor will then be executed for a given period of time. The sum of the resulting process values for all processes requested during the execution period will then provide our metric for determining the performance of each scheduling algorithm under several conditions of loads, and with several different value functions. This metric measures the long-term (relative to the individual process computation times) performance of the system with respect to its support of the application-defined value of meeting time constraints, a direct measure of the application real-time support provided.

2. Time-driven Scheduling Model

We define a computational model consisting of a set of preemptible processes P resident in a computer with a single shared memory and one or more processing elements. Each process p_i has a request time R_i, an estimated computation interval C_i, and a value function $V_i(t)$, where t is a time for which the value is to be determined[7]. Figure 2-1 illustrates these process attributes for a hypothetical process whose value function is linearly decreasing prior to a critical time D_i with an exponential value decay following D_i. The process illustrated depicts a process which has been dispatched shortly after its request time and which has completed prior to its critical time without being preempted.

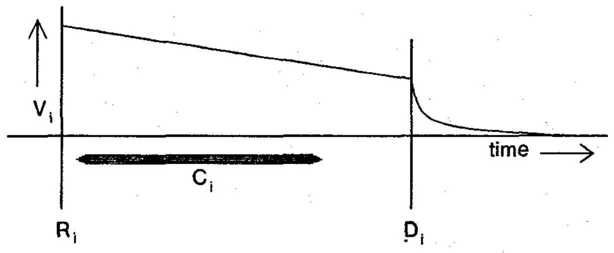

Figure 2-1: Process Model Attributes for Process i

$V_i(t)$ defines the value to the system for completing P_i at time t. The nature of V_i is determined by the range of scheduling policies supported by the operating system, particularly with respect to the handling of a processor overload in which some critical times cannot be met. The value function could come from either or both of two sources: the process implementer who understands the requirements of the internal environment as it affects that process, and the system architect who is aware of the relationship between that process and the overall system, including its external environment. For the purposes of this paper, we assume that all sources of value information are reflected in the value functions used.

We note that the existence and importance of a deadline for p_i is dependent on nature of its value function. The value function can be said to define a critical time (also called a deadline if the function is a step function) only if it has a discontinuity in the function or its first or second derivative. Functions of this type are characterized by a relatively rapid change in value with respect to time, with the deadline case only an extreme example. For example, the process illustrated in Figure 2-1, has a critical time defined by the discontinuity in its first derivative at D_i. For convenience in the discussion of the simulator, we will actually refer to a critical time as the time axis origin relative to which the value function for a process will be computed, even if the function has no discontinuity as described above.

The use of value functions as described in this model allows us to describe both hard and soft real-time environments, and, in particular, allows us to evaluate systems which mix deadlines of both types in a single system. Since the value functions used in this model may or may not explicitly define either a critical time or a deadline, and the critical time may not actually constitute a deadline, in the remainder of this paper we will avoid the use of the word "deadline", which implies the step value function. Hence, we will refer to the time of the value function discontinuity, if any, as its critical time.

The request time R_i has been defined in our simulation (see Section 3) as an arbitrary time at which P_i has been requested to be executed. The significance to the scheduler as we have defined it is that the process is not schedulable prior to P_i. Following P_i, it remains schedulable until one of the following conditions has occurred:

- It has completed,

- Its value function has become zero or negative.

We note that the value function may be negative at R_i, not rising above zero until a later time (see Section 3 for an example of such a function).

Thus, as viewed by the process scheduler, the request time R_i for a process may be either a future or a past time. If R_i is a future time, the process is not currently schedulable, but its attributes may be considered in the current computations of load from which current scheduling decisions are made.

The computation time C_i is a random variable representing the expected execution time to process P_i (estimated time remaining if P_i has already begun processing), not including system overhead or preemptions. The source of this value and its distribution for a real operating system would be either the programmer or an actual measurement by the system itself (see Section 4), but here we will simply assume that it is available to the scheduler. Clearly, the value function can be seen as a definition of application scheduling policy, and although in a real system not every policy would be subsumed in a single set of value functions, in this paper, we will assume that the value function completely defines the policies to be implemented by the scheduler. The relationship between our value functions and the global scheduling policy is a subject of continuing research in this effort.

Given the potentially diverse set of value functions which could be produced in a given system, we will show that none of the normally used scheduling algorithms can consistently produce a good schedule as the processing resources approach saturation (see Section 3). As an example, see Figure 2-2 for an example of four processes in a single processor with value functions, for which the best choice of an execution sequence is non-trivial. The figure shows the four value functions, and a potential scheduling sequence such that each completes with a high value. We will describe later the significance of such value functions in the discussion of the experiments conducted.

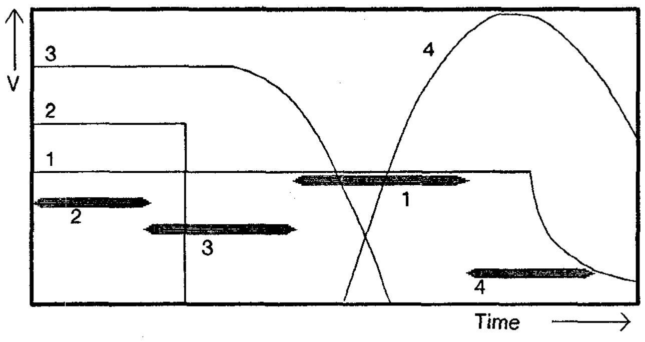

Figure 2-2: Four "Typical" Processes with Value Functions

This model encompasses both periodic and aperiodic processes in that any single execution of a periodic process can be described as shown above in exactly the same way as for a non-periodic process. The only difference is that the future request times for a periodic process are known in advance. We allow overlapping executions by allowing multiple instances of a process to be simultaneously schedulable (we assume that processes are reentrant).

It should be noted that precedence and consistency (e.g., processes involved in mutual exclusion) relations are not addressed in this model at this time, since it has been assumed that the scheduling algorithm will consider only processes which are ready to be executed and for which a request time has been determined. If one process is dependent upon completion of another, the second process' request time will not have been determined until the first process has been completed.

3. Simulation Results

3.1. Real-Time Scheduling Simulator

To provide an environment in which various scheduling algorithms can be evaluated with respect to their performance in generating a high total system value, a simulator has been constructed which provides an operating system environment in which each of the scheduling algorithms can be executed and evaluated.

The value functions to be used by the simulator are not competely general forms, but rather are limited to a specific form possessing characteristics making the expressiveness of the value very flexible while allowing the analysis of the resulting functions to be as tractable as possible. We define the value function in two parts; the value prior to the critical time and the value after the critical time, providing for a discontinuity at the critical time if desired by the application designer. For each of these two parts, five constants are used to define the value function relative to the critical time using the expression:

$$V_i(t) = K_1 + K_2t - K_3t^2 + K_4e^{-K_5t}$$

The simulator first uses a statistical model of a "typical" real-time processor load to generate a set of processes, including both periodic and aperiodic processes. The statistical model defines a real-time workload based on experience with several actual real-time systems with which this author is familiar. The processor load generated can be varied to simulate both heavily loaded systems and relatively lightly loaded systems. The nature of this statistical load, including the process characteristics, the simulated hardware characteristics, and the value functions, can be easily modified during the experimentation allowing us to evaluate a number of systems with different characteristics. It is also possible to impose an additional "spike" load in which the aperiodic arrival rate dramatically increases at certain periods during the simulation, allowing us to determine scheduler performance during transient overloads. Our test process sequences will emphasize the overloaded condition, since that is the condition of primary interest.

Once a set of processes has been constructed, the simulator "executes" the processes, applying a statistical model to generate the actual request times and computation times for each process. Execution consists of iteratively calling the selected scheduler for its decisions, "executing" the selected process, updating the clock to the next important time (e.g., time of arrival of a new process), then asking the scheduler for its next decision.

3.1.1. A Set of Classical Algorithms

The simulator has the ability to make multiple runs sequentially using an identical set of processes with identical values and request times, using various "classical" algorithms. These "classical" algorithms include:

1. **SPT**. At each decision point, the process with the shortest estimated completion time is executed.

2. **Deadline**. At each decision point, the process with the earliest critical time (interpreted by this algorithm as a deadline) is executed.

3. **Slack**. At each decision point, the process with the smallest estimated slack time (elapsed time to the critical time minus its estimated completion time) is executed.

4. **FIFO**. At each decision point, the process which has been in the request set longest is executed.

5. **Random**. At each decision point, a process is chosen (with uniform probability) from the request set and executed.

6. **RandPRTY**. At each decision point, the process with the highest fixed priority is executed. In the runs reported, this fixed priority was chosen to be the same as the highest value reached by the corresponding value function.

The simulator keeps a large number of statistics on each run, including such parameters as the number of tardy processes, the maximum lateness, the average lateness, and the total value accumulated. This data is saved and reported for each real-time run made in a simulation sequence, and is reported in a statistical summary at the end of the simulation, resulting in the data provided below.

3.1.2. A Pair of Interesting New Algorithms

In addition to these algorithms, two experimental algorithms have been implemented, and have produced some initially promising results which we include in the experiments demonstrated here. An important part of our research effort consists of the further development of algorithms such as these, and the concepts behind them, as they apply to a real-time system.

In these initial algorithms, we take advantage of three observed value function and scheduling characteristics:

1. Given a set of processes (ignoring deadlines) with known values for completing them, it can be shown that a schedule in which the process with the highest value density (V/C, in which V is its value and C is its processing time) is processed first (i.e., a Value Density Schedule) will produce a total value at every point in time at least as high as any other schedule.

2. Given a set of processes with deadlines *which can all be met* (based on the sequence of the deadlines and the computation times of the processes), it can be shown that a schedule in which the process with the earliest deadline is scheduled first (i.e., a Deadline schedule) will always result in meeting all deadlines.

3. Most value functions of interest (at least among those investigated at this time) have their highest value occurring immediately prior to the critical time.

Some of the implications of these observations are:

- If no overload occurs, the deadline schedule will have all deadlines met, and no higher total value will be possible.

- If an overload occurs, and some processes must miss their deadlines, the Value Density Schedule would produce a high value, but might miss some deadlines which could otherwise have been met.

Our first algorithm (called the BEValue1 algorithm in the experimentation description, section 3.2) exclusively uses observation 1 above, and is therefore a simple greedy algorithm scheduling first the process with the highest expected value density. This algorithm actually performs reasonably well in many cases in which the value function is a step function or rapidly decreasing following the critical time, in spite of the fact that it makes no explicit use of the critical time itself. The critical time does, of course, enter the algorithm through the expected value computation, which uses the value function and the assumed computation time distribution to compute the expected value. This algorithm fails most notably in step function situations in which no overload is present and a number of processes with close deadlines are in the request set. In this case, the processes with high value density will be run, quite possibly preventing other processes from meeting their deadlines even though all deadlines could have been met.

Our second algorithm is a modification of the simple deadline algorithm, attempting to remove its most important failing; in an overload the deadline scheduler will give priority to processes whose deadlines cannot possibly be met, delaying other processes which could still meet their deadlines. Therefore, after computing the probability of an overload, we choose processes with low value density as candidates for being removed from an overloaded deadline schedule until a deadline schedule is produced which has an acceptably low probability of producing an overload.

This algorithm (called the BEValue2 algorithm in the experimentation description, section 3.2), starts with a deadline-ordered sequence of the available processes, which is then sequentially checked for its probability of overload. At any point in the sequence in which the overload probability passes a preset threshold, the process prior to the overload with the lowest value density will be removed from the sequence, repeating until the overload probability is acceptable. This algorithm seems generally to outperform BEValue1, since it always meets deadlines as long as no overload occurs, and transitions gradually into the same performance as BEValue1 as an overload condition worsens. In this algorithm, modifications to the overload probability threshold can significantly affect its overload performance; at extremes, a threshold of 100% results in a pure deadline schedule, while a probability threshold of 0% results in a BEValue1 schedule. For the experiments described here, a threshold of 40% is used. Optimizing this threshold as well and the other critical values used in the heuristics from which BEValue2 is constructed are goals of our continuing research in this area.

3.2. Experiments Executed

To test our hypothesis that the efficacy of a scheduling algorithm in a real-time system can be measured with respect to the key distinguishing characteristic of such systems, namely that completing a process has a time-varying value to the system, we have generated a set of simulation experiments using several well-known scheduling algorithms. As with any such set of measurements, this set is necessarily incomplete, but is intended to illustrate some of the characteristics which would be observable with systems under several potential conditions.

Each experiment described here was run on a simulated, symmetric, shared-memory multiprocessor, with the number of processing elements varying from one to four (thus testing the schedulers under four load levels). All processes were considered to be fully preemptible and the effects of context switching and scheduling overhead were neglected. Thirty-six processes were selected, each with an individually determined stochastic load assumed to be normally distributed and characterized by its mean and standard deviation. The mean loads of these processes were themselves approximately normally distributed across the 36 processes, with a mean of 500 ms. and a standard deviation of 300 ms., with the limitation that the minimum process load was 1 ms. Figure 3-1 lists the actual set of processes, showing for each its process ID, its period (if it is periodic), its load characteristics, and its critical time.

As each simulation progressed, each process from this list was requested at either the periodic rate, if it was a periodic process, or using an exponential interarrival time with a mean of 10 seconds. In addition to these requests, additional aperiodic processes arrived as a "spike" load with a mean interarrival time of 1 second, exponentially distributed, every 10 seconds, with the "spike" lasting 200 ms. This simulated set of requests continued for 60 seconds of elapsed time, at the end of which the requests ended and the unfinished processes in the request set, if any, were completed by the multiprocessor, terminating the simulation when the queue was empty. At the completion of each process, its value was computed and the values from all completed processes are summed, producing a final total value over the 60 second request interval. No value was accrued for processes not competed (i.e., aborted). As the list in Figure 3-1 shows, four of the processes were periodic; the total system load from these four processes represented about 42% of the total process load. This simulation resulted in the arrival of about 305 processes, including both periodic and aperiodic processes.

Four value functions, illustrated in Figure 3-2, were used in separate executions to compute the total value generated by each of the scheduling algorithms under differing conditions of perceived system value.

The first (V1), showing a constant value prior to the critical time followed by an exponential decay after the deadline, represents a case such as a vehicle navigation problem in which a position update must be completed by a given time each cycle to provide a specified precision, but if late, the resulting error can be minimized by making up the computation, assuming the next navigation cycle is not overrun. As the next cycle approaches, the value of running the previous one becomes negligibly small. The second (V2) represents a hard deadline process, in which there is value to the system for completing it only if it precedes a

ID	Period	Mean Load	Load σ	Critical Time
1		0.474	0.058	0.683
2		0.045	0.009	0.128
3		0.580	0.159	1.746
4		0.516	0.141	1.267
5		0.492	0.132	1.069
6		0.133	0.034	0.193
7		0.627	0.129	1.402
8		0.251	0.044	0.444
9		0.859	0.271	1.894
10		0.520	0.181	0.762
11		0.574	0.080	1.405
12		0.217	0.027	0.674
13		0.920	0.166	2.752
14		0.737	0.135	1.749
15	1.731	0.247	0.069	0.618
16		0.515	0.067	1.159
17		0.748	0.345	2.160
18		0.696	0.248	1.390
19	5.762	0.782	0.229	1.260
20		0.555	0.088	1.422
21		0.805	0.197	2.490
22		0.051	0.003	0.148
23		0.684	0.166	1.583
24	11.119	0.827	0.215	1.932
25		0.408	0.133	1.254
26		0.001	0.000	0.002
27		0.894	0.185	2.126
28		0.754	0.166	2.119
29		0.387	0.099	1.166
30		0.658	0.141	1.606
31		0.249	0.103	0.562
32		0.459	0.112	1.409
33		0.650	0.100	1.788
34		0.498	0.152	0.714
35	3.663	0.245	0.045	0.618
36		0.130	0.041	0.322

Figure 3-1: Simulation Experiment Process List

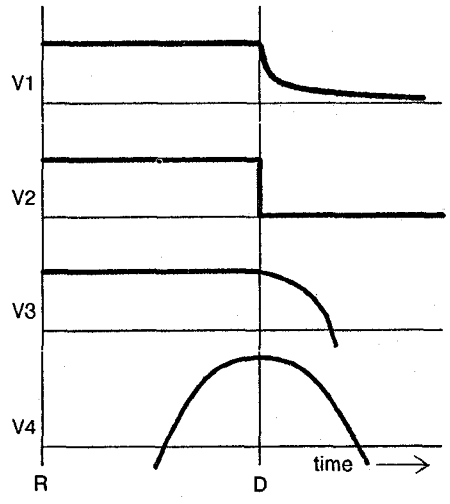

Figure 3-2: Example Value Functions

fixed deadline, such as for certain sensor inputs, in which the value to be read is no longer present if a fixed time interval is exceeded. The third value function (V3) is constant prior to the deadline, but drops off parabolically after the deadline, representing a process similar to the hard deadline above, but with a softer deadline as long as the process is not excessively late (in this experiment, the value function drops past zero at about 516 ms. following the critical time). The fourth function (V4) is parabolic both before and after the critical time, representing a process whose completion should be delayed until after some arbitrary time, such as a satellite launch, in which the desired orbit cannot be reached prior to a given time or after a later time (in this experiment, these functions pass zero approximately 516 ms. before and after the critical time).

In an actual system, it would be expected that each process in a given application would have a distinct value function, but we have used identically shaped value functions for every process in the run to isolate the effects of multiple value function shapes on scheduling quality. For each of the four value functions chosen, four experiments were conducted, with 1, 2, 3, and 4 processing elements each, representing average process loads of 246%, 123%, 82%, and 61%, respectively, and each simulation was repeated 15 times, averaging the computed parameters. For each algorithm in each experiment, several parameters were computed:

- *Total Value(V1, V2, V3, V4)* -- The sum of each of the four value functions for each process computed at its completion time, averaged over the 15 runs.
- *Mean Preemptions(MP)* -- The number of times a running process is preempted by a more critical process.
- *Mean Lateness(ML)** -- The difference between the actual process completion time and its critical time, averaged over the 15 runs. This value is negative if the process completes prior to its critical time, and positive after the critical time.
- *Maximum Lateness(MaxL)* -- The largest lateness value of the entire 60 second run, averaged over the 15 runs for which each simulation was repeated.
- *Mean Tardiness(MT)* -- The average of the non-negative latenesses over the 15 runs.
- *Number of Tardy Processes(NTP)* -- The total number of processes completing after their critical times, averaged over the 15 runs.

and the results of this measurement for the standard algorithms are graphically represented in Figure 3-3, while the more specialized BEValue1 and BEValue2 algorithms are represented in figure 3-4. In these figures, each circle represents a particular scheduling algorithm executing under a particular load. Within each circle, each line represents one of the measurements described above, with a longer line indicating that the corresponding measurement was one of the best for that load value, while a short line indicates that the corresponding measurement was relatively poor. Although these measures are all reported for each set of runs, the most important measures from our perspective are the four value function results, V1, V2, V3, and V4.

*Value functions resulting in aborting processes (i.e., all the value functions used in the experiments reported here except the exponential decay function [V1]) produce meaningless values for the lateness and tardiness, so the reported measurements for these are included only for runs which result in no abortions.

Processes whose value has passed below zero are aborted, and the resulting system value is zero, reflecting the fact that they produce no contribution to the system performance. This abortion occurs for all algorithms tested.

3.3. Experimental Results from Standard Scheduling Algorithms

There are a few observations which can be made from these measurements across all runs. It should be noted that in no case did the FIFO scheduler or the random scheduler generate the best schedule, although they produced acceptable schedules when the system was sufficiently underloaded; it is clear that neither of these would be the algorithm of choice unless we were guaranteed that an overload condition could not occur. Both of these algorithms are used extensively in certain types of actual systems; FIFO is used for many message passing schedulers, and random scheduling is effectively used for some contention bus communications links (e.g., Ethernet). Similarly, the fixed priority scheduler performed inconsistently, particularly as the overload condition became more pronounced. Thus, of these algorithms, the choice would be between the SPT and Deadline scheduling algorithms, neither of which are widely used in existing real-time systems.

With respect to the observed value variations among the different processor loads, the first observation is that, except for the fourth value function (V4) runs (parabolic value functions both prior to and following the critical time), the scheduling function makes little or no difference when the system is lightly loaded. Although in the underloaded case the deadline scheduler generally shows the best performance in most of the relevant measured parameters, the difference between the best and worst among the six algorithms with respect to total value is within about 5% on value functions V1, V2, and V3. As expected, we note that the insensitivity to the algorithm is most pronounced at the average load of 61%, and less so at 82%, where the preference for the deadline and slack time schedulers are more pronounced (and the difference between the best and worst total values) is about 12%. This result confirms our intuition that the scheduling algorithm used for a suitably underloaded system is not usually an extremely critical factor. The case with value function (V4) is radically different in that the value functions indicated that the processes should have had their executions delayed to achieve acceptable values, but none of these algorithms took this effect into account. Such an effect is typically overcome in existing real-time systems by the application processes themselves introducing delays in their processing, and relying on priority to force the delayed process to execute properly in its value function "window".

The picture becomes more complex as the load increases to an overload condition. At 123% overload, the deadline scheduler is beginning to have considerable difficulty with V1, V2, and V3, and the SPT scheduler seems to be performing somewhat better. Even the fixed priority scheduler is doing better than the deadline scheduler. This effect, while perhaps not intuitive, can be explained by noting that the SPT algorithm is designed to complete the largest possible number of processes in a given time interval, increasing the likelihood that the short processes will complete prior to their deadlines and resulting in a greater number of processes doing so, therefore producing a higher total value. The deadline scheduler, while ensuring that deadlines will be met if there is sufficient processing capacity, fails to satisfactorily handle an overload due to its propensity for

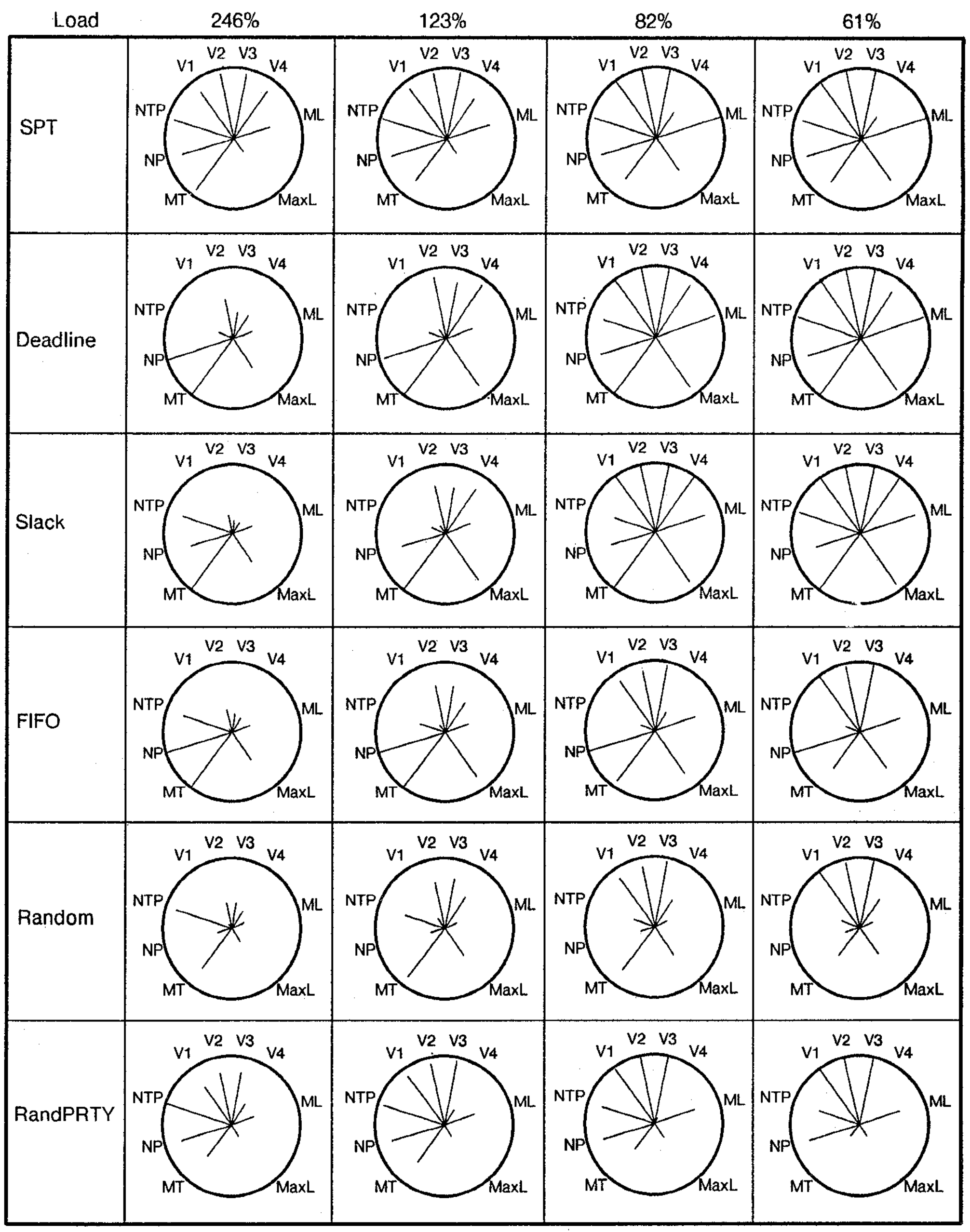

Figure 3-3: Standard Algorithm Experiment Results

giving priority to processes which are about to miss their deadlines over those whose deadlines are still to come.

3.4. Experimental Results from Value Function Algorithms

Of the two experimental value function algorithms described above, it is clear that the BEValue2 scheduler is a clear winner as shown in Figure 3-4, plotted with identical scales as Figure 3-3, using the identical value functions and loads.

It outperforms all other tested algorithms in all the overload runs by significant margins, and shows a consistency of performance which none of the other algorithms can approach. Like the others tested, it had some difficulties with the V4 function, since it, too, takes little account of the fact that the processes needed to be delayed to achieve a high value, and this is one of the critical areas in which further research will be performed. These results should be viewed as preliminary at best, but it is certainly clear that it has some important characteristics which would be needed in a value function scheduler.

The BEValue1 scheduler performance was, like many of the other algorithms, much more variable, depending on the value function and load. Because it is a "greedy" algorithm, preferring to pick up value early rather than wait to get a higher value, it misses many opportunities to meet time constraints.

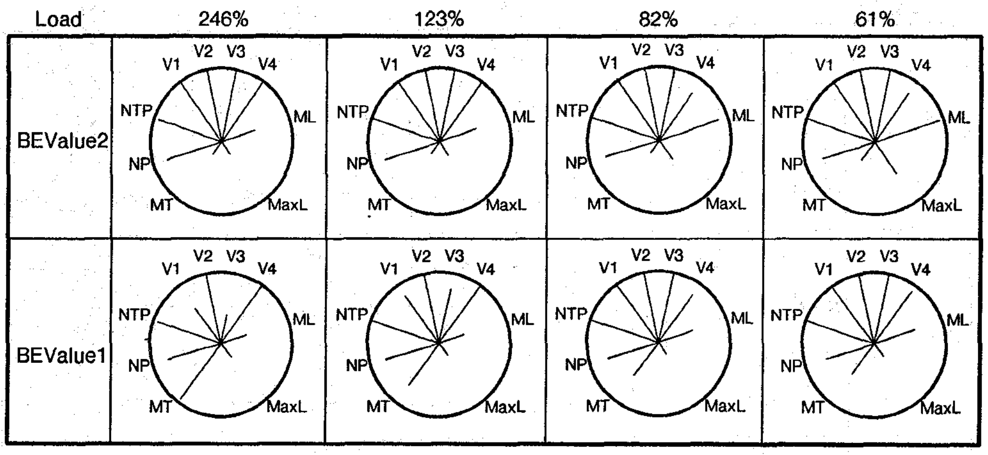

Figure 3-4: Value Function Experiment Results

3.5. Experimentation Summary

It is clear that the shape of the value function and the processing load level has a great effect on which algorithm is better for the total system performance when the system is overloaded. This experimentation seems to indicate that, with improvements, a scheduler which overtly uses the application-defined system value to handle scheduling should, at least in principle, provide a consistently better schedule. In fact, such improved performance, particularly in overload conditions, does result from our two experimental scheduling algorithms, although our analysis on them is by no means complete at this time.

4. Real-Time Operating Systems Interface

There are many potential approaches to utilizing a time-driven scheduling model in a real-time operating system. Although traditional real-time operating systems provide a simple set of time control primitives (e.g., GetTime, Schedule, Timeout), no system primitives are available to explicitly express *request* R_i, *critical* D_i, *estimated computation times* C_i, or *a value function* $V_i(t)$ for each process. Furthermore, a real-time operating system should be able to support a primitive to modify the value function for a process or a set of processes during run time, as well as primitives to specify a system-wide scheduling policy. In this way, an application designer can set up and modify a suitable scheduling policy under various system conditions.

For the purpose of describing these operating system primitives, we assume that primitives to create and kill processes already exist. We express each primitive as if it were implemented as a procedure in a high-order language such as C or Pascal.

4.1. Time Control Primitives

The arguments to these operating system primitives communicate the information needed to implement this model, but an important issue is the structure of the information to be passed to the operating system. In the following primitives, the structure of the passed information has been designed to encourage internal consistency. Using a single primitive to define each parameter would be extremely flexible, but a user might inadvertantly set mutually inconsistent parameters. Thus, the system should provide a simple way to define a consistent set of parameters.

The *Delay* primitive

```
DateTime = Delay(DelayTime, CriticalTime,
                 CompTime, SetFlag)
```

blocks the requesting process a specified length of time (e.g., in microseconds), then returns the current date and time when the delay is completed and execution has resumed. This primitive would also be used to specify the critical time D_i (see Section 2) and an estimate of the amount of processing time that will be required to reach the critical time** C_i, as well as to mark the arrival of the process at the critical event for which the critical time was established (while optionally defining the next critical time) by setting "SetFlag" value. If a value of "DelayTime" is not positive, the system would not block the process, and similarly the estimated execution time and critical time would not be changed if its value were not positive.

For instance, a simple command to alter the critical time can be defined by using the *Delay* primitive:

```
Delay(0, CriticalTime, 0, TRUE)
```

4.2. Periodic Processes

While the *Delay* primitive provides time dependent processing control at runtime, periodic repetitive processing can be specified at process creation time.

One way to describe a periodic process is by using optional arguments in a *CreateProcess* primitive:

```
pid = CreateProcess(ProcessName, InitialMsg,
                    PERIODIC, DelayTime, Period)
```

The "CreateProcess" primitive would create a new instance of a process at a specified node. "InitialMsg" indicates an initial message to be immediately sent to the new process and "PERIODIC" specifies that the new process will be periodic. "DelayTime" sets the first request time for the new process and "Period" is the indicated period.

**A system may also estimate this processing time based on observations of earlier executions.

4.3. Scheduling Policies

For a practical real-time operating system, it is necessary to provide a mechanism to express the application-defined scheduling policies needed to implement our scheduling model. The system should be able to modify these policies at runtime in order to take advantage of the flexibility of the model. The value function of the time-driven model could be provided as an executable function by the user, or the user could express various scheduling policies by defining a dynamic set of values $K_1, K_2, ..., K_{10}$ (see Section 3), so that a client need only pass these values to define a new value function $V_i(t)$:

SetValueFunction($K_1, K_2, ..., K_{10}$), or

SetValueFunction(V_i)

The first primitive is acceptable as long as the application does not require new value function which cannot be expressed by the fixed value function. The second primitive has greater flexibility in terms of passing an arbitrary value function from a client to the system, but, since the actual code of the value function would then be provided by the client, the scheduler would be required to compute its value in user's address space every time it is needed. Thus, this scheme may introduce a significant overhead to the scheduler and, in addition, such flexibility would make it very difficult to develop a future scheduler able to use a value function explicitly to make scheduling decisions.

Our approach is to use the notion of "policy/mechanism" separation[11], to increase the flexibility and reduce scheduling overhead. The basic idea is to create a user-definable system module, called *policy definition module* which consists of a *policy body* and a set of *policy attributes*. A policy definition can be placed in a kernel, a system process or a client process as needed to control the system overhead. A *policy definition descriptor* would indicate the location of the policy definition and the information related to the policy body and attribute set.

For instance, a policy definition module could be defined and the policy attribute for a specific policy could be set using the following primitives:

SetPolicy(PolicyName, PolicyDefinitionDescriptor)

SetAttribute(PolicyName, AttributeName,
AttributeValue)

The *SetPolicy* primitive would bind a user-defined policy definition module to the operating system. "PolicyName" indicates a symbolic name of policy definition module and "PolicydefinitionDescriptor" points to a data structure which describes the policy definition module's location and its properties. The *SetAttribute* primitive sets the value of a specified attribute for the specific policy module.

For example, the following calls could be used to replace the the *SetValueFunction* primitive above.

SetPolicy(SCHEDULE, MyPDD)

SetAttribute(SCHEDULE, "VALUEFUNCTION",
$K_1, K_2, ..., K_{10}$)

This *SetPolicy* primitive would bind a scheduling policy module described by "MyPDD" to the operating system. In the case of the scheduling policy module, we would place the proposed value function at the kernel level. The *SetAttribute* primitive then defines a shape of the value function by giving the values $K_1, K_2, ..., K_{10}$. We believe that such a proposed value function is rich enough to express a powerful set of scheduling policies and we have reduced the scheduling overhead while increasing scheduling flexibility by not attempting to pass the arbitrary function $V_i(t)$ itself.

5. Summary

Process scheduling is a frequently overlooked determinant of real-time performance. It is our contention that time value of process completion, even though it is the primary characteristic of a real time system, has been neglected in measuring real-time performance. We have illustrated some of the effects of varying the scheduling algorithms, particularly in the presence of an overload condition, relative to several value function characteristics.

This use of value functions to describe the performance of existing scheduling algorithms represents a first step in our goal of eventually producing a scheduler along the lines of BEValue2 which will explicitly use such value functions to make scheduling decisions[4]. Such a scheduler would attempt to enhance the overall system value by ensuring that processes providing the highest value to the system are favored in the event that some processes must be delayed or aborted due to a (possibly transient) overload. This would provide a level of predictability to overload processing in a real-time system which is not generally available at present, and should therefore allow more effective use of the available resources, lowering system cost.

This paper has presented a preliminary overview into a significant research effort recently initiated within the Archons project. The goals of this work include further understanding the scheduling problem in the presence of user-defined value functions, determining the heuristics necessary to create a tractable value function scheduler, understanding the sensitivity of such a scheduler to the parameters defined by its heuristics, and analyzing the decisions made by those heuristics. Additional measurements with modifications to the scheduling algorithms could be made providing for such capabilities as defined lower bounds for use in process abortion when the value function drops below zero, as well as studies of the effects of varying the heights and shapes of the value functions among the processes. The methods for determining which value function to use for specific types of devices and algorithms need to be studied, the implications of value function scheduling on the specification of scheduling policy, methods of decomposing value functions for a number of processes cooperating to perform a single application function need to be determined, and the use of value functions in the presence of process-to-process dependencies and mutual exclusion must be investigated. Following the initial effort, this research will be extended by the Archons project for process scheduling on a distributed system.

Acknowledgements

Professor Jensen created the initial concepts and development of continuous time-valued scheduling for real-time systems. Mr. Locke, in his Ph.D. thesis research, is further developing the model, defining the required heuristics, and evaluating them through simulation. Dr. Tokuda is responsible for the incorporation of these ideas into a scheduling facility for the Archons project's ArchOS operating system.

The authors are greatly indebted to the many members of the Archons project, especially Professor John Lehoczky and Dr. Lui Sha, who provided considerable assistance in the definition of these concepts, as well to the Archons sponsors.

References

1. Jensen, E. D., *Distributed Control,* Springer-Verlag, 1981, pp. 175-190.

2. Jensen, E. D., "Decentralized Executive Control of Computers", *Proceedings of the Third International Conference on Distributed Computing Systems*, IEEE, October 1982, pp. 31-35.

3. Jensen, E. D., "ArchOS: A Physically Dispersed Operating System", *IEEE Distributed Processing Technical Committee Newsletter*, June 1984.

4. Locke, C. D., "Best-Effort Decision Making for Real-Time Scheduling", Ph.D. Thesis Proposal, Carnegie-Mellon University, Computer Science Department

5. Conway, Maxwell, and Miller, *Theory of Scheduling*, Addison-Wesley, 1967.

6. Garey, M. R.; Johnson, D. S., *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W. H. Freeman, San Francisco, 1979.

7. Jensen, E. D., Private Communication based on his Honeywell Systems and Research Center technical report on this topic for the U.S. Army Ballistic Missile Defense Advanced Technology Center.

8. Mok, A. K., *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment,* PhD dissertation, Massachusetts Institute of Technology, May 1983.

9. Liu, C. L.; Layland, J. W., "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *Journal of the Association for Computing Machinery*, Vol. 20, No. 1, January 1973, pp. 46-61.

10. Lehoczky, J. P.; Sha, L., publication to appear.

11. Wulf, W. A.; Levin, R.; Harbison, S. P., *HYDRA/C.mmp: An Experimental Computer System,* McGraw-Hill, Inc., 1981.