CMU-CS-86-134

Best-Effort Decision Making for Real-Time Scheduling

C. Douglass Locke

May 10, 1986

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

CMU-CS-86-134

Best-Effort Decision Making for Real-Time Scheduling

C. Douglass Locke

May 10, 1986

Computer Science Department
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

This work was supported in part by the USAF Rome Air Development Center under contract number F30602-85-C-0274, the U.S. Naval Ocean Systems Center under contract number N66001-83-C-0305, the U.S. Navy Office of Naval Research under contract number N00014-84-C-0734, and the IBM Federal Systems Division. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of RADC, NOSC, ONR, IBM, or the U.S. Government.

Table of Contents

List of Figures

Abstract

The objective of this research was to develop a new approach to making process scheduling decisions in real-time computer systems for such demanding control applications as spacecraft control and industrial factory automation. The approach used applies not only to scheduling process execution time, but also to scheduling other resources such as physical and logical messages on a communications medium.

Real-time systems differ fundamentally from non-real-time systems in several ways; most significantly in that the performance metric of interest is not throughput, but rather "response time" in the sense of completing the most critical tasks according to application-defined time constraints. Stated another way, we define a real-time system to be one in which there is an application environment-defined value to the system for completing a process at a particular time, and in which assertions about this value as a function of time form an essential part of the correctness of the computation.

In response to the proliferation of expensive, fragile systems that are prone to unpredictable behavior and failure under overload conditions, we have defined a scheduling algorithm which explicitly utilizes the time-varying completion value of a process to determine the sequence in which a set of competing processes should be executed. Using these functions, the scheduler attempts to make scheduling decisions which maximize the total value to the system. Processes are assumed to be independent and preemptible, and to exhibit a stochastic processing time. The computational environment consists of one or more processors executing a single set of processes.

Having evaluated this scheduler in a real-time system simulator, we conclude that in a heavily loaded real-time system (either a transient or persistent load), this time-driven, value-function controlled algorithm provides a consistently high total value for a large number of task sets. The algorithm produced this high total value under extremely widely varying load conditions, process execution times and distributions, execution-time knowledge, numbers of processors, periodic and aperiodic processes, either steady or intermittent load levels, and with a wide variety of either homogeneous or non-homogeneous value functions.

Acknowledgements

I would like to express my appreciation to a number of people without whose assistance this work would not have been successfully concluded.

The initial concepts were inspired by Professor Doug Jensen who, as principal investigator of the Archons project, worked tirelessly in spite of health problems, to knit together this research and the other efforts on the project into a coherent whole. Professor John Lehoczky, another important member of the Archons project, consistently acted as a critical sounding board for the ideas which eventually took shape in this effort, frequently identifying holes in an argument, or weaknesses in the statistical foundations of this work.

I also owe a large debt of gratitude to the other members of the thesis committee; Dr. Herbert Simon, whose insightful questions triggered several of the investigations reported herein, and Dr. Rick Rashid who provided encouragement at key points along the way.

Dr. Hide Tokuda, while not a member of the thesis committee, provided many helpful suggestions from the inception of this research through the final drafts. As a friend, his encouragement and quiet thoughtfulness throughout this effort will be long remembered. In addition, many of the other members of the Archons team listened to my ideas and provided helpful insights, especially Dr. Lui Sha, Dr. John Stankovic, and Mr. Ray Clark.

In many practical ways, this work would not have been possible without the support of the IBM Federal Systems Division who made it possible for me to be a part of the CMU Computer Science community, and particularly my IBM manager and friend, Dr. Rodger Fritz, who never admitted to any doubt that this effort would be successful.

Finally, I must acknowledge with a great deal of love and gratitude the unflagging dedication, perseverance, and support of my wife, Kathy, who took over many of my responsibilities so that I could pursue this goal, and who now shares this achievement as an equal partner. This success would certainly not have been possible without her. There are also four very special people in my life who have been greatly affected during this extremely formative period in their lives; David, Robert, Jeannette, and Julie are no longer the small children we brought to Pittsburgh. To my parents Carl and Wilma Locke, also, I owe a debt of gratitude in training me from an early age to value a persistent effort in the pursuit of important goals.

Chapter 1
Introduction

This thesis is concerned with the decision making required to define a sequence of processes in a real-time system; our principal objective is to develop a new approach to scheduling processes for execution in real-time computer systems. The issues with which we deal range from the decision process itself to the complexity and time constraints of the real-time environment.

In this chapter, we introduce the research background and motivation for this work, discussing general decision making concepts and their application to real-time operating system resource allocation decisions. This chapter also describes the specific problem we have addressed, that of real-time process scheduling, and our approach to solving this problem.

1.1. Background

1.1.1. Decisions

Making a decision can be described as a sequence of actions which is essentially the same whether the process is carried out by a human or by a machine:

1. *Acquisition of knowledge.* There are many types of knowledge concerning the decision to be made, including estimation of the state of the decision environment, parameters affecting each factor in the decision, the rules by which inferences may be drawn from the available information, and estimation of the effects on the decision environment which may be expected to result from each potential decision.

2. *Enhancement of the available knowledge.* This includes correlating new information with previous information or information from other sources, filtering (e.g., a Kalman filter), transforming the information to a more useful domain (e.g., spatial coordinate transformations), predictions of future information, and comparisons of current information with the results of past predictions. There are

a number of questions which should be answered here: for example, if some information is inaccurate and/or incomplete, what steps, if any, can be taken to improve it from the point of view of the decision to be made? Are there inference rules which can be applied to the information from different sources which can strengthen the certainty of the conclusions drawn from such inaccurate or incomplete knowledge?

3. *Evaluation of the available knowledge.* Not all information is complete or accurate, and an evaluation of its quality (i.e., usefulness for the impending decision) is essential to its correct interpretation and inclusion in a body of knowledge. The information may be augmented by providing for hypotheses to be generated and tested (e.g., the Hearsay system [Erman 80]), performing extrapolation, etc.

4. *Making the decision.* The decision itself is made by applying a decision process (i.e., an algorithm or a set of heuristics) to the results of the previous steps. In this research, for example, when an operating system schedules processor time, the information regarding the processor load and process requirements must be converted by a scheduling algorithm into an ordered sequence of processes to be run.

1.1.2. Operating Systems Decisions

Making decisions is not only a pervasive human activity, but is a critical part of the management of resources in a computer operating system. Both humans and computers make decisions at many levels of abstraction; for computers, typical decision levels range from low level storage allocation decisions to relatively complex application level decisions made by expert systems in such areas as medical diagnosis and robotic visual scene analysis. The research described here constitutes a contribution to the science and engineering of an important aspect of decision making in resource allocation performed by a real-time operating system on behalf of a real-time application such as a military command and control system or an industrial supervisory process control system.

Making decisions within an operating system is an especially difficult problem. For an expert system, for example, the most important criterion is the quality of the decision rather than its time constraint; the processor on which the expert system is running can be dedicated to making these decisions. A computer operating system, on the other hand, must make its decisions using the same resources which it is trying to allocate, thus decreasing the resource availability. This means that the expenditure of these same system resources by the operating system itself must be commensurate with the value of the remaining resources to the application; the management of resources is not an end unto itself. Clearly, if the

operating system uses excessive processor time to make nearly optimum scheduling decisions using all available information, it would have been better to use a purely random scheduler which makes decisions using no information at all.

Decisions within the operating system share with many expert systems the property that they must frequently be made in an environment of incomplete and inaccurate data. This situation arises because of several factors;

- If a system is distributed, the global system state is changing much faster than it can be communicated throughout the system, so such state information cannot be known by any one processor; this forces global decisions such as those for load balancing and communication routing to use incomplete and inaccurate data.

- Even within a single processor, such information as process CPU requirements can at best be known only stochastically due to varying data and device dependencies. This is also true of current processor load, network traffic, and most other processor state information.

- Since many optimal decision problems, including most scheduling decisions, are known to be NP-complete or NP-hard, they must be made using heuristics based on whatever information is available. Decisions in an operating system, unlike those made in many other environments, cannot be delayed until sufficient information is available to make an optimum decision.

1.1.3. Decisions in Real-Time

In this research, we have studied a particular resource allocation problem with respect to the relevant decisions required for its solution. Although there are several such problems which could have been used in this study, an important problem in operating system resource management for which difficult decisions must be made is that of real-time process scheduling. These decisions are difficult whether the system is resident in a distributed system, a multiprocessing system, or a uniprocessing system. If an optimal solution is desired, the scheduling problem is known to be NP-complete or NP-hard even in some of its simplest forms, in either a uniprocessing or a distributed environment [Karp 72, Sahni 76], so a *best effort* to determine a process schedule is necessary. In this work, we restrict ourselves to logically centralized scheduling in the sense that our decisions will be made by a single decision maker, rather than using a multilateral approach (e.g., a team of nodal decision makers). Although we are also quite interested in multilateral decision making for process scheduling in a distributed system, we leave this as a future extension to our work.

Normal multiprogramming systems (e.g., time-sharing systems) are expected to process multiple job streams simultaneously, so the efficient scheduling of these jobs to maximize throughput and maintain fairness among competing jobs is critical to the performance of the entire set of jobs taken as a whole. The principal difference between normal time sharing systems and real-time systems is the imposition of application-defined time constraints within which responses must be made [Jensen 75]. In real-time systems the management of the processor resources differs fundamentally in that the primary performance objective of such systems is not to maximize throughput or maintain fairness, but instead to perform critical operations *according to a set of user defined critical time constraints*. Our research has been directed toward making a best effort to manage these critical times, especially in the presence of (possibly temporary) processor overload conditions.

This statement of the distinction between time sharing systems and real-time systems may seem counter-intuitive, since real-time applications have been principally characterized by their high speed requirements. It is true that almost all existing real-time system architectures are approached by ensuring that an abundance of processing power is available and is sufficient to handle the worst possible real-time situation, including transient conditions. Approached in this way, the use of high speed equipment is clearly necessary. However, such systems generally are extremely inefficient and therefore extremely expensive, and may still fail to meet important response time requirements, causing failures to occur in unpredictable ways. Such failures are particularly susceptible in real-time systems which may sustain physical damage, causing virtually all exceptions to occur simultaneously (e.g., a vehicle control system). It is in such systems that greatly underutilized resources are not likely to be available due to weight, power, and cooling constraints; thus it is vitally important to efficiently control resource utilization in the event of overload.

Thus, the issue of meeting response time requirements, implying that time constraints exist and must be met to satisfy system specifications, must be separated from the issue of throughput. The nature of the required response time characteristics (e.g., must every deadline always be met, or can some be relaxed under certain conditions?) can be taken into account by explicitly exploiting the user's critical times and user policy directives in the real-time scheduling algorithms. The response time requirements and their characteristics in a real-time system are generally derived from the external physical environment with which the system must interact (e.g., a hypothetical aircraft's position must be computed every 100 milliseconds to ensure a positional accuracy of 25 meters).

1.1.4. The Archons Project

This research has been undertaken as a part of the Archons project 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 operating system which will be used to design and construct a "decentralized computer" [Jensen 81, Jensen 82, Jensen 84]. The operating system for such a decentralized computer is required to make many resource allocation decisions using inaccurate and incomplete information, and the results of this research are being used to construct techniques for making these decisions, initially in the process scheduling area. The Archons hardware configuration is expected to include separate processing capability for the operating system and its application program at each node [Wendorf 83]. This is expected to greatly enhance the capability of the system to support more cycle-intensive approaches to the required decision algorithms, both for process scheduling and other resource management, by allowing them to execute concurrently with the application.

1.2. Problem Statement

The purpose of any operating system is to present a virtual computer interface to a set of application programs such that the user can effectively utilize the resources of the computer to solve a particular problem. This virtual computer has, as its goal, a presentation of the resources of the underlying real computer in a form such that they can be efficiently and correctly used by an application program.

Resource allocation decisions are made on the basis of the available information within constraints; the information may be inaccurate and incomplete, the processing resources required to make an optimal decision using the information may not be reasonably constrained, or the decision-making algorithms may be intractable regardless of the processing resources. The decisions must still be made, however, so we must make a *best effort* using the available information to make the decision [Jensen 84].

1.2.1. Best Effort Decision Making

By *Best Effort Decision Making* we mean making decisions in an environment in which the best possible decision must be made regardless of the presence of inaccurate or incomplete information, or the intractability of the decision problem. In traditional data processing, the GIGO rule (garbage in, garbage out) has meant that it is considered wise not to make a decision at all rather than to produce an incorrect decision. To be sure, there are many situations in which this is valid, but an operating system must allocate resources for any progress to be made by the application, so its resource management decisions must be made in any case; it would not be appropriate for the operating system to delay the decision until "sufficient" data or an optimal decision can be determined.

A best effort decision may be contrasted with a statistical decision in which a decision is made based on imprecisely known, but stochastically valid information. A statistically optimum decision is quite possible if the optimality criterion can be defined, while for the scheduling model in which we are interested, an optimum decision is frequently not even definable, and is generally intractable if it can be defined.

We note that a great many human decisions are made in exactly this way; as much as possible of the available information is observed and the decision is made. There are a number of parallels between decision making by human beings and resource allocation decision making by the operating system. Some of these parallels are:

- The human decision maker will generally put off the decision only if he/she believes that the result of no decision will not be more adverse than a suboptimum decision. In this research, we explicitly deal only with decisions for which it is already clear that a decision must be made (e.g., processes must be scheduled or the system will stop. It would be better to randomly schedule processes than to stop the system.)

- The amount of processing to be applied to the problem will vary with the importance of the speed with which the decision must be made; for example, if a driver must either turn a car or brake to avoid an accident, the decision maker will make a best effort to make the decision very rapidly, even if pertinent information (such as the presence of a speeding truck behind the car) must be ignored.

- Even if a great deal of time is available in which to make the decision, the decision analysis will continue only until the decision maker believes that the decision is satisfactory, regardless of whether all data has been considered. The extent of this processing will be determined by the perceived importance of making a "good" decision (e.g., the choice of a restaurant for dinner will receive much less decision processing than a large investment decision, even though the amount and quality of the available information is probably quite similar.)

In the same sense, we must make a best effort to make resource management decisions. In this research, we ensure that a decision is made in any case, using the available information in the best way we can within the constraints of the quality of the available information, the time available for making the decision, and the importance of the decision.

1.2.2. Real-Time, Time-Driven Scheduling

A study of best effort decision making would be of limited value in a vacuum, however, so we have studied a particular problem, and have made several best effort decision making approaches to solve it, resulting in an algorithm which embodies a set of heuristics resulting in making "good" decisions. The particular problem we have studied is that of time-driven[1] scheduling in a real-time operating system.

An operating system managing a real-time application has many of the same functions as a normal time sharing operating system, but differs most significantly in one principal area -- the management of time, both as a resource and as a metric with which resource allocation can be evaluated. The typical time sharing operating system manages a number of independent applications, usually promoting some form of fairness (defined by the system administrator) among them. In the real-time system, a single application program[2] uses the system resources to solve a single problem. The correctness of such a problem includes assertions with respect to time; thus, completion of each process in such an application has a value to the system which is a function of time. It can be said that the value of all processing varies as a function of time, but the primary distinguishing characteristic of the real-time program is that the impact of time on its value is particularly severe.

In the current work on real-time scheduling, this value is generally simplified for computational tractability by approximating it with a step function with one value prior to a deadline and another following the deadline. Problems involving scheduling processes in the presence of such deadlines (i.e., due dates) have been studied for many years, giving rise to

---

[1] We refer to this problem as *time-driven* scheduling rather than *deadline scheduling* to distinguish our algorithms from the simple deadline scheduling algorithms [Liu 73] in which the process with the earliest deadline is scheduled first. See Chapter 2 for a further discussion of this subject.

[2] A single application program consists of a set of processes working together and sharing resources toward a common goal. Such a system is characterized by the fact that resource contentions can be resolved by appealing to the common goal. While it is possible for more than one such application to run concurrently in a real-time system, it is usually true that the real-time commitment of the system will be made only to one; any others will execute as background applications.

algorithms such as deadline scheduling, slack time scheduling, and a number of variations of these. However, the value of real process completion is rarely well modeled by a step function [Jensen 85, Locke 85]. In an actual system, the value of completing a computation after its deadline may rapidly decay but remain positive (e.g., being late on an aircraft navigation update might result in a loss of positional accuracy, while missing it altogether would exacerbate the loss, making it preferable to process it late unless it became so late that it impacted making the next update), or completing a computation greatly before its deadline may be more or less desirable than completing it exactly at its deadline.

Real-time systems have been divided into two classes: *hard* real-time systems and *soft* real-time systems (see, for example, [Mok 78]). 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, resulting in the loss of life and/or property), 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. The use of continuous value functions described in this thesis allows us to handle both hard and soft real-time environments if unbounded value functions are used, and, in particular, allows us to mix time constraints of both types in a single system, identifying which are soft and which are hard. The value functions used in this research include both step functions and continuous[3] functions, so we will avoid the use of the word "deadline", which implies only the step value function. Hence, we refer to the time of the discontinuity in a value function, if any, as its *critical time*. The use of such arbitrary value functions for process scheduling potentially makes the scheduling function much better equipped to solve realistic problems, but renders the scheduling problem even more intractable. Thus the use of *best effort* decision making for these decisions becomes even more important.

With the simplification of using only step value functions, a considerable amount of research has been done on scheduling situations in which it is postulated that all critical times can be met, but relatively little information is available about scheduling decisions when available resource limitations require that one or more processes must be delayed while the system must continue with its most important functions. Our approach, then, is designed to make scheduling decisions which will maximize the overall value of the resulting computation in either overload or non-overload conditions, but with particular emphasis on overload conditions.

---

[3] In this research, almost all of the value functions to be considered will contain at most one discontinuity which occurs at the deadline, and may be either in the function itself or in its first or second derivative.

Questions of policy/mechanism separation [Wulf 81] are also related to this problem. If one or more processes must be delayed, which ones? Clearly the decisions made must depend on user policy, and a range of interesting potential policies for this decision (see Chapter 4 for examples) has been studied, with their required mechanisms defined, implemented, and evaluated. We have paid considerable attention to the extent to which these policies (and possibly others) can be carried out by the collective action of a small set of lower level system scheduling mechanisms. It is this set of mechanisms which reflects the best effort approach to process scheduling, and their support of the potential policies has been analyzed.

Particular attention has been paid to policies affecting the overload condition. A major goal of this research is to allow the user to specify the policies controlling the behavior of the system in the presence of an overload, thereby avoiding the unpredictable failure modes otherwise occurring in a real-time system.

In this research, we define and evaluate techniques with which to schedule processes with continuous value functions, but a critical issue for the system designer is to determine the type of value functions which should be used for an application. Although we present some ideas on this topic in Chapter 4, we have not conducted extensive research on this question, which is therefore left as a future extension to this research.

1.3. Motivation

There are, of course, many existing real-time systems being used in a number of environments including space or airborne platform management, factory process control, and robotics. It is interesting, however, to observe some of the characteristics found in the operating systems[4] designed for these systems to provide support for real-time operation. Two primary characteristics 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, usually providing a low-level interface to starting I/O devices and handling their interrupts. Scheduling is almost always provided by some combination of FIFO (for message handling), fixed priority ordering for processes requiring real-time response characteristics, or round-robin for background processes, with the choice made arbitrarily by the operating system designers and/or the application designers.

[4] They are usually called *executives* when a full operating system is not implemented.

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

Conspicuously missing from these systems at the operating system level is any support for 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 actual worst-case user requirements, and the implementation is followed by an extensive testing period to verify that this assumption is maintained under "normal" and "worst case" loads. In priority-driven systems, deadline management is attempted by assigning a high fixed priority to processes with critical deadlines, disregarding the resulting impact to processes with less critical deadlines. This approach can work only for relatively simple systems, since the fixed priorities do not reflect the time-varying value of the computations with respect to the problem being solved, nor do they reflect the interaction of the priorities with the computations themselves. In addition, implementors of such systems find that it is extremely difficult to determine reasonable priorities since, typically, each programmer feels that his or her program is of high importance to the system. This problem is usually "solved" by deferring final priority determination to the system test phase, so the resulting performance problems remain hidden until it is too late to consider an effective solution.

1.4. Thesis Organization

The remainder of this thesis describes our approach to the solution of the real-time scheduling problem in the presence of continuous value functions, as well as the evidence of the success of this approach. In Chapter 2, we describe the related research from which our ideas and approaches are derived. The computational model used to define the scheduling decision environment within which we make our scheduling decisions is described in Chapter 3. Chapter 4 considers the application of continuous value functions to the actual real-time systems implementation process, including a discussion on expressing scheduling policy using value functions.

Chapter 5 describes a process scheduler which is capable of using the continuous value functions for a set of processes and efficiently producing a schedule which will consistently achieve a high total system value, even in an overload condition. To evaluate and demonstrate the effectiveness of this algorithm, we constructed a real-time, multiprocessor

simulator which is described in Chapter 6. The evaluation itself, in which a large number of experiments were conducted to tune the algorithm, exercise it under a variety of loads, and demonstrate its robustness in many conditions of inaccurate knowledge, are described in Chapter 7. The research is summarized with discussions of contributions and extensions in Chapter 8.

Chapter 2
Research Environment

Although there has been little work focused on this problem, there has been a great deal of effort in related research. This work has been most apparent in the areas of

- *Scheduling* -- Originally considered as a part of Operations Research, there is a large body of knowledge on the general problem of scheduling activities in many environments, including specification of optimality measures, complexity evaluations, and a number of optimal scheduling policies (i.e., scheduling algorithms).

- *Continuous Value Function Scheduling* -- The concept of using a continuous value function to describe the scheduling problem has been studied for a few classes of value functions.

- *Real-Time Scheduling* -- There has been somewhat less emphasis on process scheduling in support of real-time systems, but there are several important efforts which are closely related.

- *Decision Making* -- This work encompasses a number of contexts including Bayesian theory, and use of various approaches to logic (e.g., first order logic, non-monotonic logic).

- *Policy vs. Mechanism* -- In recent years, the concept of separating the specification of a system policy from the mechanisms supporting it has become increasingly important, and is somewhat relevant to this research.

In this section, we describe these related efforts and their relationship to our research.

2.1. Scheduling

Research into the general scheduling problem has progressed for a long time primarily as a part of Operations Research (OR), where it has found application in the scheduling of such activities as manufacturing production. Process (in the OR paradigm, processes are called "jobs") scheduling is separately considered in several classes;

1. Single Processor, Single Stage

2. Multiple Processor, Single Stage

3. Multiple Stage

Of these classes, the second relates directly to our work. The first class has been found to be quite tractable for many criteria; examples are the Shortest Processing Time (SPT) algorithm which maximizes flow, and the Earliest Due Date (i.e., Deadline) algorithm which minimizes the mean lateness. These results are available in many standard texts, such as Baker [Baker 74]. Even in this simple class, there are many scheduling criteria which render the problem intractable. For example, even such simple criteria as minimizing weighted tardiness has been proved to be NP-complete [Karp 72].

The second class, particularly for problems involving deadlines, has proven more difficult. Although a number of optimal algorithms have been found (see, for example, [Nunnikhoven 77]), they are all intractable, and the problem has been found to be NP-hard. Problems with multiple stages, also generally found to be NP-hard, are beyond the scope of this research. Graves [Graves 81] has provided an excellent synopsis of this background, including a taxonomy of production scheduling problems.

There are basically two approaches to the conceptualization of schedulers which have been identified by researchers in computer process scheduling:

- Priority Schedulers -- Conceptually, priority schedulers make scheduling decisions by invoking a priority function to determine the highest value process, then assign processor control to that process.

- List Schedulers -- List processors evaluate the set of ready processes and produce an ordered list of all such processes in the "best" order to optimize some performance measure.

The priority function described can have any of a large variety of forms, and can be driven by any suitable information (e.g., elapsed time, processor loading, working set size) available to the scheduler.

The use of a policy function shares some similarity with the technique used in our research, but differs from earlier work in the nature of the performance measure which we are attempting to maximize. Earlier work with policy functions envisioned a policy function of the available information whose shape is the same for all processes, and which depended only on the resource utilization and not on the performance requirements of the individual process for its evaluation [Bernstein 71, Ruschitzka 78, Ruschitzka 82]. Such schedulers simply

evaluated the priority of all available processes and selected the highest priority process for execution. Representative examples of priority functions in this context are constant functions for which the scheduler always selects the highest fixed priority ready process, linearly increasing functions with respect to elapsed time, resulting in FIFO scheduling, or linearly decreasing functions with respect to elapsed time, resulting in LIFO scheduling. The choice of times to make scheduling decisions determined whether the scheduler is a preemptive or time-slicing (time-slicing reflects a fairness policy which is primarily important in time-sharing systems) scheduler.

It has been shown that either of these types of schedulers, and, indeed, all schedulers, can be described as a form of priority scheduler, since the list produced by the list scheduler could have been produced by iteratively invoking a priority scheduler [Ruschitzka 77]. The scheduler produced by our research, however, is best classified as a list scheduler.

In the case of the multiprocessor, almost all optimal scheduling problems have been shown to be intractable. Using value functions (described as "payoff functions") similar to those used in our research but limited to exponential decay functions and non-preemptive schedules, Kain and Raie [Kain 84] compute bounds on the total value obtainable with list schedulers in which an ordering on available processes is constructed and the execution proceeds through this ordering. Their work is dissimilar to ours in their extremely limited value functions and non-preemptibility, which are imposed to allow their analytical results, and in their emphasis on time-sharing as opposed to real-time (i.e., time constraint driven) scheduling.

Sahni [Sahni 76] summarizes work in which it is shown that in a multiprocessor, even such straightforward performance measures as minimizing finish time or minimizing weighted mean flow time are NP-complete, and he further describes a heuristic linear-time algorithm which produces a "good" schedule minimizing finish time [Sahni 84]. Pinedo [Pinedo 83] similarly summarizes a number of efforts showing that even scheduling using simple linear value functions on lateness or tardiness (i.e., non-negative lateness) on a single processor is NP-hard.

2.2. Continuous Value Function Real-Time Scheduling

E. Douglas Jensen has frequently stated that the primary distinction between real-time systems and other computer systems is that in a real-time system, there is a value to the system for completing a process which is dependent on the time at which it completes. As part of an effort related to work for the United States Army in the mid-1970's, E. Douglas Jensen [Jensen 75], considered the idea of using such a value function as a guide to the process scheduler to schedule application processes to achieve a high value. This initial idea, which has remained unpublished, was the basis of some preliminary experimental work at that time, but no rigorous attempt to utilize this idea was undertaken prior to this thesis research, which has been completed under Professor Jensen's guidance. As has already been noted, however, this is not the first use of continuous value functions in scheduling processes in computer systems, but does represent the first explicit application of time-varying value functions to real-time process scheduling.

2.3. Real-Time Scheduling

In general, the problem of process scheduling in a real-time system, while it is touched upon by the scheduling literature in the form of "due-date" problems, has not been as widely studied as the general scheduling problem. Normal scheduling problems with deadlines which characterize real-time systems, are almost always (with a notable exception discussed below for single processors) shown to be NP-hard in either single processors or multiprocessors.

Real-time systems are characterized as either hard or soft real-time, depending on the seriousness attached by their implementers to missing a deadline. The hard real-time system is slightly more amenable to analysis, in that no positive lateness (i.e., tardiness) need be considered, and there have been several efforts dealing with this special case. For example, Liu and Layland [Liu 73] show that the simple hard-real-time deadline scheduling problem with a set of independent periodic processes whose computation times are exactly known and are identical for every period, whose deadlines are equal to their period, and which are running in a single processor, is solvable with the simple $O(n)$ deadline scheduling algorithm, and that the schedulability of such a simple system is easily determinable in advance. While this result is interesting, the cases covered are much too unrealistically simplified to be directly applicable to our problem, although we make use of this result as a part of one of our heuristics. Although this optimal solution exists when all deadlines can be met, it can be

easily demonstrated that this algorithm, (i.e., scheduling first the process with the earliest deadline), will provide very poor performance if some deadlines must be missed.

In a single processor, it can be shown that the minimum slack time (frequently called "minimum laxity" by real-time researchers) scheduling algorithm is also optimal in the same sense as the deadline scheduling algorithm [Mok 78]; given a set of processes whose deadlines can be met, the slack time algorithm will meet them. While neither the slack time nor the deadline algorithms are optimal in the multiprocessor, the minimum slack time algorithm greatly increases the incidence of preemption when multiprocessors are used or processes are considered to be preemptible as they are in our research.

In multiprocessor scheduling, there has been some important work in the problem of determining the processing node at which a real-time process can be "guaranteed" to meet its deadlines [Stankovic 84]. While the problem of process placement in a distributed, real-time system is beyond the scope of our research, the guarantee processing essentially evaluates the schedulability of a set of processes by heuristically determining the most likely successful schedules out of the set of all possible schedules and guaranteeing the time constraints of the process set of a successful schedule is found. This work concentrates on the context of systems containing processes with highly predictable processing times and containing a generally predictable overall load, since in the event of an overload there will be processes with undetermined importance which cannot be guaranteed.

2.4. Decision Making

Statistical decision theory describes the process of decision making in an environment in which an agent, having exhaustively defined:

- The set of possible states (which cannot be directly observed) which a system can assume and their probability distributions for a specific decision context (usually called "states of nature").
- A set of possible observations which can be made on the states of nature, and their conditional probability distributions relative to those states.
- A set of possible actions to be taken.
- A utility function which provides a measure of the desireability of each action given one of the states of nature.

applies a rule (e.g., Bayes' Rule) to this information to choose among the possible actions

maximizing the expected utility. Bayesian decision theory is described in many works, such as [DeGroot 70].

In a paper describing a technique for process placement, Stankovic [Stankovic 85] describes a heuristic for job assignments in a decentralized, distributed environment using Bayesian decision theory [Jeffrey 84]. In this work, each decision maker (i.e., node in a distributed computer network) estimates the state of all other nodes and makes process placement decisions based on this estimation, continuously collecting information with which to refine the state estimates.

Thus, decision theory is applicable to an environment in which a decision must be made whose value is determinable only when evaluated in the light of a (usually stochastically) predictable future state. The decision maker makes the decision by applying his assumed (probably stochastic) knowledge of the current system state (and the relevant distributions, if the knowledge is stochastic) and the utility or value of each potential final system state to decide on an action from a set of potential actions which will lead to the most desirable state.

Although it seems possible that decision theory could have applicability to the process scheduling problem, there are several difficulties applying it:

- Determining the utility values requires understanding the relationship between the potential states and the desired result to be obtained. This is the purpose of the value functions which we will define for each process; the value function expresses the desirability or utility to the system for completing a process at any particular time.

- If the probabilities of each potential state occurring are dependent on the decision to be made, as is certainly true for scheduling decisions, the probability matrix may also be difficult to determine, particularly when the relationship between the action and its effect cannot be effectively computed. In the scheduling environment, it is not only difficult to determine the effect of a decision, it is also difficult to accurately determine the current system state with respect to process schedulability.

In making our scheduling decisions, we begin with an estimate of current system state with respect to overload, a set of value functions which express the desirability of proposed actions (e.g., scheduling a particular process), and heuristics which can be combined to make an appropriate decision to produce a high expected value for the system.

2.5. Policy vs. Mechanism

The use of continuous value functions to control real-time process scheduling represents an attempt to provide a mechanism for scheduling control which is separate from the definition and imposition of user-defined scheduling policy. The process scheduler in Hydra [Levin 75], for example, was implemented as a parameterized policy handler for which the parameters provided for the control of a priority schedule, with the priorities fixed for a given period of time and for a given class of processes in the system. Since Hydra was intended to be a time sharing system, it was assumed that the classes represented independent users, and the underlying mechanism attempted to enforce a fixed, predefined "fairness policy" between classes. In our work, we assume the existence of a single user for the entire system, so we need no such underlying fairness policy. Our scheduler is also a parameterized one, but the parameters provide for the complete specification of a value function, which allows for a high degree of control over the actual scheduling decisions, and furthermore provides control over the actual desired effects of scheduling (i.e., total value over a given time interval) rather than merely providing explicit control over the scheduling mechanism itself.

Chapter 3
The Scheduling Computational Model

Before we can discuss techniques for scheduling a real-time system, it is important to define the context in which our scheduling decisions will take place. This chapter describes the model of a process, its time-value function which determines the nature of the scheduling decisions we will make, and the parameters which will be used in the decision computations.

3.1. The Process Model

This research assumes an environment containing a set of processes which are requested at arbitrary times, with a scheduling decision to be made whenever a process is requested or terminates. Precedence constraints are handled by assuming that a process will be requested only when its precedence constraints have been satisfied, and periodic processes are unique only in that their request times occur at regular (predictable) intervals. As is usually true in real-time systems, it is assumed that all processes and their local state information are permanently resident in memory.

At the time a scheduling decision must be made, the model for this problem consists of a set of processes $p_1, p_2, \ldots, p_n$ resident in a shared-memory multiprocessor. Each process p_i has a request time R_i, an expected computation time C_i, a critical time D_i, and a value function $V_i(t)$, where t is the time at which the value is to be determined. Figure 3-1 illustrates these process attributes for a process with a linearly decreasing value function prior to its critical time D_i, and an exponential value decay following the critical time. The illustration depicts a process p_i which is dispatched after its request time and which completes prior to its critical time without being preempted.

$V_i(t)$ defines the value to the system due to completing p_i at time t. The parameters and value functions for all schedulable processes are used by the scheduler to determine an appropriate sequence in which to schedule each process. The type of functions definable for

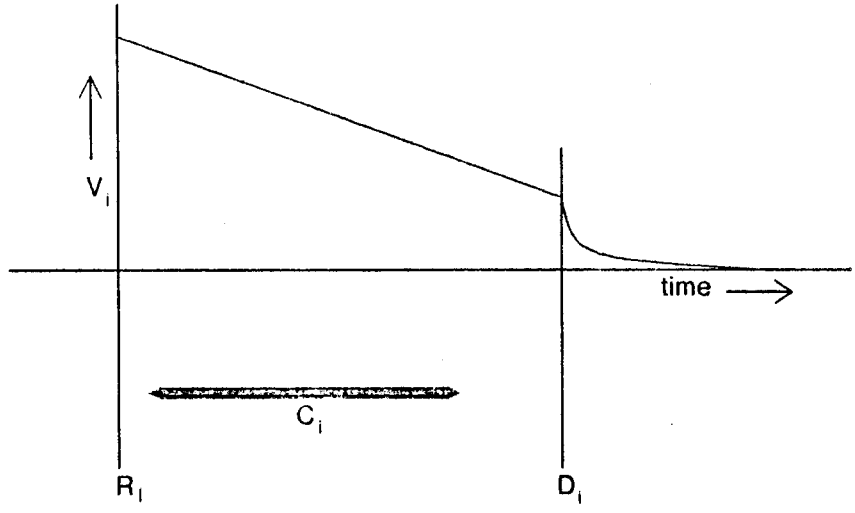

Figure 3-1: Process Model Attributes for Process *i*

$V_i(t)$ will determine the range of scheduling policies supportable by the operating system (see Chapter 4), particularly with respect to the handling of a processor overload in which some critical times cannot be met.

We assume that all processes are fully preemptible and restartable once they have been requested. Preemptibility means that a process can be suspended at any time during its execution and another process may be executed, normally followed later by the restart of the original process from the point at which it suspended processing. In our evaluation, we account for the costs of preemption explicitly by adding an overhead time for each preemption-resume.

We note that the existence and importance of a process' deadline is dependent on its value function. The value function can be said to define an explicit deadline only if it has a discontinuity in either the function or its first or second derivative, and a value which is lower or decreasing after the discontinuity. For example, in Figure 3-1, the deadline is defined by the discontinuity in its first or second derivative at the critical time D_i. Of course, if a value function has no such discontinuity (e.g., a constant function), the choice of a critical time is arbitrary.

3.2. Process Model Implications on Scheduling

The request time R_i could also have been defined by the value function rather than as an arbitrary time by allowing the value function itself to define the earliest time at which completing p_i would create a positive value. In this work, however, we have defined R_i to be an arbitrary time, because we expect that in most cases it will be determined arbitrarily by the application as a direct result of its use of the time management operations provided by the operating system. Value functions can, of course, effectively delay a process by defining a small (perhaps negative) or rising value prior to the earliest time at which the related process should be allowed to complete. Allowing the application to define R_i arbitrarily also provides the resulting operating system with a simple definition of the schedulability interval (i.e., the interval of time during which a process is a candidate for receiving processing resources): a process becomes schedulable when R_i has been reached. It remains schedulable until one of the following conditions has occurred:

- It has completed.
- It cannot be executed such that at completion its value function remains positive.

The computation time C_i is a random variable representing the execution time required to process p_i (estimated time remaining if p_i has already begun processing), not including system overhead or preemptions. The determination of this time is expected to be either the programmer or an actual measurement by the system itself. The method used to estimate the processing time is one of the questions to be considered as part of the policy/mechanism discussion (see Chapter 4). In this research, the distribution of C_i is an important issue to be investigated, and our simulation of our best effort scheduling algorithm using this model includes a wide variety of distributions of C_i. The distribution is considered to be known to the scheduler, although we do study the impact of misspecification of the distribution in Section 7.

At any particular point in time, there will be n processes ready for scheduling, resulting in $n!$ possible scheduling sequences. Each of these sequences consists of a process ordering $(m_1, \cdots, m_n)$, where p_{m_j} would be the j^{th} process to be scheduled. A scheduling sequence will be considered optimal if, with respect to the available information at the time of the scheduling decision, β is maximized, where $\beta = E(\sum V_i(T_i))$ and T_i is the actual completion time of p_i using this scheduling sequence (i.e., if p_i is the j^{th} process to be scheduled, then $T_i = \sum_{k=1}^{j} C_{m_k}$). See Figure 3-2 for an example of four processes with four separate value

functions, for which the choice of a best effort schedule is non-trivial. The figure shows the four value functions, and a potential scheduling sequence such that each completes with a high value. In addition, the scheduling algorithm will be expected to determine future key times at which the scheduling decision should be reconsidered. for example when a process has executed longer than expected and completion of a high value process is jeopardized. At such times, as well as at the request of new processes, the scheduling decision will be repeated, allowing preemption and abortion of processes depending on the changing workload.

While we have thus defined an optimal sequence for a set of processes, this definition assumes a static set of processes to be scheduled, not taking into account the possibility of additional aperiodic processes arriving prior to the completion of the schedule, so it does not result in a useful definition of optimality.

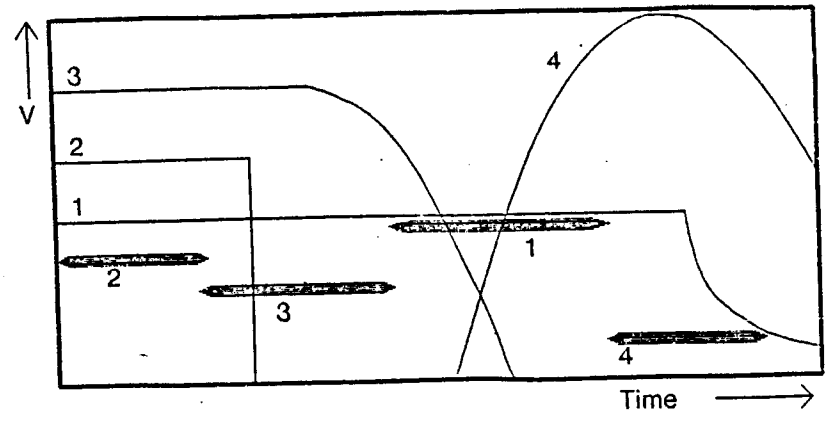

Figure 3-2: Four "Typical" Processes with their Value Functions

Note that no priority has been defined for p_i. In existing real-time systems, each process typically has a (usually fixed) priority related in some way to its perceived importance. The use of the value function renders such a concept unnecessary for this research, since the decision of which process to execute, at any point in time, will be determined by their respective value functions at their estimated completion time rather than on some concept of importance. The concept of importance is defined by the policy under which the value functions will be interpreted. A separate question is how an application designer will define the relative importance of a number of competing processes (policy). A number of potential solutions to this problem can be defined; the choice will depend on the behavior desired and the actual type of value functions being used (see Chapter 4).

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 an aperiodic process. We can beg the question of overlapping executions since, with our model, they can either be avoided (by ensuring that a periodic process' value function drops below zero prior to its next request time), or handled by allowing multiple instances of a process to be simultaneously schedulable.

It should be noted that precedence and consistency (e.g., processes involved in mutual exclusion) relations are addressed through the definition of the request time, but are not otherwise explicitly addressed in this model at this time; thus, if one process is dependent upon completion of another, its request time will not yet have been determined, and it will not be eligible for scheduling. The extension of this work to consider precedence relations in the scheduling process will be a future activity. In addition, future work will add consideration of the effects of process synchronization (i.e., in which one process must wait for another process to exit a critical section) on the scheduling decision.

Chapter 4
Value Functions in Practical Systems

The determination of the value functions is one of the first concerns to strike a system designer unfamiliar with the use of time value functions as the decision criteria for a process scheduling system. Given a scheduler which can satisfactorily make scheduling decisions using these value functions, the formulation of the value functions for a particular system is obviously critical to the performance of the system. In this chapter, we will consider this question in some detail, providing not only the rationale for the use of these functions to control process scheduling in an operating system, but also some concrete guidelines to the selection of value functions for practical real-time systems.

Among the questions to be discussed in this chapter are:

- At what point in the development process is the understanding of the complete system requirements sufficient to develop the value functions for the entire system?
- Who should define the value functions?
- Are the functions constant in time, i.e., should the value functions be dynamic or static in real time during execution of the system?
- If the value functions are dynamic, under what conditions should they be changed, and how often?
- What is the effect on system performance of the different types of value functions?

The answers to all of these questions require an understanding of the scheduling policy to be implemented by the operating system scheduler. For our purposes, we define policy as *a set of rules which guide and, in general, completely determine the scheduling decisions that are made as each process becomes ready to be executed.* Thus, the term "policy" can refer to a number of different types of rules, from very low level ones, such as "*Execute the shortest process first* which we might encode as a scheduling algorithm, to rather high level ones,

such as "*Processes, once started, should never be preempted*", which potentially affect the entire structure of the operating system.

Following this introductory section, in section 4.1 we introduce a taxonomy of scheduling policies and in section 4.2, will discuss the justification for the use of value functions as a scheduling control mechanism, discussing some of the limitations of the use of value functions for controlling scheduling policy. Finally, in section 4.3 we consider the sources of, and controls on, the actual value functions to be used by the operating system scheduler, in addition to some of the ways in which the system designers might define value functions and some of the principal effects of certain types of value functions on the resulting system performance.

4.1. A Scheduling Policy Taxonomy

We postulate with little fear of contradiction that the number of possible scheduling policies is unbounded, and we will deal with only a relatively small subset of them, but this subset will be based on experience with the design and implementation of real-time systems. In addition, it is likely true that the number of possible characterizations or taxonomies of these policies is limited only by the imagination of those constructing them. Nevertheless, in this section, we will attempt to describe such a taxonomy by considering policy examples in several classes, making the unproved (and almost certainly unprovable) claim that most of the "useful" policies for real-time scheduling will have been included.

1. **Inter-Process Relationship Policies.** An important class of policies involve the direct relationships between processes. Such inter-process relations must guide the scheduler when tradeoffs must be made between conflicting demands for processing resources (i.e., when a resource is overloaded). For example, a policy in this class might be of the form:

    - Process X is more important than process Y.

2. **External Environment Time Constraint Policies.** Time constraints are the principal characteristic of the real-time system, and the expression of policies governing time constraints form an important part of the overall definition of scheduling policy. Policies of this type are one of the principal focuses of this research and the scheduling algorithms being defined. An example of this class of policy is:

    - Process X must complete within Y seconds after event Z.

3. **Scheduling Performance Criteria Policies.** Another important class of policies reflect the choice of the scheduling performance criteria to be optimized

by the process scheduler. This class includes some criteria for which optimum scheduling algorithms are known, and a great number for which no optimal algorithms are known. Policies in this class include:

- Minimize the maximum lateness among all processes.
- Minimize the average tardiness among all processes.

4. **Long-Term Time and Event Related Policies.** Although policies related to time constraints are the principal focus of real-time process scheduling, there is an extensive set of longer term policies involving the relationship between process scheduling and time. Examples of such policies are:
   - Process X becomes twice as important whenever one of its deadlines is missed.
   - If Process X has not completed prior to its critical time, it should be aborted.
   - If the system is in Melt-Down Mode, the cooling system control process becomes 100 times more important than it was in Start-Up Mode.
   - If the probability that one of more deadlines will be missed exceeds 25%, a set of candidate processes should be identified for load reduction.
5. **General Control Policies.** There is a set of policies which govern the philosophy under which the process scheduler makes its decisions. These policies may be parameterized to allow some tailoring of the scheduler, but many of them are quite static, and represent the intent of the system architect to determine the portion of the design space in which the system should be constrained to run. Examples of such policies include:
   - Processes, once started, should never be preempted.
   - In its search for the next process to be executed, the scheduling algorithm must never examine more than 30 processes.

In the list of scheduling policy classes described above, it appears that classes 1, 2, and 3 can be described in terms of the value function associated with each process during scheduling. In class 3, there may be some criteria for which the nature of the value functions is difficult, but there are clearly a large number of such criteria for which a careful choice of value function would lead a value-driven scheduler to achieve a good (if not optimum) schedule. For example, if a set of processes all had an identical set of linearly decreasing value functions, a value-function scheduler should attempt to minimize the average lateness.

Class 4, while perhaps not easily expressible in terms of the value functions as described in this research, could readily be implemented by a dynamic policy module with the capability to

modify or replace the current set of value functions as a function of time or system state. The policies in class 5 would not be amenable to characterization in the value functions, but would require parameterization within the scheduling process itself, and could even affect the entire structure of the operating system. As a result, the members of class 5 to be supported by a particular implementation of a scheduler are normally determined directly by the process scheduler designer by enumerating them and handling them directly in the operating system specification.

4.2. Rationale for Using Value Functions to Define Policy

Scheduling policies, like the programs which implement them, can be described at several levels, with significant ramifications on the architecture and design of the mechanisms by which they will be carried out. In existing real-time systems, scheduling policies are usually fixed at system design time, and are almost always implemented using fixed priorities assigned as a function of the application architect's concept of process importance, resulting in anomalous behavior in the event of higher than predicted load levels.

We note that the expression of policy can take a number of forms, which we will now characterize as high or low level policy definitions. These levels correspond directly with levels of programming languages; the lower levels provide for extremely detailed specification of policy, with no bounds on the types of policies expressible, and with the results of the policy difficult to ascertain from the statement of policy, while the higher levels of policy expression relate directly to observable performance, with little insight provided as to the mechanisms or implementation underlying it. Policy for existing real-time systems is usually expressed at lower levels, but we will make the claim that mechanisms supporting higher levels of policy definition can provide significantly more predictable performance, particularly with respect to issues relating to reliability and fault tolerance in the presence of transient overloads.

4.2.1. Low Level Policies

Consider a set A of low level policies including such examples as:

1. At all times, the ready process with the shortest expected completion time shall be executed.

2. At all times, the ready process with nearest deadline shall be executed.

3. At all times, the ready process with smallest slack time shall be executed.

Such statements can legitimately be called policies in that they specify the user's wishes regarding the selection of a process for execution. These policies, however, can be characterized as extremely low level, in the same sense that assembler language is considered to be low level compared to, for example, Lisp. These policies actually are algorithms which can be used to construct the scheduler; they do not specify what effect (i.e., what performance criterion to be optimized) is desired to support the application, but rather they directly specify how the selection is to be made.

4.2.2. Medium Level Policies

Consider a second set B of policies, from which a few examples might be:

1. At all times, the process shall be executed that will minimize the mean flow time.

2. At all times, the process shall be executed that will minimize the maximum lateness of all requested processes.

3. At all times, the process shall be executed that will maximize the minimum lateness of all requested processes.

These also are legitimate policies in that they specify the user's wishes regarding the selection of processes for execution, but they are higher level than policies from set A in that policies from A consist of algorithms which might be used to implement the policies given in B[5]. Note that in the same sense in which assembler language is "more powerful" than Lisp, policies in set A are "more powerful", in that a broader range of policies are possible: such policies as "Schedule first the ready process whose programmer's name is first listed alphabetically" could be specified in set A, but cannot be described in set B. The difference is that the second set contains objective scheduling performance measures resulting from the application of some algorithm, but not the algorithm itself. A policy utilizing the programmer's name, for example, cannot be included in set B precisely because there is no objective performance measure which can be optimized using such a policy.

---

[5]Not coincidentally, the three algorithms used as examples of policies in A will optimally satisfy the criteria used as examples in set B.

4.2.3. Higher Level Policies

There is a still "higher level" policy set which could be used rather than sets A or B, which can be characterized as a single policy, with parameters with which it can be tailored so that it will specify virtually all the policies in set B. This policy set X can be simply stated:

1. At all times, the process shall be executed which, when it terminates, will result in the greatest total value to the system *over the long term*.

This policy requires the specification of the value to the system for terminating each process at any particular time. Specifying this value appropriately can result in satisfying the policies in B, and can also handle combinations of such policies applied to different classes of processes. For example, a class of processes (Y) could be specified with constant values, and another class (Z) could have constant values prior to a critical time, with zero value following the step. Such a combination might be appropriate in a system which was concurrently controlling a set of flow valves for a process control application, and doing a regression on data collected by flow sensors to predict long-term variations in flow rate to be used to modify the settings for use tomorrow. In this case, the constant-value processes might actually be increasing in value over time, but the change might be sufficiently slow to appear constant to a scheduler[6]. If the total value to such a system is to be maximized, the processes in class Y will automatically be processed whenever necessary to ensure that they terminate (barring an overload) prior to their value changes, since the value to be derived from executing processes in class Z can be achieved at any time.

By *long term*, we mean a length of time much greater than the process completion times of individual processes in the system (e.g., if the mean execution time of a set of processes is 1 second, a *long term* policy might extend over a minute or longer). This implies simply that the scheduler should not unnecessarily sacrifice the value obtainable from a future process merely to gain a high value in the immediate future from a process in its run queue.

---

[6] By this we mean that their values can be held constant over the time required to make a scheduling decision.

4.2.4. Qualitative Differences in Policy Levels

The question must then be addressed: "Is set X 'better' than sets A or B?" This judgement might be based on several factors:

- Is X more powerful (i.e., expressive) than A or B? Policy set X seems to be considerably more expressive than set B, but is it more expressive than set A?
- Is the set of policies expressible with X more "valuable" in some sense than A or B?
- Is the nature of the expression more "natural", leading to greater predictability of system performance?

The answers to all three of these questions can be determined by considering the arguments to the value functions which will determine the value at the termination time of each process. Certainly, these functions will have time as an argument, but they could also have other arguments. In fact, if the set of arguments allowable is unbounded, it seems quite intuitively clear that all the policies expressible in sets A and B can be expressed. For example, if the programmer's last name is allowed as an argument, the undesirable policy described above could easily be implemented by a value function scheduler.

If, as we believe, it is the case that the policies in set X include all policies in sets A and B, we can hardly claim that set X is "better" than A or B, unless X is (a) larger than $A \cup B$ and (b) includes useful policies in the set $X-(A \cup B)$. Given unbounded arguments, it is possible that set X is larger than $A \cup B$, but it is not clear that there are useful policies in $X-(A \cup B)$.

What, then, is the advantage of specifying policies using set X, and how can such policies be controlled to prevent the implementation of "bad" policies such as exemplified previously? To a great extent, we cannot control the specification of "bad" policies, except by limiting the arguments available for creating value functions (e.g., we need not allow programmers' names to be available to a value function). Even if we cannot prevent the specification of "bad" policies, the primary advantage of specifying policies through their value functions is that the concept of a time value for a real-time process is a natural one, reflecting an intrinsic property of a real-time system, and that, as shown by this effort, a coherent scheduler making a best-effort scheduling decisions is possible, which, given a set of value functions collectively and individually expressing the completion value for each process, will implement the user's policy with a great deal of accuracy. Such a scheduler can, through functional composition as will be discussed later, unambiguously combine apparently conflicting

policies from competing portions of the system. In addition, such a scheduler need not be modified during the lifetime of the system, as long as it supports a sufficiently powerful form of the value function for all processes.

4.3. Scheduling Policy Sources

Given that policies are defined as in set X previously described, who should specify them? There are apparently at least three potential sources for policy information: (1) the system architect, (2) the process implementer, and (3) the user. We might note that each of these policy sources typically shares an innate distrust of each of the others; hence there is good reason for each to desire to limit the range of policy choices for the others. The architect is primarily interested in the externally observable performance of the system, most frequently dictated by physical external constraints, including such behavior as the the maximum elapsed time from stimulus X to response Y (e.g., if a reaction temperature rises beyond a given level, the incoming flow must be cut off by a certain time to prevent a dangerous condition). The implementer is concerned with the performance of individual components of the system (e.g., when process M is executed, it must be completed by some given time with probability p in order to enable the remainder of the system to meet its externally defined time constraint). The user, particularly in a real-time system, may need to override the architect's and implementer's values with his/her own, depending on the system mode or other external consideration (e.g., war having been declared, or meltdown being imminent).

Using these multiple disparate sources for policy information, how can value functions be used to construct a coherent, consistent policy? It is clearly critical that some form of composition of the value functions defined by these sources be used in making each particular scheduling decision. To answer this question, let us consider how each of these sources would (or should) construct their value functions and how a value function scheduler would respond to the functions defined.

4.3.1. Value Functions Defined by System Implementer

Let us assume that the system implementer is producing a particular process, and we shall assume that this process is part of a larger function which has been specified by the system architect. The implementer is aware of the needs for input and the requirements for output for this process in terms of the input sources and the eventual destination of the output, and he is aware of whether the interface is between processes or directly with an external interface

such as a sensor or actuator. Since the implementer is aware of the function the process is performing, he will also have an idea of the value to the system for completing that process as a function of time. Of course, this value can be specified as a function of whatever parameters are available to the implementer, but here we will consider primarily the time parameter, i.e., the function of time that the implementer will be using to define the process value.

For a moment, let us separately discuss two aspects of a value function at any moment in time: (1) the level of the function, i.e., its actual value at a particular time, and (2) its slope, or rate of change with respect to time at a particular time. From the point of view of the system implementer, the level can be defined only relative to other process time values within a particular system function, but he is unlikely to be in a position to accurately estimate the relative importance of this function with respect to other functions in the system, and it may be very difficult for him to evaluate the level of this function versus the remainder of the system under different mode and other external conditions.

Thus, the implementer's decision with respect to its actual value of the function at any point in time is probably less important than his decision with respect to it's rate of change (slope). The slope of the value function is used by the value function scheduler as a critical part of its scheduling decision process. There are three cases to consider at any moment: an upward slope in which the value is increasing as time increases, a level slope (i.e., a slope of zero) in which the value is not significantly changing at that moment, and a negative slope in which the value is decreasing as time progresses.

Excluding, for a moment, consideration of the fact that the scheduler is comparing the value function of this process relative to those of other processes in the system, let us consider what its action will be with respect to the slope of this particular value function.

- If the slope is currently increasing, it is an indication to the scheduler that it may achieve a higher value by executing this process at a later time. In fact, the scheduler would attempt to schedule such a process at such a time that, given its expected completion time, it will complete near to the time of its maximum value. Thus, the use of a rising, or positive slope, by the implementer will have the effect of delaying the execution of the process, unless the scheduler believes that delaying the process is likely to result in not completing it at all.

- A negative slope, on the other hand, implies that the value of the process is decreasing, so the scheduler will then attempt to complete execution of that process as soon as possible (assuming that it believes that it can do so without

incurring a zero or negative value at its completion), attempting to achieve as high a value as possible from its execution. Thus, the implementer would wish to construct a negative slope for any process which must be executed as quickly as possible.

- Processes whose value functions are constant, or nearly constant, will tend to be executed by the scheduler when no processes with negative slopes are ready for execution. This will take place because the scheduler will believe that it can achieve the value from this process at any time, so there is no particular rush in executing it. Processes whose slopes are zero, then, will tend to behave as fixed priority processes, in that the scheduler will generally schedule first those constant-valued processes whose value is highest, but only after all negative slope processes.

The preceding discussion has been perhaps slightly oversimplified in that the scheduler will also, of course, be looking ahead in time, and, for example, given a value function which is constant but containing a future slope downward will attempt to complete it prior to the decrease in value, thereby achieving the highest possible total value.

The implementer must resist the temptation to misuse the value function. It might intuitively seem that if the implementer wishes to maximize the likelihood that a process will complete prior to a particular time, he might consider using a value function which is dramatically rising prior to that time, then dramatically falling at, or subsequent to, that time. Such a value function would not have the desired effect, because the positive slope preceding the critical time would cause the scheduler to delay the processing until the last moment, attempting to schedule the process to complete exactly at the critical time, whereas the actual desire of the implementer is to have it completed at some time prior to the critical time. Thus, in such a case, the implementer would probably want to use a constant value or a negative slope prior to a critical time, followed immediately by a rapid (perhaps instantaneous) drop after the critical time. His desire to increase the chances that that process will, in fact, be completed prior to the critical time, will be best achieved by insuring that it's value is as high as possible prior to the critical time relative to the other processes within its function.

In creating a value function for a process, only the implementer is aware of the collection of processes which comprise a specified function. Therefore, the relative values of the processes within a particular function will usually be under his control, and he should be careful about specifying the relative levels of each of the processes which comprise a particular function. In addition, it is frequently the case that a particular process is used for more than one specified function, in which case the implementer would specify that the value

function for that process would depend on the function on whose behalf the process is executing. For example, the process of displaying a value on the CRT is likely to be common between a navigation function and a control function, so the value function to be used in scheduling the display process should be controlled by the requesting function. In the event of additional requirements for such a collection of processes, the implementer will need to reconsider the relationships between processes in the collection.

4.3.2. Value Functions Defined by System Architect

The architect knows the value of each of the externally defined functions relative to each other and especially with respect to the physical environment of which the real-time system is a part, thus the architect will need to be able to control the specification of the value functions. This control could extend to a number of the value function parameters, such as the overall amplitude and its critical time(s), but it seems that the architect's control need not extend to the shape of the value functions of individual processes, nor need the architect specify the relative levels of process value functions within an externally visible (and specifiable) system function.

4.3.3. Value Functions Defined by System User

The user has a direct interest in the value function specification, but lacks insight into the internal structure of the system, so his value concern will reflect only observable responses at the user interface. The existence of certain events or modes not directly observable by the software might need to be controlled by the user, but the nature of the user's input to the value function definition will be directly under control of the architect. Thus, the architect may allow for some modification of value to be generated by the user via manual input or other user interaction. Such functions as system configuration management, process migration, fault tolerance, and related functions will probably dictate modification of value functions, under some level of user control, for specific externally definable system functions.

4.4. Summary of Practical Value Function Specifications

For those policies which can be expressed as time-value functions, we believe that the different sources of value can be composed in straightforward ways. It is not yet clear whether this composition need consist of more than a linear composition, but it seems clear that the composition need not result in value functions more complicated than would be needed if only a single source of value were to be used. If a simple linear composition were sufficient, it might look like: $V_c(t) = V_a(V_i(t)) = \alpha + \beta V_i$, where $V_i(t)$, $V_a(t)$, and $V_c(t)$ are the implementer's, architect's, and composite value functions, resp., and α and β are considered constants by the scheduler (i.e., they will be held constant during the decision-making process at the time of each decision). An extension to this work should include investigation and experimentation with the separation of value function concerns among the participants in a real-time system project, determining the effects of different levels of control by each of these project control levels.

Chapter 5
A Best Effort Scheduling Algorithm

Given that an application system designer has appropriately defined the set of processes which correctly perform the functions specified for that application, and further assuming that the appropriate value functions and scheduling policies have been defined for these processes, it becomes possible to discuss the nature of a scheduling algorithm which will be capable of assimilating this information and the state of the system and make scheduling decisions. In this chapter, we define the heuristics applicable to these decisions, develop a specific algorithm which utilizes these heuristics, and discuss some of its operational characteristics.

5.1. Scheduling Assumptions

There are a number of assumptions regarding the system state and our knowledge of the application processes, which we use in constructing a scheduling algorithm. In this section, we describe these assumptions and define how they relate to the heuristics and the resulting scheduling algorithm.

5.1.1. Operating System Processor Availability

It is assumed that this scheduler operates in an environment consisting of one or more processors attached to a (shared) memory. Further, we assume that the scheduler itself need not run in the same processor as the processes to be scheduled. The environment in which this scheduling algorithm is expected to be used initially is that of the Archons project at Carnegie Mellon University. In this project, a distributed computer consisting of nodes each of which is a multiprocessor, provides the environment in which this scheduler will be executed. A global operating system containing this scheduler as part of its resource management will reside in the distributed processor and within each node a separate processor will be allocated and eventually tailored for the purpose of managing the system resources for the Archons system.

The principal consequence of this assumption with respect to the scheduling problem is that we can use an algorithm which is somewhat more computationally intensive than we might consider otherwise. While it is true that we are concerned with building a real time system that must make its decisions within a bounded and, in fact, small period of time, the characteristic about which we are most concerned is the elapsed time required to make a decision once its supporting information is available. Because of the assumption that a processor is available which is separate from the application processor or processors in this system, we can design our simulator to pre-compute as many values as necessary concurrently with the execution of application processes.

This assumption is not crucial to the design of this algorithm although the algorithm is somewhat more computationally intensive than most scheduling algorithms. If a multi-processor were not configured with a dedicated operating system processor, the operating system could execute this algorithm in the processor completing an executing process or receiving a new process request. If the resulting process list indicated that another processor should change contexts, an inter-processor interrupt could be generated to cause that processor to perform the change. We expect that the computational complexity of our algorithm should still allow application time constraints to be met. For a detailed discussion of the computational complexity of this algorithm see Section 5.4.5 below.

5.1.2. Preemption of Executing Processes

We assume that all application processes can be preempted whenever required in order attain a high value to the system for completing each process at an appropriate time. We would note that this is a significant departure from most existing real time systems. In such systems it is typically the case that a high priority process is designed making the tacit assumption that it will never be preempted simply because it is assumed to be the highest priority process at all times, and the normally utilized fixed priority scheduler is designed never to preempt a process with a lower priority process during its execution. We might note that in such existing systems this rule is frequently violated during the handling of interrupts (e.g., device completion and other events), thus imposing stringent restrictions on the nature of interrupt processing in these systems and further reducing the predictability of their performance in the event of an overload.

This assumption results in the ability of the scheduler to continuously execute those processes which will result in a high cumulative value to the system; it will not be the case that

a low value process can use system resources to cause a high value process to violate its time constraints. If allowed by appropriate user policy decisions, the algorithm may, for example, execute a portion of a process with a future optimum completion time significantly prior to that time, interrupt it, then wait to complete that process as its time draws near, having run other processes preemptively in the meantime. This provides the scheduler with the capability of finely controlling the ending time of processes which must be controlled with great precision in order to achieve a high value. In addition this capability allows the scheduler to insure that an incoming aperiodic high priority or high value process is executed prior to the completion of a lower value process in the event that the processor becomes overloaded. For examples of this effect in an actual scheduling situation see, Chapter 7.

5.1.3. Available Process Information

Whenever a process is requested, we assume that certain information will be maintained about that process instance throughout its lifetime until it is terminated or aborted. This information will be maintained and updated as necessary throughout the life cycle of this process instance and will be used as the available process knowledge with which to make the scheduling decision.

The single most important piece of information required for a process instance at the time it is requested is its value function. The value function can be specified in any of several ways. For the purpose of the evaluation performed in this research, the value function is specified using a set of constants which define a polynominal describing its value at every point in time (see Chapter 6 for a description of this value function formulation). We refer to the parameters of this polynominal as constants although there is no reason to assume that they will not change throughout the lifetime of the application system being scheduled. We do, however, assume at this point that these parameters remain constant once an instantiation of a process has been requested, and will remain constant throughout the scheduling of this particular process instance. Thus, we assume that the value function has been completely defined at request time for a process instance. Future changes to the scheduling parameters used for this process for future requests will affect only the future process instances.

The primary result of this assumption is that the scheduler can make decisions with respect to the expected value it believes to be achievable by scheduling each process at a particular time, without considering the possibility that these parameters might change during the scheduling operation. This assumption is not essential to the construction of our algorithm

but it does mean that some scheduling computations need not be unnecessarily repeated. This assumption can be relaxed if the implementer of the scheduler is willing to recompute the schedule at more frequent intervals, based on the changes to the scheduling parameters for each process.

Additional information assumed to be available for each process at the time of its request includes the current expected time required to complete this process and the expected execution time distribution (as well as the parameters describing this distribution) with which the computation time may be estimated. The algorithm assumes that this distribution and its parameters have been made available, although we make no statement as to the source of this information; we would assume that the distribution itself has been supplied by the user and that the parameters defining that distribution have been supplied through either the user interface or through actual operating system measurements of previous executions of the process. If the execution time is being evaluated by the operating system, we would expect that the performance of this algorithm will improve over the extended period of time which is characteristic of a real-time system, due to the improvement in the quality of the information on which its decisions are based. This improvement is a valuable property of the decision making process, but it requires that the operating system continuously make appropriate resource utilization measurements throughout the system execution.

5.2. Observations About Value Functions

In this section, we note some important characteristics of the value function shape about which we make assumptions upon which our scheduling algorithms heuristics are based. Although our heuristics make few assumptions regarding the details of the value function shape, there are a few critical features of the value functions which are important to the scheduling process.

It is important that a value function not be monotonically increasing following its critical time; the scheduling algorithm assumes that following the critical time the value for completing this process either remains constant or generally decreases in some form, eventually reaching the value of zero or less. The value need not monotonically decrease, but it must drop eventually.

Following the critical time, it is not essential that the value of zero be achieved but not achieving it has the potentially negative effect in an overload that the process will never be

removed from the processing queue and thus will introduce scheduling decision overhead forever unless process completion eventually occurs. A value function which passes zero makes the associated process eligible for being aborted, thus keeping the queue cleared of processes whose value has become so small that they need not be competing with incoming higher valued processes.

5.3. A Best Effort Scheduling Algorithm

Having defined our assumptions, we construct our algorithm using a sequence of heuristics. This sequence has been designed to parallel closely the thought processes used by a human decision maker in determining how to sequence a set of processes.

5.3.1. Background

In defining the Best Effort scheduling algorithm, we build on two observed value function and scheduling characteristics:

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

2. Given a set of processes (ignoring deadlines) with precisely known constant values for completing them, it can be shown that a sequence of processes in decreasing order by value density (V/C, in which V is its value and C is its processing time) will produce a total value at every process completion time least as high as any other schedule.

The former observation is well known (see, for example, [Baker 74], page 24); the latter observation, although similar to a number of results in other contexts, is new in the context of real-time process scheduling, and we prove it here:

Theorem 1:

Given processes $P_1, P_2, \ldots, P_n$ with actual execution times $C_1, C_2, \ldots, C_n$ and constant completion values $V_1, V_2, \ldots, V_n$, if each process is executed in order of decreasing value density, where the value density of P_i is defined by $D_i = V_i/C_i$ (assuming that processes are ordered $D_1 \geq D_2 \geq \cdots \geq D_n$), the total value $T_i = \sum_{k=1}^{i} V_k$ accumulated at the completion time of P_i ($1 \leq i \leq n$) in the resulting sequence will be at least as great as for any other sequence at time $\sum_{j=1}^{i} C_j$. We break any value density ties by executing the process with the shortest execution time C_i first.

Proof: We prove this theorem by induction on the number of processes.

After the completion of P_1 at time $\tau_1 = C_1$, the accumulated value is V_1. The mean value density at this point is $D_1 = V_1/\tau_1$. If, at this same time τ_1 some other schedule has accumulated a higher value, then its mean value density D is higher, and one or more of its completed processes must have had a higher value density than P_1, a contradiction of the hypothesis.

Similarly, assume that the theorem holds after completion of P_i. After completing P_{i+1}, the total value accumulated so far is T_{i+1}, and the value density of P_{i+1} is V_{i+1}/C_{i+1}. In order for another schedule to have achieved a total value at the completion of P_{i+1} greater than T_{i+1}, one or more of the processes completed since the completion of p_i must have had a value density greater than P_{i+1}'s, a contradiction of our induction hypothesis.

An interesting corollary to this theorem can be conjectured if one imagines that each process in the set of processes described in this theorem are subdivided into successively smaller processes, each with the same value density as its parent, but with a proportionately smaller execution time. Since the sequence of value density is unchanged by this process, it is evident that such subdividing may be carried out indefinitely, resulting in the observation that accruing the value linearly in time as each process executes will produce an optimal total value at any arbitrary time in the sequence, rather than merely at the completion of each process. While a proof for this conjecture could be produced, the conjecture has no direct application to this research, and will therefore be omitted.

Theorem 1, having established the optimality of the total value at the completion of each process with constant value functions, is important in this research in two ways. First, construction of such a schedule off line using the actual process execution times after a set of processes have been scheduled provides an upper bound (although far from a least upper bound) to the total value achievable by any on-line algorithm, and this is used in the experimentation performed as part of this research (see Section 6.6). Second, the importance of the value density in producing a high total value is used as the foundation of the heuristic used to remove overload conditions (see Section 5.3.4).

5.3.2. Best Effort Scheduling Starting Point

To construct our algorithm, then, let us temporarily assume that all our value functions have a constant value prior to their critical times, and that they decrease in some fashion following their critical times. In this case, all processes could be executed at any time after their request times, and the total value accumulated will be unchanged as long as they complete prior to their critical times. In such a case, we take advantage of observation 1 above: we construct a simple deadline schedule using the critical times as if they were deadlines to compute the nearest deadlines. At this point, if all deadlines can be achieved (i.e., there is no overload), the schedule is complete.

In a real-time system, this condition should generally prevail if the system is to be normally well-behaved and able to perform to its specification. Thus our algorithm should perform exactly as well as a deadline schedule when no overloads are allowed to exist, and as long as all value functions have constant values prior to their critical times.

At this point, however, we have grossly oversimplified the definition of a deadline by equating it to the critical time. If the value function is a step function going to zero after the critical time, this definition of a deadline will suffice, but if it decays slowly, perhaps exponentially or quadratically, there is still some value to be achieved after the critical time. For the purpose of this algorithm, we define the "deadline" to be used in our initial deadline schedule to be the latest time at which a process' value drops below some specified percentage (e.g., 90%) of its maximum value.

5.3.3. Increasing Values

We have also oversimplified the situation by assuming that the value functions were constant prior to their critical times. If the value function is rapidly decreasing, there is no problem, since the definition of a "deadline" given above will ensure that execution of such a process takes place in time to complete with a high value, since the deadline schedule should result in a low maximum lateness. If, on the other hand, the value is increasing prior to the critical time, the process should usually be delayed before being allowed to complete, so that as high a value as possible can be achieved.

The question, of course, is how much the process should be delayed before being executed. It seems intuitively clear that if the system is heavily loaded, we must be more

willing to accept a lower value for a process by scheduling it early than if the system is lightly loaded, since the probability that we can schedule the process to terminate at the time of its maximum value is reduced. Similarly, if this process has a relatively low maximum value in comparison with other processes in the system, we should also be willing to accept a lower value by scheduling it earlier. Further, it is clear that these two attributes should be considered together, since in a lightly loaded system, process importance will have little effect on our ability to control process termination times, and similarly, if a process is quite important, the loading of the system can be ignored.

Thus, we use the following heuristic to compute V_{ok}, the minimum acceptable value when this process terminates:

$$V_{ok} = (1 - p_u p_v (1 - a)) v_{max}$$

where p_u is the estimated probability that the system is in an overload condition (including this process in the load), p_v is the estimated probability that another process will have a higher maximum value than this process, v_{max} is the maximum value possible for this process, and a is the minimum fraction of a process' maximum value which we will accept in any case (i.e., we will attempt never to schedule any process to complete prior to the time we can achieve a value of at least av_{max}. The result of using this equation is that the minimum acceptable value will be near to av_{max} only if the probability of an overload and the probability that a higher value process could come in are both high; otherwise, the minimum expected value will be near to V_{max}.

In the experiments conducted as a part of this research, the value a is a policy parameter provided by the user. v_{max} can be determined from the distribution of values across the overall process load using either a discrete or continuous distribution, and is approximated in our experimental implementation by assuming a normal distribution of value function amplitudes across the process load.

Once the minimum value acceptable to the system for this process has been determined, the earliest time at which this value is attainable is computed, the estimated execution time for the process is subtracted, and a scheduling event interrupt is enabled for that time. When that time arrives, the scheduling decision is remade, with the subject process now a candidate for execution.

5.3.4. Overload Processing

Up to this point, we have assumed that the "deadline" for every process which is ready for execution can be met; the only special heuristic needed was to provide for the delay of processes whose value was increasing prior to the critical time. Once a process is declared ready for execution, since all the deadlines are attainable, the simple deadline schedule will guarantee that all processes will meet their deadline.

Now, however, we must consider the situation when the ready processes will not all be able to meet their deadlines because we are overloaded. The overload condition may or may not be temporary, although in a well designed system it should normally be the case that any overload is due to some sort of unusual transient condition in the system environment, and should thus not recur frequently or for a long duration. It is important to observe, however, that it is particularly at such times, frequently at times of stress in the external environment, that an overload is most likely to occur, and is also the time when the system is most needed to respond appropriately to the emergency. For example, it is precisely when a nuclear reactor is overheating that the greatest number of overtemperature, overpressure, and operator commands are entering the system, and therefore that is the moment when it is most likely to be overloaded. A system which responds to such an eventuality by simply stopping, or by executing processes of low importance, has ceased to be useful in controlling an emergency, but has instead become part of the emergency.

The first problem for our scheduler is to determine the probability that an overload has occurred. This is accomplished by the heuristic of considering each process in the deadline order (nearest deadlines first), computing the total available slack time for each process considering the processes which will execute before it in the deadline sequence, computing the variance of the total slack time, and from this information, computing the probability that an overload has occurred (i.e., the probability that the available slack time is less than 0). For expediency in performing this computation, our heuristic assumes that the total slack time exhibits a normal distribution. This assumption is made because the total slack time is computed from the sums of several random numbers (i.e., the expected computation time remaining for each process) potentially drawn from different distributions, and, as the number of such values in the sum increases, the Central Limit Theorem [Allen 78] states that the resulting distribution will show an increasing similarity to a normal distribution.

Having computed, at each point in the deadline ordered execution queue, the probability

that an overload exists, this probability is compared to a user policy value defining the minimum overload probability to declare that an overload exists (for a discussion of the effect of this value on the scheduling performance, see Chapter 7). If the resulting comparison indicates that an overload has occurred, all of the processes involved in the overload at that point are checked to find the one with the minimum expected value density, which is then removed (temporarily[7]) from the deadline sequence. The slack time and overload probability is then recomputed, and this processing continues until the entire remaining set of executable processes has been checked and no overload conditions are found to be present.

This results in a sequence of processes, in deadline order, which do not result in an overload condition at any point in their sequence, and should, therefore, be executable in that order. Given that there are m processors in the system, the first m processes are assigned to them, and their execution commences (or continues if execution is already in progress). Note that a complete sequence for executing these processes is then in existence, and if no further events requiring the sequence to be modified occurs (i.e., a new process arrival or a scheduled process completing with a significantly different process time than expected), the schedule need not be re-computed.

5.3.5. Statistical Computations

We have used the expressions *expected remaining computation time* and *expected value* in the above discussion, but we have not discussed how these values are to be computed. These are simple statistical functions defined in the usual way:

$$E(C)=\int_{t_0}^{\infty} t f(t)\,dt, \qquad E(V)=\int_{t_0}^{\infty} v(t) f(t)\,dt$$

where t_0 is the computation time already elapsed, $v(t)$ is the value if the process is completed at time t, and $f(t)$ is the distribution function at time t given that t_0 units of execution time have already occurred. For each of the distributions used in our experimentation, straightforward polynomial approximations to these integrals (when analytical solutions do not exist) could be used in the computation, but for our experiments an actual numerical integration was used to ensure accuracy for the algorithm simulation. For further information on the statistical functions used, see Chapter 6.

---

[7] i.e., it is removed from consideration for scheduling currently, but is not removed from the run queue; it could be later executed if another processor completes earlier than expected

5.4. Implementation Notes on this Algorithm

The discussion above on the algorithm design was left somewhat general to assist its understanding, but there are several practical details which need to be discussed.

5.4.1. Under-utilized Processors

The overload checking is omitted for the first m processes in the deadline queue, since it seems undesirable to leave a processor idle when work is ready, however unlikely it may be that the work can be completed in time to produce a high value to the system. In many real-time systems, there are a set of "background" processes which are designed to take all idle system resources, such as self-test programs. Such processes in a system with a value function scheduler would be modified to operate as periodic processes with low constant values, and would thus be executed whenever time is available.

5.4.2. Pre-execution of Delayed Processes

Delaying the initiation of a process whose value is increasing has the effect of increasing the load once its ready time is at hand, possibly resulting in a temporary overload. Noting that the value of a process is related to its termination time, it seems probable that the process could be pre-executed as soon as it is requested, as long as it is not allowed to complete prior to its minimum value time. With this in mind, we allow the user to provide a policy value in the form of a standard deviation multiple which expresses the minimum portion of the expected computation time which should be left until the process can be expected to complete with a high value. Using this policy value, the scheduler will usually pre-execute a new process immediately until it is in danger of completing, then it will begin its delay. This capability significantly increases the achievable value in our experimentation, but is kept under user policy control, since it introduces the danger that a process could prematurely terminate with potentially disastrous results. Using a standard deviation multiple to control it would probably be replaced with an actual execution time value for each process in an actual implementation, but that was impractical for our experimentation, since the user determines actual execution times only stochastically.

5.4.3. Overload with a Single Process

The determination of the overload probability using the total slack time at each point in the deadline queue can hide the possibility that an individual process could represent an overload by itself, for which no multiple processing capability could compensate, so the overload detection criterion is applied to both the process' own overload probability and the total overload probability, checking for process elimination in either case.

5.4.4. Interrupting an Overtime Process

An executing process can be interrupted to recompute the schedule if the executing process exceeds its expected computation time plus the minimum expected slack available in the subsequent schedule. This provides the capability to determine if an executing process has exceeded its desirable processing time and may be adversely impacting a higher valued process.

5.4.5. Computational Complexity

It is important to determine the computational complexity of this algorithm, particularly because it is intended for use in a real-time system. For this purpose, we will refer to Figure 5-1 which contains an annotated, skeletal expression of the algorithm described in this chapter. Each statement is numbered, and each loop structure in the algorithm is delineated by a bracket labeled with a single letter for quick reference.

As previously described, it is assumed that a run queue of n processes is input to this algorithm at its entry, and it is our goal to determined the execution time of this algorithm as a function of n. The main body of the algorithm, contained in statements, 1-16 consists of a single loop, examining each of the n processes one at a time. Thus, it is clear from its basic structure that statements 2 through 10 and 16 can be executed at most once for each of the n processes. For each iteration through loop a, statements 11 through 15, comprising loop b, can potentially be repeated once for each of the n processes, assuming that an overload is detected for every process, so that statements 12 through 15 could be executed as many as n^2 times. At this point, it might be observed that loop c could also be executed for every process, making it possible that statement 13 could be executed as many as n^3 times. However, we must note that each iteration of loop b will remove one process from the input queue, so it is not possible for its contents (statements 12, 14, and 15) to be executed more than n times,

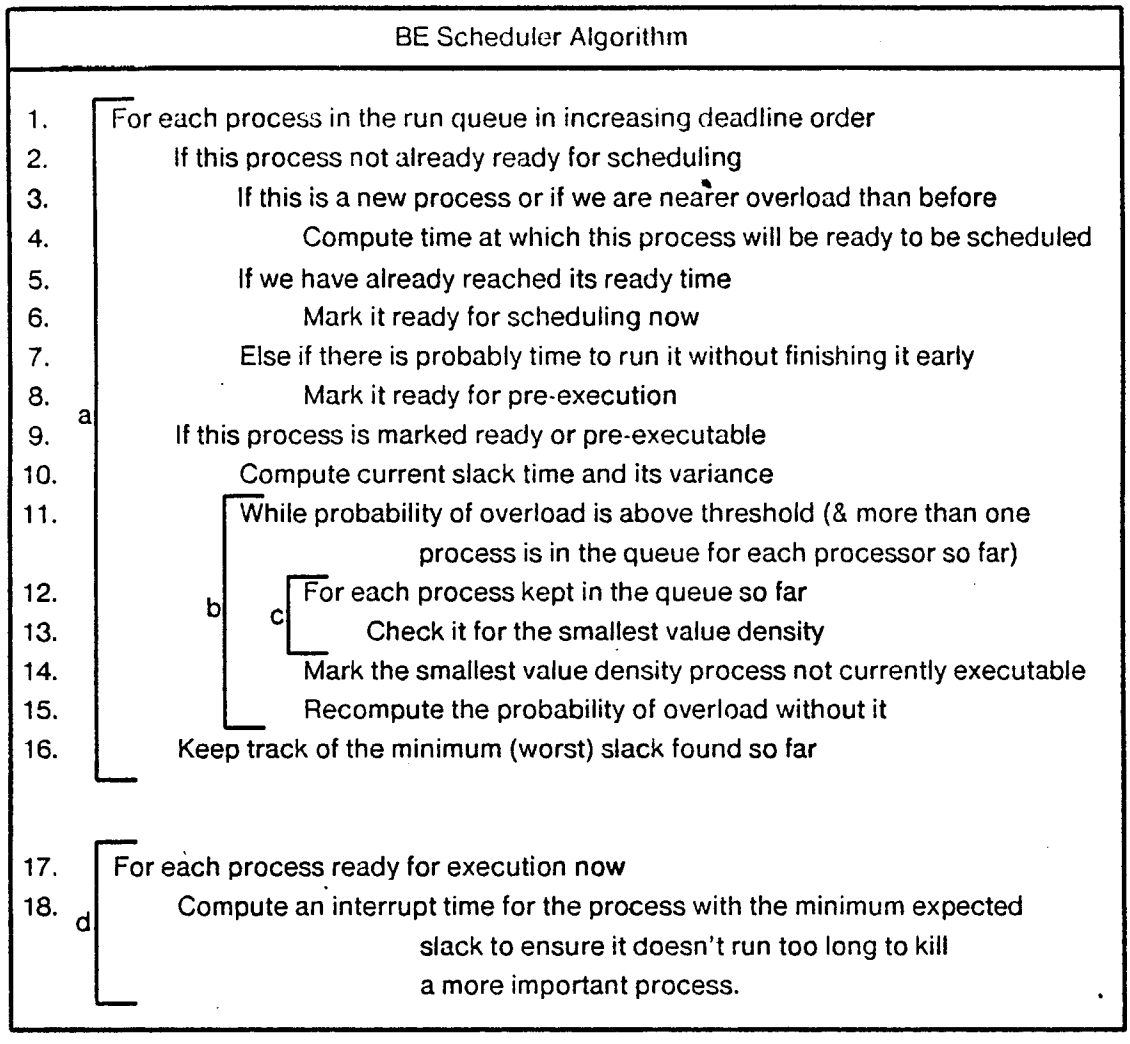

Figure 5-1: Scheduling Algorithm Structure

since only n processes can be removed from the queue. Thus, even if statement 13 could be executed n times each time loop c is invoked, statement 13 can be executed at most n^2 times.

Loop d is a simple loop which can repeat at most n times, so it is clear that the processing of loops b and c dominate the time complexity of the algorithm, and are, in turn, dominated by statement 13 which we have shown can be executed at most n^2 times. Thus, this algorithm can potentially exhibit a worst case time complexity of $O(n^2)$ in the event of overload conditions. If no overloads are detected, the procedure remains $O(n)$. Because of the dependence of its performance on factors beyond the scheduling algorithm itself, the actual time required is best determined empirically, and this measurement is discussed in more depth in Chapter 7.

This computation time is, of course, incurred at each execution of the scheduler. As described earlier, the scheduler is executed at the request and completion of each process, and at other times if key times are encountered. These key times are:

1. When a process with an increasing value function which is being pre-executed must be preempted to prevent it from completing early. This can happen at most once per process.

2. When a process with an increasing value function has now become ready for scheduling (based on reaching a time when the value V_{ok} can be expected). This can happen at most once per process.

3. When a process has executed sufficiently longer than expected that it now jeopardizes another process in the run queue which was not removed to eliminate overload. This could, in principle, occur more than once per process, but the heuristic used to remove overload should prevent it from happening frequently.

Thus, excepting the third possible key time, the scheduler will execute a fixed number of times per process. Assuming a Poisson arrival with mean arrival rate λ, if the scheduler is executed for τ seconds, the number of processes likely to arrive is $n=\lambda\tau$. Since n is the greatest possible size of the run queue, the total time which could possibly be used by this algorithm is $O(n^3)$. In practice, it has been found that none of these key times occur excessively, and in the event of an overload, the number of such interrupts actually taken tends to be lowered slightly, since few processes are pre-executed, and processes aborted before executing do not cause a scheduling computation (See Section 7.4.4).

Chapter 6
Process Scheduling Simulation

Because it is important to provide an environment in which our value-function scheduler and the currently used scheduling algorithms can be evaluated with respect to their performance in generating a high total system value, a simulator has been constructed which provides a complete multiprocessing system environment. This environment consists of a memory of unlimited size connected to one or more processors and a set of data structures defining the processing to be performed by the simulated system.

There are two critical data structures involved in the simulator. The first is a list of the processes available for scheduling during a run. This list is created during the definition of the load and remains unchanged throughout the simulation. The second structure consists of what is usually called a "run queue[8]". This queue is dynamic in that processes are continually entering and exiting the run queue as they become available to be executed (i.e., requested) and are subsequently completed or aborted. The primary purpose of the simulator is to manage these data structures such that the data needed by the various scheduling algorithms will be available at the time that each scheduling decision is to be made.

This simulator, which has been written specifically for this research effort, was written in the C language and has been executed on various host processors, including the DEC VAX 11/780, a Sun 2/120 Workstation, and various versions of IBM Personal Computers[9].

---

[8] It is called a "queue" because it is usually, but not always, implemented as an ordered list of executable processes in order of their intended execution, usually ordered by their fixed priorities in real-time systems. In our context, it is better modeled as an unordered set.

[9] The terms *VAX*, *Sun*, and *IBM* are registered trademarks of the Digital Equipment Corporation, Sun Microsystems, Inc., and International Business Machines, Inc., respectively.

6.1. Simulated Load Definition

The simulator begins by reading a large set of user defined parameters describing the simulation to be performed, then constructing the load to be used during the simulation. This load consists of a set of processes, each of which has an independent execution profile containing its expected computation time distribution from which its actual computation time will be drawn, a value function, and the repetition period for those processes which are periodic.

The resulting set of process descriptions remain fixed during the entire run of the simulator. During the process simulation, all processes executed will be drawn from this set. The load parameters for each of these processes is determined using the user-specified random number generators included among the statistical functions within the simulator (see Section 6.3 below). During execution, more than one instance of each of these processes may be simultaneously available for scheduling, may therefore compete for processing resources, and may subsequently execute concurrently.

The process list contains the following information for each process:

- Process ID -- a number which uniquely identifies an individual process.
- Process Period -- a value which is positive if the process is periodic and represents the interval between request times for that process.
- Process Time Constraint -- the interval between the request time and the critical time for this process.
- Load Parameters -- an execution time distribution with a list of up to five parameters, defining the nature of the load produced when this process is executed. The first parameter is the mean execution time and the second parameter (used for all distributions except exponential distributions) is the standard deviation of the load. The remaining parameters are used only for a bimodal distribution. Of these, the third and fourth parameters are the load and standard deviation of the second peak, while the fifth is the probability that the actual processing time is described by the first peak for a bimodal distribution rather than the second. (for more information on the bimodal distribution see Section 6.3).
- Value Function -- the set of constants defining the value function to be used for this process. These values consist of the five constants describing the value function prior to the critical time, followed by the five constants describing the value function following the critical time, plus the minimum completion value. This minimum value defines the value to be accrued for a process which is finished late (if it is above 0) or aborted.

- Priority -- a fixed value used only by the importance-based fixed priority scheduler, and which is set to the maximum value defined for this process' value function; thus, this value is intended to approximate the importance of this process (see section 6.4.1.)

After the list of processes has been generated, the simulator constructs a list of actual processes to be executed, each with its actual execution time (drawn independently from its individual execution time distribution) and its request times. This list remains fixed for each iteration of the simulation and provides the sequence of process requests for each algorithm, thus ensuring that every algorithm will be responding to exactly the same process load. Note that the actual computation time is thus pre-determined, but is not made available, of course, to the scheduling algorithm under test.

Clearly, it is not possible to simulate our Best Effort algorithm described in chapter 5 with every possible value function, so we have limited the actual value functions to be handled by the simulator to an important subset of possible value functions, selected to provide test cases with a representative sample such as might be used for an actual real-time system. This subset has been defined in a specific form possessing characteristics making the expressiveness of the value extremely flexible while allowing the generation and analysis of the resulting functions to be as straightforward as possible.

Thus we define the value function in two parts; the value prior to the critical time and the value following 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:

$$V_i(t) = K_1 + K_2 t - K_3 t^2 + K_4 e^{-K_5 t}$$

This form provides for functions which can arbitrarily increase or decrease at any rate both before and after the critical time, and can asymptotically approach a limiting value, or precipitously move to extremely high or low values. In addition to the ten value function constants described here, a lower limit is also definable which can be used to provide a lower bound to the value accrued in the event that a process must be aborted. We should note that we are not particularly interested in the degenerate case of value functions which unboundedly increase following the critical time, since such processes would never be scheduled, but would remain pending "forever" awaiting a higher value.

6.2. Simulator Execution

The simulator execution proceeds in three primary steps:

1. It generates the complete set of loads (i.e., the set of processes that are to be executed during this entire simulation). This takes place using user defined parameters which stochastically specify the load to be applied to the multiprocessor.

2. A set of executions of some sequence of process requests is initiated repeating the same set of process requests for every algorithm, then repeating the entire process request sequence generation and every algorithm as many times as the user has specified for iteration.

3. The final statistics are collected from the entire set of iterated runs, correlated, and output both to host processor files for further post-processing and to printed output for evaluation by the user.

The execution of a particular sequence of processes starts by taking the first requested process and placing it into the run queue. Whenever the run queue is non-empty, the simulator calls the algorithm under test to select a process from the queue for executing, then executes that process either until the process completes, until some critical time identified by the algorithm has occurred at which a new decision should be made, or until a new process has become ready to be entered into the queue. At any of these events the executing process is interrupted and the queue is updated appropriately. At each decision point, the simulation statistics of the process being simulated are updated, reflecting any new or deleted processes and accounting for each scheduling-related decision. The scheduling decision is then made again by the algorithm under test. For the purpose of simulation, the decision for the execution of all processors is made simultaneously each time a decision for any processor is needed. Thus it will be true that at all times the set of processes which the algorithm has determined is best run at this time will always be running in all processors attached to the simulated system.

At the end of the simulated elapsed time interval specified by the user for the complete set of requests, the next algorithm to be tested is initiated against the same set of processing requests and the entire processing is repeated for the new algorithm. When all algorithms have completed processing a set of requests, the entire sequence of processing for all these algorithms is again reiterated, starting with the generation of a new random request list. This entire procedure is repeated the number of times specified by the user, thus providing a statistical verification of the performance of each algorithm.

6.3. Simulator Statistical Functions

Because the process scheduling model and the process execution being studied in this research is stochastic in nature, it is necessary that a number of statistical functions be included in the simulator. The required statistical computations appear in several portions of the simulation computation:

- Generation of the process load whose characteristics were determined stochastically by the user.
- Generation of the sequence of processes to be executed (i.e., the run queue) each time an iteration of the simulator is initiated. To compute their arrival times, an exponential random number generator is used, generating a set of Poisson arrival times for the aperiodic processes. The actual execution time for each process to be requested is determined using the user-defined execution time distribution for that process.
- Computing the expected remaining computation time for processes whose execution has been interrupted.
- Computing the values (e.g., value density) to be used in the value function scheduling decisions.

The statistical functions must support five distributions for use in each stage of the simulation. The following subsections describe each of the supported distributions used within the simulator. For each distribution to be used for execution times or value function scheduler decision making, several functions are implemented to provide the statistical computations needed:

- The random number generator function which will produce a pseudo-random sequence for each of the distributions supported.
- A computation of the expected value of a random variable, given a distribution and the constraint that the value cannot be less than some threshold. This function provides, for example, the expected execution time of a process which has already executed for some time.
- The expected value for a process taking into account its value function and the distribution of its execution time. This function can be expressed as:

$$E(V) = \int_{t_0}^{\infty} v(t) f(t)\, dt$$

where $v(t)$ is the value at time t, and $f(t)$ is the distribution function at time t given that t_0 units of execution time have already occurred..

In the simulation, these integrals are computed using straightforward numerical integration to ensure the validity of the simulation results, but in an actual real-time system these functions could be implemented using simple linear or polynomial approximations to avoid unnecessary overhead.

6.3.1. The Uniform Distribution

The uniform distribution is provided although it is generally not used directly by the user. This distribution is the foundation for all of the other distributions and simply consists of the uniform random number generator which generates random numbers in the range $0 \leq x < 1$.

6.3.2. The Normal Distribution

The normal distribution, characterized by its mean and standard deviation parameters, is a commonly used distribution, frequently identified by its characteristic bell-shaped curve. When used for describing execution time, for example, this distribution makes the implicit assumption that the execution time is generally constant, and that the causes of the variation (e.g., differences in a process' input parameters from one execution to another, or interrupt processing unrelated to this process) together generate a normally distributed delay.

A normally distributed random variable, however, may assume any value regardless of its mean, including negative values, although the probability of values far from the mean becomes arbitrarily small. Thus, the random variables with bounded values, such as process execution time which cannot be negative, cannot be accurately modeled by normal distributions, although the normal distribution might provide a sufficiently good approximation to still be useful.

The normal distribution has the important property that a random variable which can be described as the sum of a number of other random variables of *any* distributions having a finite variance will approach a random distribution as the number of its components increases (i.e., the Central Limit Theorem). For this simulation, normally distributed random numbers are generated using the Box-Muller algorithm [Knuth 69]. See Figure 6-1 for an example of a normal distribution curve for the execution time of a process with a mean execution time of 300 milliseconds, and a standard deviation of 100 milliseconds. The distribution function for the normal distribution is

$$f(t,\mu,\sigma) = \frac{e^{-0.5t_1^2}}{\sigma\sqrt{2\pi}}, \qquad t_1 = \frac{t-\mu}{\sigma}$$

where t, μ, σ are the value at which the distribution function is to be computed, the mean, and the standard deviation, respectively.

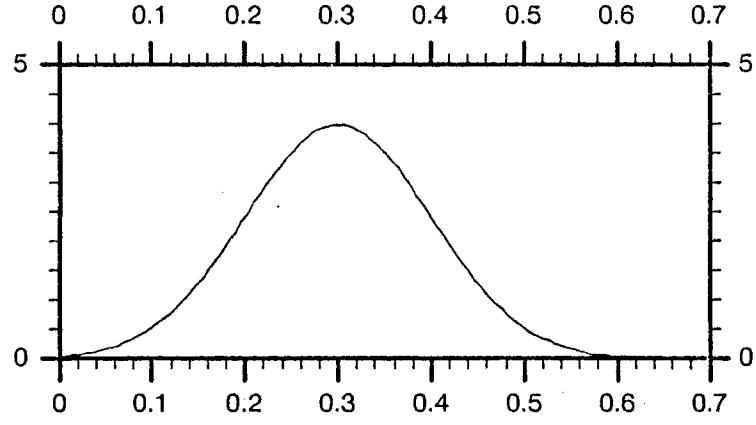

Figure 6-1: Normal Distribution with μ 300 ms., σ 100 ms.

6.3.3. The Lognormal Distribution

The lognormal distribution is one in which the natural log of the random variable (e.g., execution time) is normally distributed, and, like the normal distribution, may be characterized by its mean and standard deviation. The general shape of the lognormal distribution is somewhat bell-shaped, but does not allow negative values, thus making it appropriate for describing such variables as execution time. Thus, the lognormal distribution is frequently used as an approximation to a normal distribution when negative values are to be disallowed.

Lognormally distributed random numbers are generated using the normal random number generator with its mean and standard deviation selected by the uniform random number generator. See Figure 6-2 for an example of a lognormal distribution curve for the execution time of a process with a mean of 300 milliseconds, and a standard deviation of 100 milliseconds. The distribution function for the lognormal distribution is

$$f(t,\mu,\sigma) = f_N(ln(t),\mu_1,\sigma_1)$$

$$\mu_1 = ln(\mu/\sqrt{a})$$

$$\sigma_1 = ln(\sqrt{a})$$

$$a = \frac{\sigma^2}{\mu^2} + 1$$

where t, f_N, μ, σ are the value at which the distribution function is to be computed, the normal distribution function, the mean, and the standard deviation, respectively.

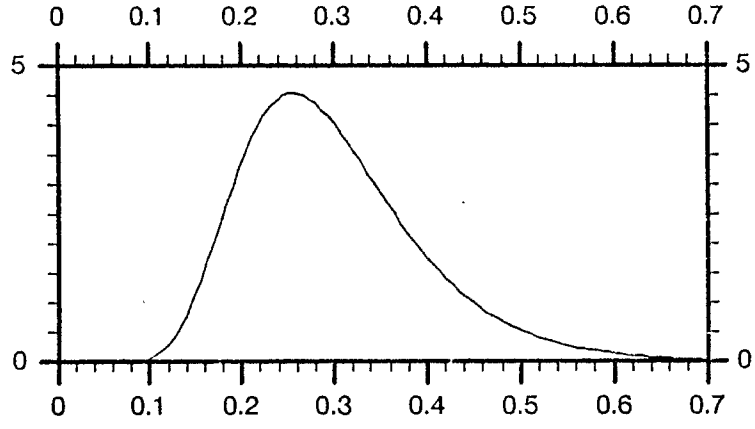

Figure 6-2: Lognormal Distribution with μ 300 ms., σ 100 ms.

6.3.4. The Exponential Distribution

The exponential distribution is completely characterized by its mean, so exponentially distributed random numbers are computed using the uniform random number generator and the exponential operator to generate a set of exponential random numbers with a given mean. This distribution, illustrated in Figure 6-3 by a process with a mean execution time of 300 milliseconds, also disallows negative random variables. This distribution is frequently used for random arrivals of processes, and makes the implicit assumption that the next arrival time is independent of the interval since the last arrival.

When used for process execution times, it assumes that the remaining execution time of a processor is independent of the execution time already elapsed for the process. The exponential distribution has properties which make it quite amenable to mathematical analysis, but is unlikely to accurately describe most actual process execution times. An exception might be when the execution time is dependent on the number of arrivals of some event which is generated by some Poisson process. The distribution function for the exponential distribution is

$$f(t,\mu) = \frac{e^{-t/\mu}}{\mu}$$

where t, μ are the value at which the distribution function is to be computed and the mean, respectively.

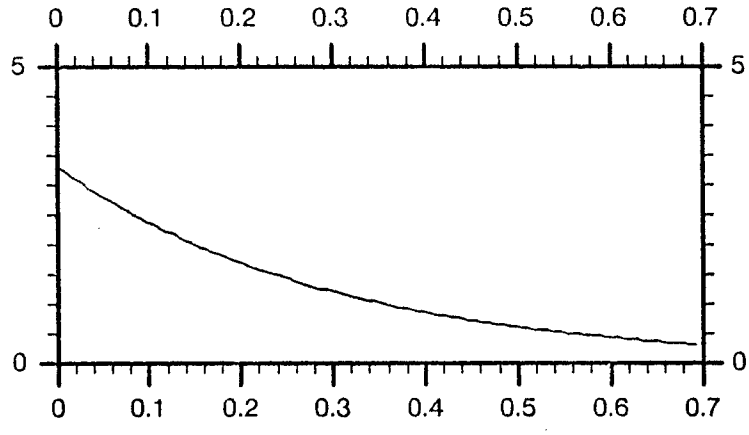

Figure 6-3: Exponential Distribution with μ 300 ms.

6.3.5. The Bimodal Distribution

The bimodal distribution used in this study is a combination of two normal distributions, and carries the implicit assumption that the execution time is normally either of two normally distributed values, with the causes of the variance together resulting in a normally distributed delay. The distribution is characterized by two means and standard deviations, together with the probability that the random variable is controlled by the first rather than the second. See Figure 6-4 for an example of a bimodal distribution curve for a process with a mean execution time of either 300 or 500 milliseconds, a standard deviation of either 100 or 50 milliseconds respectively, and with the probability of .6 that a particular execution will exhibit the first mean. The distribution function for the bimodal distribution is

$$f(t,\mu_1,\sigma_1,\mu_2,\sigma_2,p) = pf_N(t,\mu_1,\sigma_1)+(1-p)f_N(t,\mu_2,\sigma_2)$$

where $t, f_N, \mu_1, \sigma_1, \mu_2, \sigma_2, p$ are the value at which the distribution function is to be computed, the normal distribution function, the means and standard deviation of the two distributions, and the probability for the first one, respectively.

6.4. User Controls

The user of the simulator has a number of controls with which the simulator execution can be tailored to produce a particular scheduling environment. These controls consist of a sequence of selections, beginning with the selection of the scheduling algorithms to be run. All of this user information is placed in a control file and one or more load definition files (see Figures 6-5 and 6-6 for a sample control file and its load definition file) which are read by the simulator during its initial setup, providing the information which is thereafter used to control the simulation.

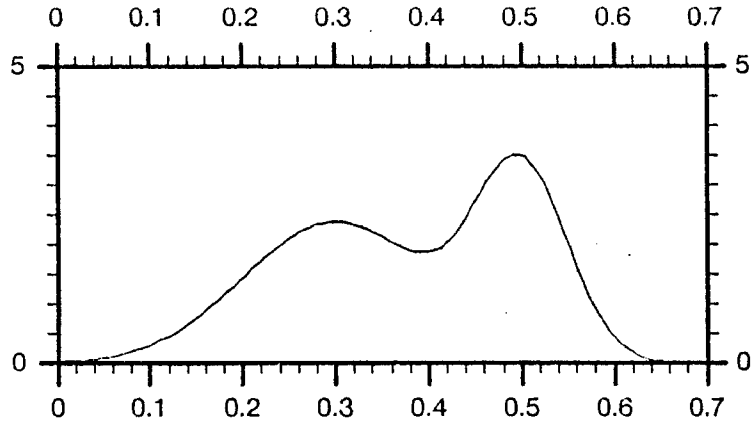

Figure 6-4: Bimodal Distribution with μ_1 300 ms., σ_1 100 ms., μ_2 500 ms., σ_2 50 ms., p_1 .6

6.4.1. Scheduling Algorithm Selection

There are several scheduling algorithms implemented in the simulator besides our value function scheduler, providing the capability for us to compare the performance of the value function scheduler with that obtainable from the other scheduling algorithms used in either real-time or non-real-time systems. These scheduling algorithms, whose performance relative to value function scheduling are described in Chapter 7, are:

- The Shortest Processing Time (SPT) scheduler, in which the process with the shortest expected processing time scheduled first. This algorithm has not generally been used in real-time systems scheduling, probably because of its need for processing time information, but given deterministic execution times, it can be shown (in a single processor) to minimize the average lateness in a set of processes with fixed deadlines. In addition, the SPT scheduler will complete the greatest number of of processes in a given time interval.

- The Deadline scheduler, in which the process with the earliest deadline is scheduled first. The Deadline scheduler can be shown (in a single processor) to generate schedules which minimize the maximum lateness, resulting in the ability to guarantee that if all deadlines in a set of processes can be met, the deadline scheduler will meet them. This algorithm is not generally used in actual real-time systems, both because it requires information on process deadlines, and because its performance is quite poor if every deadline cannot be met. In such an event, the deadline scheduler will attempt to complete processes whose deadlines cannot be met further delaying processes whose deadlines can still be met.

- The Slack Time (i.e., laxity) scheduler, in which the process with the smallest excess available time (i.e., time before the deadline minus the expected computation time) is scheduled first. This scheduler can be shown to produce a schedule in which the minimum lateness is maximized (see, for example, [Baker 74], page 25), a property of dubious usefulness, but it can also be shown that it

```
Simulation Description:
General Test Simulation Run (Old algorithm)
Algorithms
(MaxVal, SPT, Deadline, Slack, BEValue, FIFO, Rand, RandP, DeadP, VDensy):
BEValue, SPT, Deadline, FIFO, RandPRTY
End
Number of Processors
2
Number of iterations:
20
Delay processes until value function goes positive?
no
Abort processes whose value drops below zero?
yes
Print Summaries (prior to final summary)?
no
Print Schedules?
no
Print Queue whenever decisions are made?
no
Put summaries in a file?
no
Put schedules in a file?
no
Print BEValue scheduling decision information?
no
Total elapsed simulation time:
20.
Random number seed:
14000
Cost Factors (usec.) (time to compute ready time, check ready time, get proc,
                     remove proc, check proc for overload, compute expected value)
100, 20, 100, 100, 25, 400
Algorithm's assumed execution time distribution (Normal, Exponential, LogNormal)
Normal
Mean acyclic arrival time distribution & parms. (Uniform, Poisson, LogNormal):
Poisson 20
Spike acyclic arrivals (repetition period, duration, distribution, mean)
4.0 0.2 Poisson 1.0
Threshold (n*std. dev) before which a preview must be completed:
4
Overload probability threshold for removing low value density processes
.4
Value threshold to define "deadline" (% max)
90
Minimum value before which a process should not be allowed to complete (% max)
20
Cost (Microseconds) of a context swap (Charged for preemptions only)
400
Load Groups (No. processes each group and the group control file):
40 loadfile1
```

Figure 6-5: Sample Scheduling Simulator User Control File

```
Mean Execution Time Distribution (Normal, Poisson, LogNormal, Bimodal):
Normal .4 .3
Mean St. Dev. (Mean % of execution time, % Std. Dev.):
30 0
Mean Deadlines (Mean % of execution time, % Std. Dev.):
250 50
Percent Periodic:
10
Stochastic value distribution (Normal, Exponential, LogNormal, Bimodal)
Normal
Overload value functions (mean, standard deviation of K1, K2, K3, K4, K5, K6)
0.0, 0.0  0.0, 0.0  0.0, 0.0  5.0, 4.0  6.0, 0.0  0.0, 0.0
Underload value functions (mean, standard deviation of K1, K2, K3, K4, K5, K6
5.0, 4.0  0.0, 0.0  0.0, 0.0  0.0, 0.0  0.0, 0.0  0.0, 0.0
```

Figure 6-6: Sample Scheduling Simulator Load Definition File

will, as with the Deadline scheduler, result in the ability to guarantee that if all deadlines in a set of processes can be met, the deadline scheduler will meet them. This algorithm, also, is not generally used in actual real-time systems since it requires not only information on process deadlines, but also computation time, with no improvement in performance over the Deadline schedule. The slack time scheduler, in a multi-processor system, has the further disadvantage that when preemption is allowed, it results in an unstable schedule; whenever the scheduling decision is repeated, processes which have not been executed recently (whose slack time is necessarily decreasing) will preempt processes which have been executing (whose slack time is therefore constant), resulting in excessive preemptions.

- The FIFO (First In First Out) scheduler, in which the oldest processor in the run queue is executed. This scheduling algorithm is frequently used in real-time systems incorporating message passing between processes. FIFO scheduling promotes fairness among a set of processes competing for resources, but the FIFO schedule optimizes no performance parameters.

- The Random scheduler, whose position in the run queue is selected at its request time randomly, with uniform distribution. Once in the run queue, processes are scheduled in their queue order, preventing senseless preemption. There are no real-time systems which schedule processes randomly in this way, but processor interrupts frequently occur with essentially random priorities. The inclusion of this scheduler illustrates the performance of a system in which no process or state knowledge is used in the scheduling decision, contrasting with all the other algorithms in which more or less process and state information is used to make scheduling decisions.

- A Fixed Priority Scheduler in which the fixed priorities for each process in the run queue are set at request time based on each process' elapsed time to its deadline. Once requested, the highest priority process in the run queue is scheduled at each decision time. Most real-time systems use this algorithm for

scheduling aperiodic tasks, although the fixed priorities are usually fixed by the system designer based on knowledge of the process' importance as well as its deadline criticality.

- A Fixed Priority Scheduler in which the fixed priorities for each process in the run queue are set at request time based on an estimate of the importance of the process (for this simulator, importance is determined by computing the maximum value defined for its value function). Once requested, the highest priority process in the run queue is scheduled at each decision time. Most real-time systems use this algorithm for scheduling aperiodic tasks, using importance as the principal priority determination, adjusting the priorities during system test as needed to meet performance specifications.

6.4.2. Process Set Performance Parameter Selections

There a number of parameters which determine the resource utilization/performance of the set of processes to be simulated. These parameters include:

- The number of processors to be made available (up to 20) for executing the processes in the run queue.
- The number of times the entire set of schedulers is to be run, using a new sequence of process requests from the process set each time.
- The total simulated elapsed time that each simulation run is to take.
- The mean aperiodic arrival time distribution and its parameters (all simulations performed in this research effort use an exponential distribution.)
- For simulations in which times of increased periodic loads (i.e., load "spikes") are desired at periodic intervals, the spike aperiodic arrival parameters including the spike repetition period, the duration of the spike period, the aperiodic arrival distribution to be used for arrivals during the spike, and the mean arrival time to be used for the duration of the spike period.

6.4.3. Value Function Scheduler Parameters

The value function scheduler has a number of "tuning" parameters which will determine the nature of its scheduling decisions. These parameters are critical to its performance, and may therefore be set by the user. These parameters are:

- The percentage of a process' maximum value which will be define a deadline for the purpose of determining an overload condition for the value function scheduler. The deadline of a process is defined to be the latest time at which this process may terminate to achieve at least this percentage of its maximum value.
- The overload probability threshold at which low value density processes should

be removed. This value defines the meaning of "overload" for the value function scheduler; whenever the scheduler determines that the probability the system will miss a deadline (see previous tuning parameter) exceeds this value, it will remove enough low value density processes to remove the overload.

- The process execution time distribution which is to be assumed by the value function algorithm,

- Minimum value (expressed as a percentage of maximum value) acceptable for any process before which the process should not be allowed to complete. The value function scheduler will delay the completion of a process, if necessary, until it is likely that the completion value will exceed this percentage.

- Amount of a process' expected computation time which may be used by the process before it must be delayed to achieve the value described by the previous parameter. This parameter, expressed as the number of computation time standard deviations which must be reserved for processing after the delay, allows the scheduler to pre-execute processes when processor time is available, then delay completion until a high value becomes possible.

- The execution time cost to be used for each constant time segment of the value function algorithm so that its total cost can be computed. These costs are described more completely in Chapter 7.

6.4.4. Process Set Specification

In defining the process load to be imposed, the user divides the total set of processes to be defined into one or more groups. For each group, the number of processes in the group are specified, along with several parameters which determine its scheduling and execution characteristics. These parameters define distributions which are used during initialization to compute the actual parameters for each process, which will then remain fixed throughout the simulation. These parameters are:

- The execution time distribution and its parameters for processes in this group.

- The relationship between the process' critical times and their execution times, expressed as a stochastic function of the execution time for each process.

- The probability that each process in this group will be periodic. If the process is periodic, its period will be stochastically related to its critical time.

- The mean and standard deviations of each constant defining the value functions for this process group. The actual value function constants for each process are defined using a normal distribution over these values.

6.4.5. Simulation Execution Controls

The user may specify a number of parameters to control the actual simulation execution. These parameters include both the simulated environmental conditions under which the processes are to execute, and the generation of output by the simulator for subsequent processing and evaluation. We should note that if all of these output controls are enabled, a very large quantity of information will be generated which will be very hard to handle. This information is critical, however, for the evaluation of individual decisions and individual algorithms. These controls take the form of yes/no questions to be answered by the user:

- Should processes be aborted when their value function drops below zero?
- Should simulation summaries be printed after each iteration of the simulation?
- Should the actual schedules resulting from each of the algorithms during execution be printed?
- Should the entire contents of the run queue be printed whenever a decision is being made?
- Should the summaries for each iteration be placed in a file for post-processing?
- Should the schedules themselves be placed in a file for post-processing?
- Should the information on which the value function scheduling decisions are made be printed to provide for subsequent analysis?

In addition, the user may specify the cost (in microseconds) of a context swap (i.e., the exchange of one process for another in a processor in which the first process has not completed) to be charged when a preemption is performed. This time is used by the simulator to delay each such preemption, allowing the preemption costs to be taken into account for each algorithm.

6.5. Total Value Upper Bound

For each set of process requests, an upper bound to the total value attainable in the elapsed execution time of the simulator is computed by using the algorithm described in Theorem 1 in Chapter 5. This value is obtained after the simulation is completed by sorting all the requested processes by their actual value density, then adding their respective values until the elapsed simulation time has been allocated. The value density used for this computation is obtained by dividing the maximum value obtainable for the process by its actual execution time.

This bound is quite optimistic (i.e., it is not a least upper bound) in that it assumes that the maximum possible value could be achieved for every process and does not take into account the fact that in an overload condition, many of the processes would have been required to compete with each other, and would therefore not have been schedulable with positive values.

Since we do not have a least upper bound, it would be interesting to know how optimistic our upper bound is. While we cannot answer this question definitively, we did experiment with it for one of our simulation loads. Using a single processor and an average load of about 165%, we found that our Best Effort (BE) and the Value Density (VD) schedulers generated about 79% of the upper bound value. Temporarily modifying the BE scheduler to use the actual execution times (rather than the expected ones) for its scheduling decisions, its total value increased to 87% of the upper bound, while the VD scheduler with a similar change increased to 83% of the upper bound. At the same time, we attempted a schedule "by hand", which achieved about 82% of the upper bound. Such a set of tests is by no means conclusive, but we felt that it is unlikely that the actual least upper bound is greatly higher than that generated by the modified BE algorithm.

6.6. Data Collection and Output

One of the most important functions of the simulator is to collect statistics describing the performance of each scheduling algorithm, with the ability to write this information either at the end of each iteration or at the end of the entire simulation run, generating this output either to a mass storage file for post-processing analysis or to a display device for immediate viewing by the user, or both.

This output consists of a large amount of data, completely describing the simulation performed, including not only the statistical information describing the algorithm performance, but also the information used by each algorithm to make its scheduling decisions; thus, the decisions may be evaluated after the simulation. The information thus generated includes:

- The complete list of processes from which the request list is taken and the parameters used to describe each process' execution (i.e., execution time distribution, critical time interval, value function constants, and period for periodic processes)

- Each process request list which was generated at the beginning of each iteration

of the simulator (i.e., the time at which each process was requested, the actual execution time for that process request instance, and the identification of that process)

- The complete schedule generated by each algorithm (i.e., the actual start and stop time of each process, including any preemptions), including the amount of time it executed during the entire iteration.

- The total value upper bound which is computed by the simulator for each process request list, providing for an evaluation of the value function scheduling algorithm.

- A number of statistical values, maintained separately for each algorithm, and averaged over each of the iterations performed:

    - Maximum queue size -- the largest size the run queue ever reached during the execution of the simulator for each iteration.

    - Mean number of preemptions generated.

    - Mean lateness generated. This number is negative if the process on average completed prior to its critical time.

    - Maximum lateness generated by this algorithm in an iteration.

    - Mean tardiness generated by this algorithm in an iteration. This value can never be negative but becomes greater than zero whenever a process is exceeds its critical time.

    - Total value generated by this process.

    - Number of tardy processes produced by this algorithm.

    - Number of processes aborted due to the algorithm.

6.7. Simulation Post-Processing Analysis

A number of routines have been generated in conjunction with the simulation effort to do post-processing analysis of the simulator outputs. The results of these routines are used to generate the charts and graphs included in the evaluation chapter (see Chapter 7) of this report. These routines generate such graphic output as:

- Gantt charts illustrating the actual schedule generated by an algorithm during one of its iterations.

- A chart showing the actual execution period for each execution of process. This chart consists of a set of Cartesian time graphs (time increasing to the right) for

which the time scale for each graph is defined by its request time at the left, and the longer of (1) the process' longest actual termination time in the simulation iteration or (2) 130% of the process' critical time interval measured from the process request time defining the left end of the graph. The vertical axis for each such graph corresponds the process' value, with 0 on the bottom and 120% of the maximum value of all processes in the system at the top. On each graph, the value function is plotted, superimposed with a line showing each of the actual execution times for each request of the process. The vertical placement of the process execution lines shows the first execution at the top and the last at the bottom, allowing them to be viewed chronologically. Thus these graphs illustrate the time relationship between the value function and the process initiation and termination times for each of process instances that was scheduled, graphically illustrating the scheduling performance for each process with respect to the amplitudes and shapes of value functions.

- A clock-like diagram which compares a large number of simulation runs to each other indicating by the distance from the center of a circle the relative value generated by each algorithm for each of the cases presented. The length of each line is determined by the total value achieved, with high values generating the longer lines. The maximum length of any line in a set of such graphs is scaled to the highest value attained for any algorithm in the set, with all lines for other algorithms using the same scale. This allows a large number of data points for separate simulation executions to be made visible on a single graph at one time.

- A graph showing the the time varying load actually generated by the simulation, and the apparent load as estimated by the value function algorithm during its computation.

6.8. Simulator Validation

In any research evaluation depending on computer simulation, the problem of ensuring that the simulator results are valid is critical. In general, validation can be broken into two slightly different concepts: verification of the implementation of the simulator, and validation of the model implemented by the simulator.

For this simulator, both the verification and validation of the simulator were handled through the use of extensive trace facilities built into the simulator. As shown in Figure 6-5, there are a number of outputs which can be generated using the control files, in addition to other trace outputs which can be enabled as debug output when the simulator is compiled. Using a large number of such detailed traces, the ability of the simulator to generate and execute a sequence of processes under control of each of the schedulers was verified.

Since there does not yet exist a "real-world" system which can be compared to this system,

and because the portion of a real system which is simulated is conceptually very simple, validation techniques involving comparison of the output of the simulator to an actual system were not used. However, the execution of a sequence of processes in an actual multiprocessor is quite easily visualized, so our printed traces seemed adequate. As a later part of the Archons effort, an actual implementation of this scheduling technique is expected to be constructed, so comparisons with the output of this simulator can then be made.

Figure 6-7 is a brief sample of such a trace, showing the decisions being made for a few milliseconds of execution during a period of overload. The data in this figure is, of course, somewhat cryptic. The first line, for example, shows the process attributes for process 7, which is shown to be an aperiodic process with 857 ms. between its request time and its critical time, with a normally distributed mean execution time of 310 ms. and a standard deviation of 93 ms. Its value function is constant at 7.7 prior to its critical time, and 0.0 following its critical time. Following the ellipsis several lines lower, we find that process 7 began execution at time 4.002, with an expected execution time of 612 ms., but with an actual execution time of 249 ms. At time 4.190, a new request was made, for process 27, so the scheduling decision was remade, eventually scheduling process 27 both because of its impending critical time 86 ms. later, and its high value density of 295.7.

Extensive traces such as these were used to verify that each decision was made at the appropriate times, that the value, time, and statistical computations were made correctly, and that the multiprocessor was correctly simulated, especially the real-time clocks of each processor.

```
 7            0.857     Normal   0.310   0.093
  K = early(   7.7   0.0   0.0   0.0   0.0), late(   0.0   0.0   0.0   0.0   0.0)
10            0.989     Normal   0.308   0.092
  K = early(   9.1   0.0   0.0   0.0   0.0), late(   0.0   0.0   0.0   0.0   0.0)
13  3.139     2.264     Normal   0.795   0.238
  K = early(   3.8   0.0   0.0   0.0   0.0), late(   0.0   0.0   0.0   0.0   0.0)
14            1.697     Normal   0.673   0.202
  K = early(   5.4   0.0   0.0   0.0   0.0), late(   0.0   0.0   0.0   0.0   0.0)
20            1.102     Normal   0.454   0.136
  K = early(   7.1   0.0   0.0   0.0   0.0), late(   0.0   0.0   0.0   0.0   0.0)
22            1.532     Normal   0.730   0.219
  K = early(   1.4   0.0   0.0   0.0   0.0), late(   0.0   0.0   0.0   0.0   0.0)
27            0.184     Normal   0.111   0.033
  K = early(   6.7   0.0   0.0   0.0   0.0), late(   0.0   0.0   0.0   0.0   0.0)

                                  . . .

 4.002  1    7                    0.612       0.249
D 4.190:27(0.060,0.086),7(0.435,0.669),7(0.612,0.779),22(1.261,0.910),
        20(0.894,0.944),10(0.608,0.955),13(1.567,1.686),14(1.327,1.697),
    id=27,ev/ci=295.723572     id= 7,ev/ci= 48.357491     id= 7,ev/ci=42.072117
    id=22,ev/ci=  1.663851     id=27,ev/ci=295.723572     id= 7,ev/ci=48.357491
    id= 7,ev/ci= 42.072117     id=20,ev/ci= 17.284143     id=10,ev/ci=34.728355
    id=27,ev/ci=295.723572     id= 7,ev/ci= 48.357491     id= 7,ev/ci=42.072117
    id=10,ev/ci= 34.728355     id=13,ev/ci=  5.381019     id=14,ev/ci= 8.161775
  id=27,ci=0.060078
  id=7,ci=0.435122
  id=7,ci=0.611879
  id=10,ci=0.607701
  id=14,ci=1.326906
 4.190  1   27                    0.060       0.023
D 4.199:27(0.051,0.076),7(0.435,0.660),7(0.612,0.769),22(1.261,0.901),
        20(0.894,0.935),10(0.608,0.946),22(1.440,1.532),13(1.567,1.677),
        14(1.327,1.688),
    id=27,ev/ci=493.923431     id= 7,ev/ci= 48.357491     id= 7,ev/ci=42.072117
    id=22,ev/ci=  1.663851     id=27,ev/ci=493.923431     id= 7,ev/ci=48.357491
    id= 7,ev/ci= 42.072117     id=20,ev/ci= 17.284143     id=10,ev/ci=34.728355
    id=27,ev/ci=493.923431     id= 7,ev/ci= 48.357491     id= 7,ev/ci=42.072117
    id=10,ev/ci= 34.728355     id=22,ev/ci=  1.366095     id=27,ev/ci=493.92343
    id= 7,ev/ci= 48.357491     id= 7,ev/ci= 42.072117     id=10,ev/ci=34.728355
    id=13,ev/ci=  5.381019     id=14,ev/ci=  8.161775
  id=27,ci=0.051160
  id=7,ci=0.435122
  id=7,ci=0.611879
  id=10,ci=0.607701
  id=14,ci=1.326906
 4.213  1                                                27      -0.063     6.702
D 4.213:7(0.435,0.647),7(0.612,0.756),22(1.261,0.888),20(0.894,0.922),
        10(0.608,0.933),22(1.440,1.519),13(1.567,1.664),14(1.327,1.674),
    id= 7,ev/ci=48.357491     id= 7,ev/ci=42.072117     id=22,ev/ci= 1.663851
    id= 7,ev/ci=48.357491     id= 7,ev/ci=42.072117     id=20,ev/ci=17.284143
    id=10,ev/ci=34.728355     id= 7,ev/ci=48.357491     id= 7,ev/ci=42.072117
    id=10,ev/ci=34.728355     id=22,ev/ci= 1.366095     id= 7,ev/ci=48.357491
    id= 7,ev/ci=42.072117     id=10,ev/ci=34.728355     id=13,ev/ci= 5.381019
    id=14,ev/ci= 8.161775
  id=7,ci=0.435122
  id=7,ci=0.611879
  id=10,ci=0.607701
  id=14,ci=1.326906
  4.213  1              7         0.435       0.159
```

Figure 6-7: Sample Simulator Trace Listing

Chapter 7
Evaluating a Best Effort Scheduling Algorithm

7.1. Introduction

In this chapter, we demonstrate the performance of the scheduler described in Chapter 5 (called the BE scheduler in this chapter) in as close to a fully realistic real-time execution environment as possible. Using the simulator described in Chapter 6, we have completed a large number of simulations of this scheduler, comparing its performance with eight other schedulers chosen either because they are commonly used in existing real-time systems, or because they have been proposed for, or studied in, the context of real-time systems.

Our goal is to demonstrate the highly **consistent** performance of the BE scheduler in extremely stressful environments made possible by explicitly using the time-value functions to make time-driven scheduling decisions. In the experiments described in this chapter, we utilize the upper bound value computed by the simulator (described in Chapter 6), measuring all values achieved as a fraction of the upper bound value. In performing this evaluation, we begin in Section 7.2 by evaluating the process load sets which need to be applied to each of the schedulers to demonstrate their scheduling properties with respect to total system value, choosing a specific set of value functions from which the loads used in the remainder of this chapter will be drawn. In Section 7.3, we describe a set of experiments from which the "tuning" parameters for the BE scheduler can be selected to be used for the remainder of the experiments. The experiments showing the effects of varying the processing load on each scheduler are described in Section 7.4.

With a heuristic scheduler such as the BE scheduler, there are specific scheduling situations in which we would like to evaluate the scheduling decisions in some detail, and this is done for several interesting cases in Section 7.5. In addition, such heuristic schedulers are, of course, sub-optimal, and it is important to understand the principal areas of weaknesses of the scheduler, which we discuss in Section 7.6, followed by a brief summation of the BE scheduler experimentation in Section 7.7.

7.2. Definition and Rationale of Loads to be Applied

The performance of any scheduling algorithm is completely dependent on the processing load to which it is subjected, so the definition of the load to be used in this experiment set is the first topic to be discussed. It is our goal to define a set of loads which is representative of the real-time environment to which our algorithm is targeted, but also one which will stress the BE scheduling algorithm as well as the more traditional scheduling algorithms with which ours will be compared.

7.2.1. Methodology for Load Definition

As the experimental results confirm, we would expect that almost any reasonable scheduler algorithm will perform acceptably in an environment which produces no overload and/or is characterized by no deadlines. This is also the case if the process value functions are flat prior to their critical times, and with critical times which provide an abundance of slack in which all the processes may be scheduled. Conversely, an environment containing occasional overloads or containing processes with low slack prior to critical times will stress any scheduling algorithm, and will clearly point out the differences in scheduling performance between the various algorithms.

There are a number of attributes which define the actual load to be applied, and we describe here the criteria which we use to decide the nature of these loads. The attributes which we can vary in each load are:

- Processing time required for each process.
- Number of periodic processes in the load.
- Elapsed time from request time to critical time.
- Value function.
- Aperiodic arrival times.

Each of these attributes actually consists of a set of related parameters which together define the load to be applied during an experiment. In the remainder of this subsection, we describe the nature of each of these attributes, and the parameters we will use to define it.

7.2.1.1. Processing Time Required for Each Process

The processing time for each process is normally a stochastic value characterized by its distribution and the parameters defining that distribution. In an actual real-time system, the distribution will be defined by the application design, and would most likely differ for each process, but would probably not be known precisely in advance. Since a knowledge of the distribution is important to the scheduling decision process, our BE scheduler will assume that a processing time distribution is known for each process. The importance of knowing this distribution accurately is demonstrated by the set of experiments described in Section 7.4.2.5. Every possible processing time distribution is defined by its density function and the parameters controlling that function. In the simulator used for this set of experiments, there are four possible execution time distributions supported (i.e., Normal, Lognormal, Exponential, and Bimodal) described in Chapter 6.

7.2.1.2. Number of Periodic Processes in the Load

In a typical set of processes constituting a real-time load, both periodic and aperiodic processes are present. Periodic processes are those whose request times occur at regular intervals, while aperiodic process requests occur based on some external stimulus (e.g., operator switch depression or sensor reading). In our experiments, we can control the percentage of the processes which are to be periodic. Once defined, the periodic processes will be requested at their individual periods, while the aperiodic processes will show exponentially distributed arrival times.

7.2.1.3. Elapsed Time from Request Time to Critical Time

The interval from the request of a process to its critical time determines the slack time for that process (if any), and thus determines the criticality of the scheduling decisions which will be made. Even an apparently underloaded real-time system will perform poorly if processes are frequently completed at the wrong time, and this value will determine the ease with which a set of processes can be scheduled to provide a high system value.

7.2.1.4. Value Function

The shape and amplitude of the value function will, of course, be the primary component of the process attributes which will determine the performance of the BE scheduler for a set of processes. The implementation of the BE scheduler places no restriction on the value function (see Chapter 5), but for the purpose of this experimentation we have provided for the

experimenter to use a pair of very flexible continuous functions, one defining the value prior to the critical time, and one defining the value following the critical time.

Thus, the value functions to be used for these experiments are not competely general forms, but rather are limited to a specific form providing for shapes with strong similarity to those expected in actual real-time systems (see Chapter 4 for a discussion of the relationship between value functions and scheduling policy). 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. For each of these two parts, five constants are used to define the value function relative to the critical time using the expression:

$$V_i(t) = K_1 + K_2 t - K_3 t^2 + K_4 e^{-K_5 t}$$

7.2.1.5. Aperiodic Arrival Times

The frequency of the exponentially distributed acyclic process arrivals are determined by the user specification of the mean time between arrivals. This value is used for all acyclic arrivals during the simulation period.

In addition, our experimentation provides for "spike" loads at regular intervals, during which the aperiodic load can substantially increase for a brief period of time. This allows us to simulate significant events in a real-time processing experiment such as emergency interrupt processing, multiple simultaneous sensor arrivals, etc. In our simulation, we provide for these spike loads to arrive in addition to the normal load, specifying an additional mean time between acyclic process arrivals to be imposed during the spike. In setting up the experiments, we specify the period at which these spikes will arrive and the duration of each spike.

7.2.2. Description of Each Load Type

For the purpose of this experimentation, we define 16 different process sets which can be used for any experiment, either homogeneously (i.e., all processes in an experiment are drawn from the same set) or heterogeneously (i.e., processes in an experiment are drawn from more than one set). We have defined these sets to represent as many load conditions found in actual real-time systems as possible with a minimum number of load sets. Of course, not every real-time system can be represented with a finite number of such sets, but we believe that the sets we have defined cover an extremely large subset of real-time systems.

The 16 sets consist of four variations of four basic sets. The four basic sets correspond to a single load level each with a different value function shape. The four variations on these sets each use a different process execution time distribution (see Paragraph 7.2.1.1 above). The 16 sets are defined in Figure 7-1.

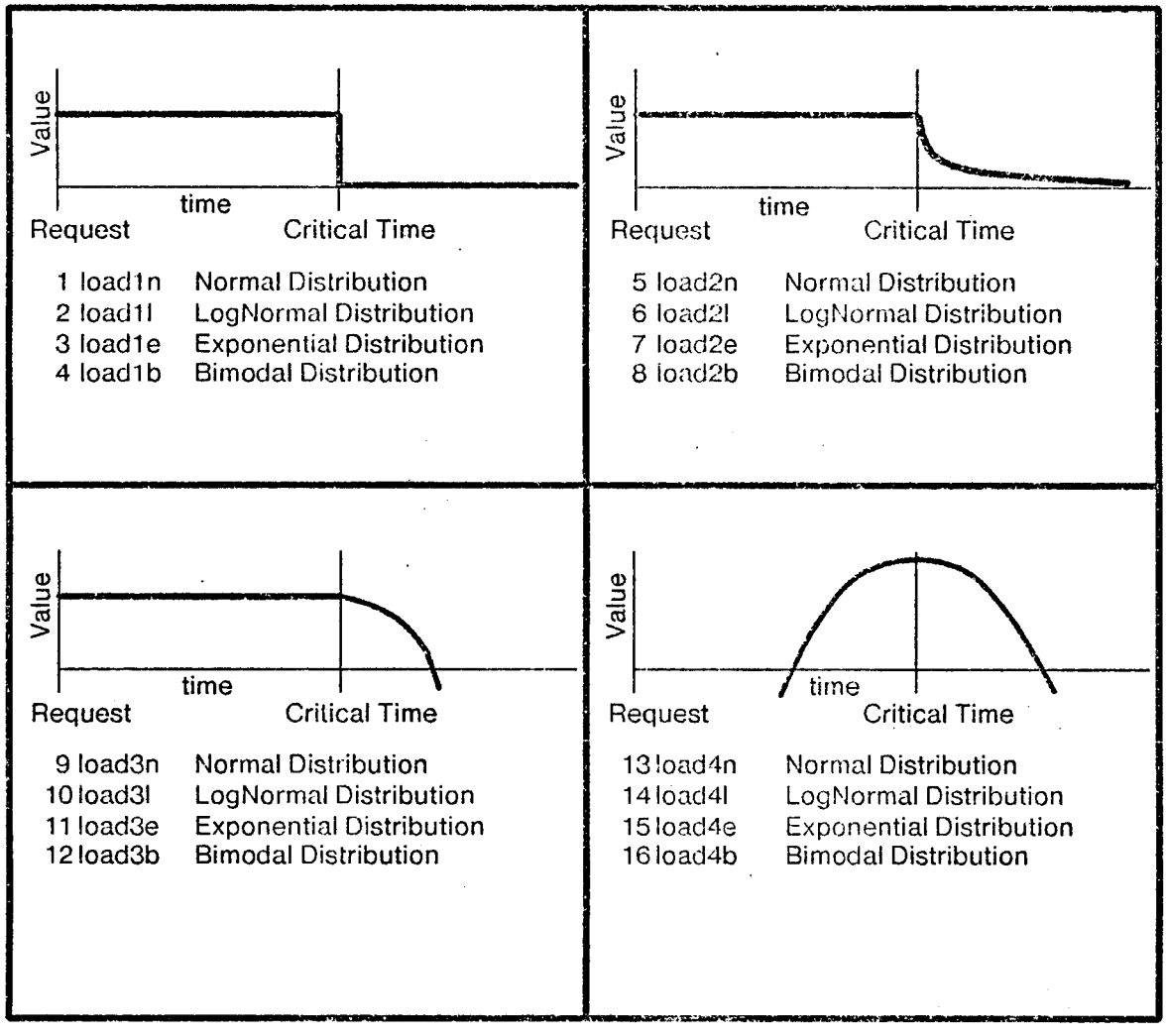

Figure 7-1: Sixteen Load Sets for Experiment Loads

The four basic sets use the value functions illustrated in Figure 7-1. The first value function (used in sets 1 through 4), represents a hard deadline process, in which there is value to the system for completing it only if it precedes a 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 second value function (used in sets 5 through 8), 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 third value function (used in sets 9 through 12), is constant prior to the deadline, but drops off quadratically 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 500 ms. following the critical time). The fourth function(used in sets 13 through 16), is quadratically rising before, and falling after, the critical time, representing a process whose completion should be delayed until after some arbitrary time, such as a satellite orbit insertion burn, 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 500 ms. before and after the critical time).

The overall mean execution time (400 ms.) and its standard deviation (300 ms.) have been used to provide for a load to be generated which will have a wide variety of short and long processes. While many of the processes will execute for about 400 milliseconds, the overall standard deviation ensures that both very short (<10 ms., but not less than 1 ms.) and very long (>1 second) processes will be included in the process list. The overall load means and standard deviations are used to produce a set of processes, each of which will have its own distribution parameters. The requests of each of these processes will then create a stochastic sequence of processes for each experiment. Thus the means shown are overall mean execution times for the entire set.

7.3. Tuning the Algorithm for Each Load

In defining the BE scheduling algorithm, we have described four parameters which can affect the performance of the algorithm in different environments. In this section, we discuss the effect of varying each of these parameters, and show the results of a set of simulations in which these parameters have been changed to verify the performance described.

7.3.1. Description of Each Tuning Parameter

The four parameters are:

- Overload probability threshold θ -- the overload probability above which low value density processes must be removed from the processing sequence being generated.
- Deadline threshold ν -- to define the meaning of a "deadline" in this system.
- Minimum value λ -- used to determine the earliest time at which a process should be allowed to complete.
- Pre-execution Limit T_λ -- in the event a process is started before it could be expected to complete after λ is reached, the amount of processing time which must remain to be completed after its value has reached λ.

7.3.1.1. Overload Probability Threshold

The BE scheduler defined here depends for its performance on its ability to accurately detect the existence of an overload. As described in Chapter 5, this is done by computing the probability that the addition of a process to the list of executable processes will result in a negative slack time. When this probability exceeds θ, the overload probability threshold, the system is declared to be in an overload condition, and one or more processes (based on their value density) are removed until either one process is left, or the overload condition is removed.

7.3.1.2. Deadline Threshold

The BE scheduling decision procedure starts with the construction of a deadline schedule (i.e., a sequence of processes with the earliest deadlines first). This construction requires that the deadline for each process, if present, be defined. In our BE algorithm, we define the deadline as being the latest time at which the process can be completed to produce a specified deadline threshold percentage ν of its maximum value.

7.3.1.3. Minimum Value

If a process has a rising value function (i.e., contains processes drawn from sets 13-16 in figure 7-1), the algorithm must determine how long the process completion should be delayed to achieve a high value for its completion. The computation, as described in Chapter 5, considers both the likelihood of an incoming competing process being of higher value and the probability that the system is overloaded to determine how early to schedule such a process,

but it will always attempt to schedule it with at least the percentage of the process' maximum value specified by the minimum value, λ.

7.3.1.4. Pre-execution Limit

As this algorithm was developed, it became clear that to make a best effort to achieve a high value for a process with a rising value, it would be highly profitable to "pre-execute" a process (i.e., even if a process should not complete until far in the future, its processing could start early), to be halted at an appropriate time preventing its completion until it is likely to be completed with a high value. In an actual system, this value would probably be specified individually for each process by its designer based on an understanding of its function and its probable minimum processing time. For the purpose of our experimentation, however, we define the portion of a process' execution time to be left as a multiplier T_λ of the process' standard deviation, thus defining a pre-execution limit. Thus, a smaller pre-execution limit implies a higher probability that a process might be completed when it is "pre-executed" rather than at a later time of high value.

7.3.2. Tests Run to Evaluate Each Parameter

To determine the sensitivity of the scheduling algorithm to the values of each of these parameters, a set of simulations has been executed against a heavy load, measuring the total value generated as a proportion of the computed upper bound for that value in each run. The simulated environment for all of these runs consisted of a single processor running for 40 seconds of simulated real-time, executing a set of between 20 and 40 processes with normally distributed execution times. All processes for these runs were drawn from an identical set of processes with about 10% periodic, and 90% aperiodic, with the aperiodic processes exhibiting an exponentially distributed 10 second mean arrival period. The value functions for these runs were evenly mixed from the four types described in Figure 7-1 for the runs evaluating θ and ν, since these parameters are not dependent on any particular type of value function, while the value functions for evaluating λ and T_λ uniformly contained quadratically rising values and quadratically decreasing decay values, since these parameters have effect only for processes with rising value functions.

7.3.2.1. Overload Probability Threshold

A number of runs were made varying the overload probability threshold θ from 0 to 0.9, (i.e., any overload probability higher than 0 or 0.9 would result in dropping one or more processes from the run queue prior to scheduling). The value of 0 is, of course, extremely low, and means that any measurable overload, regardless of how slight, would result in dropping processes of low value density, thus rendering the schedule quite similar to a simple value density schedule (see Chapter 5). On the other hand, the value of 0.9 means that an overload must be quite pronounced before processes will be dropped, which will increase the likelihood that processes unlikely to be completed will be scheduled to run. The resulting schedule for this case is similar to a simple deadline schedule.

We would not expect that the BE scheduler would demonstrate a very high sensitivity to this parameter, since we observe that the overload probability computation most frequently produces values of either 0 or 1, except when the load and processing resources are quite closely matched. Based on the results described in Section 7.4 with our simple value density (VD) scheduler vs. the deadline (D) scheduler, we would expect that a low value of θ will produce somewhat better total values. Indeed, the runs we have made show exactly this effect (see Figure 7-2). The results illustrated in this figure show this for runs with loads of 20, 28, and 40 processes (producing average processor loads of 129%, 178%, and 243% respectively).

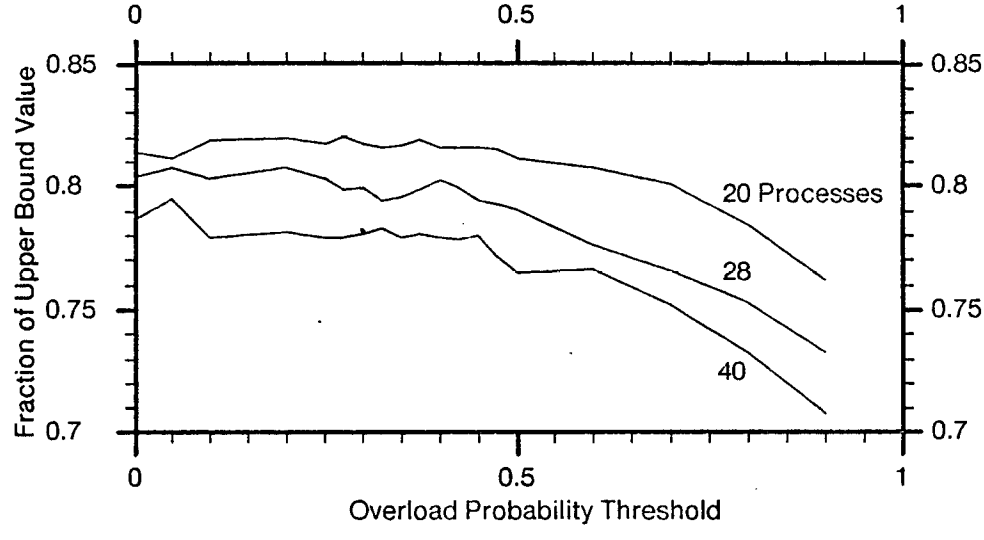

Figure 7-2: Overload Probability Threshold Evaluation Results

From Figure 7-2, we observe that the entire range of value variation from one extreme to the other is within about 7% of the value upper bound for all loads tested, confirming our expectation that the algorithm is not extremely sensitive to this parameter. Further, we note that the value drops off toward the high values of θ, thus we conclude that this parameter should be set at a fairly low value (< 0.5) for the remainder of these experiments. We therefore have run the remaining experiments with $\theta = 0.2$, meaning that processes will not be removed due to overload until the overload probability is higher than 20%.

7.3.2.2. Deadline Threshold

The set of runs made to evaluate the deadline threshold ν were similar to those made for the overload probability threshold, varying ν over the range 0.1 through 0.98, thus defining a deadline as the latest time at which a process can be terminated with 10% of its peak value through 98% of peak value. This value is principally used by the BE scheduler to determine the initial deadline schedule from which processes will be removed in event of overload. The effect of lowering this value will depend on the shape of the value function. Clearly, for a step function, this value will have no effect as long as it stays above the lower value of the step. For the exponential decay, it will have a significant effect only when it is quite low. For the quadratic decay, however, a lower value of this parameter will prolong the interval in which the process can be scheduled, thus slightly increasing the likelihood that such processes will finish late. Since our runs used a uniform mixture of these types of value functions, it would not be expected that varying this parameter will have a large effect, and this was confirmed in the results shown in Figure 7-3.

As with Figure 7-2, three sets of runs were made with loads of 20, 28, and 40 processes, and the fraction of each run's upper bound value achieved was plotted against the deadline threshold. It will be noted that for all three load levels, the general effect is the same; increasing the threshold increases the resulting total value achieved until a threshold of between 92% and 98% is used. This means that for the type of value functions used in our experiments, in which the highest value possible for the process occurs at the critical time, the deadline definition will be nearly the same as the critical time.

It is also important to note that the effect of changing this value is small as long as it remains above about 50%; the resulting change in value is never larger than about 3% of the upper bound value throughout this interval. As a result, we will set the value of the deadline threshold to 90% of the peak value for the experiments described following this section.

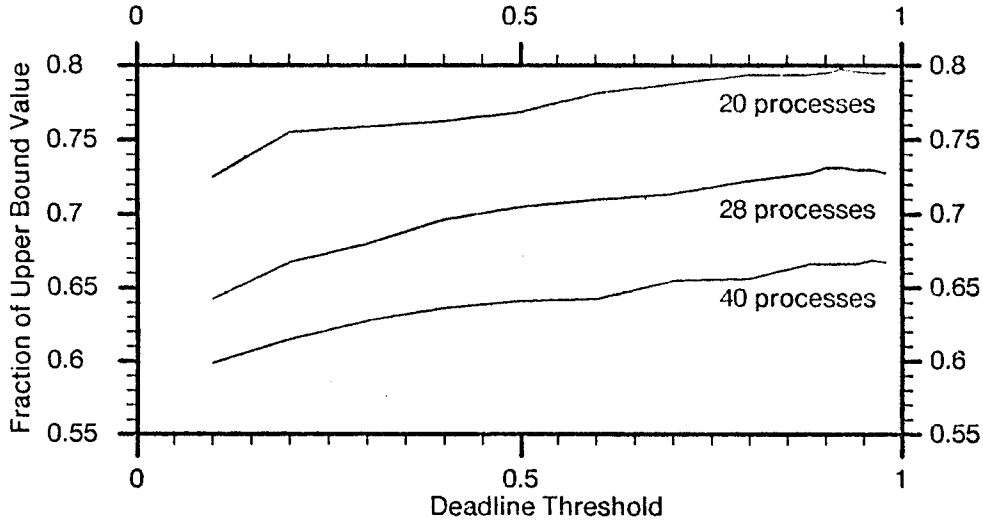

Figure 7-3: Deadline Threshold Evaluation Results

7.3.2.3. Minimum Value

The experimentation to determine λ, the minimum value acceptable for processes with rising value functions has been carried out using the same set of processes as those used for the other parameters except that all value functions have a quadratically rising value prior to their critical times. Even using these value functions, we do not expect this value to be critical to the overall performance of the algorithm, since it will significantly influence only the scheduling of low value processes in the presence of significant overloads. We have therefore defined two sets of loads with which to make this evaluation; one set was run with 20 processes and the other with 40. The results, plotting the fraction of the upper bound value achieved against λ, are shown in Figure 7-4.

As expected, this value shows no great effect; the overall excursion is no greater than 1% of the upper bound value. In fact, the effect of changing this parameter is so small that it is completely insignificant; in our experimentation following this section, we arbitrarily set λ to 20% of the maximum value of the process.

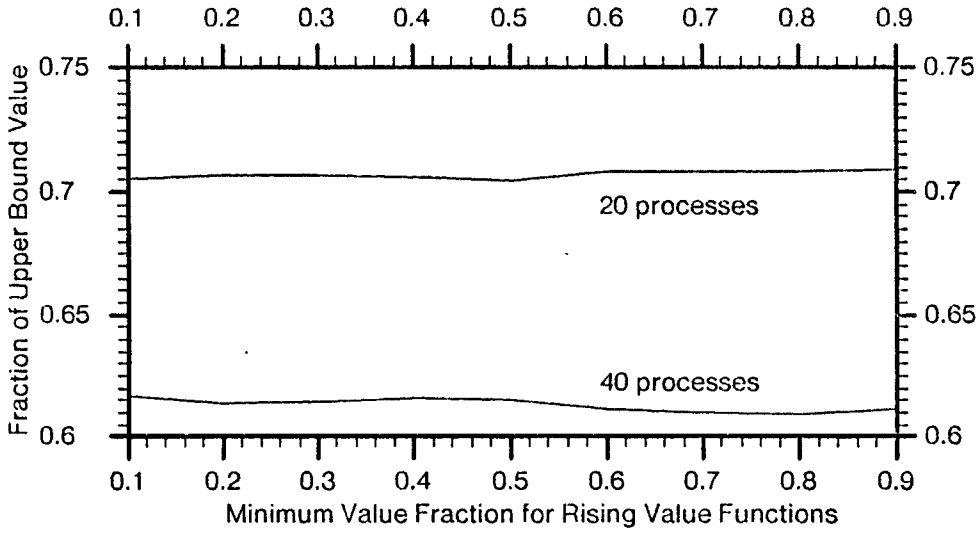

Figure 7-4: Minimum Value Evaluation Results

7.3.2.4. Pre-execution Limit

Unlike the other parameters evaluated in this section, the pre-execution limit T_λ represents a significant user policy decision, in that it determines the acceptable level of risk that a process will be completed prior to its correct time interval. In a real system, there would probably be processes for which this risk is relatively unimportant, and others which are more critical, so this policy decision would be specified individually for each process in terms of the process' time constraints. However, in this experimentation we have defined this value as a function of the variance observed for each process so that its effect on the overall value achievable for a given set of processes could be measured.

In these runs, we use two load levels, 20 and 40 processes respectively, varying T_λ from 1.0 through 4.0 times the assumed execution time standard deviation, thus allowing the algorithm to pre-execute a process for only its mean execution time minus $T_\lambda \sigma$ before interrupting it to wait for its proper execution interval to resume. The value functions used in this set of runs are all quadratically rising in value prior to the critical time. The results of these runs, plotted as a fraction of the upper bound value achieved vs. the pre-execution limit T_λ are shown in Figure 7-5.

We note from Figure 7-5 that T_λ has a much more pronounced effect in a lighter load than in the presence of a heavy load. This effect is the result of the fact that a process will be

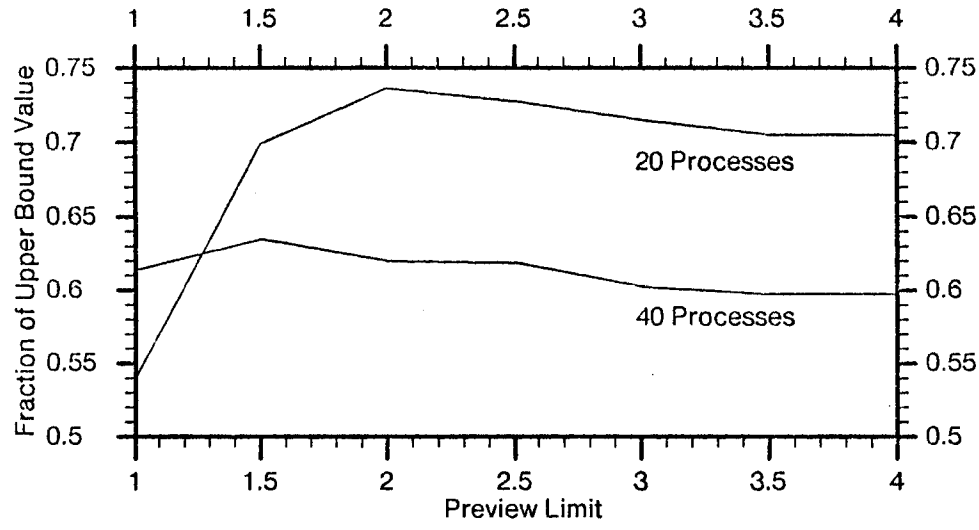

Figure 7-5: Pre-execution Limit Evaluation Results

pre-executed only if time is available, which occurs less frequently in the presence of a heavy load. For either the heavy or light load, the resulting total value seems to be best when T_λ is set to between 1.5σ and 2.0σ, which represents a compromise between never pre-executing a process and allowing it to inadvertently complete prematurely. Thus, we set T_λ to 2.0 in the remainder of our experimentation.

7.4. Results of Evaluation Against Selected Loads

One of the primary motivations for this research is to produce a real-time scheduler which will produce a consistently high value from scheduling a set of processes with time constraints, especially in the presence of transient or persistent overloads. There are a number of existing scheduling algorithms which are currently used or have been proposed to be used in real-time systems, and it is important to compare the performance of our BE scheduler with their performance. Because of the complex environment in which such systems operate, the most effective method of comparing them is by the use of the system simulator described in Chapter 6. There are a number of critical situations which a scheduler must handle to produce a consistent overall value, and it is the goal of this set of experiments to subject these schedulers to these situations, comparing their resulting total value.

The scheduling algorithms to be tested and the reasons for their inclusion are:

- *BE* -- Our best-effort scheduler described in Chapter 5.

- *VD* -- A pure value density scheduler, which always schedules the process with the highest value density. This scheduler has been included to illustrate the significance of value density to the scheduling decision.

- *SPT* -- The shortest processing time scheduler which schedules the process with the shortest expected execution time. This scheduler is included because, if deterministic execution time were used, it is provably optimal against several performance measures in a single processor (e.g., Minimum mean flow time, and minimum mean lateness [Baker 74]). In this application, it uses the expected remaining execution time each time the algorithm is invoked (when any process is requested or terminates). The SPT algorithm is widely used in such applications as job shop scheduling, but has not been used in real-time scheduling.

- *FV* -- A fixed priority scheduler which schedules the process in the run queue with the highest priority. The priorities for this scheduler are selected at the initialization of the simulation when the processes are defined, and are based on the maximum value attainable by that process. Once the processes are defined, the priority remains fixed for that process thereafter. The maximum value represents an estimate of the "importance" of the process, which is then used to schedule it. This is, in the author's experience, the most frequently used scheduler in actual real-time systems.

- *FD* -- The fixed priority scheduler, which schedules the process in the run queue with the highest priority. The priorities for this scheduler are selected at the initialization of the simulation when the processes are defined, and are based on the tightness of the time interval between the process' request time and its critical time, with smaller intervals having higher priority. Once the processes are defined, the priority remains fixed for that process thereafter. Although fixed priority schedulers are widely used in real-time systems, the priorities are seldom chosen by the tightness of their deadlines, but rather by their presumed "importance". When a system is under test, however, it is not uncommon to find that such a system exhibits difficulty meeting its time constraints, and these time constraints are then frequently used to adjust the fixed priorities in the hope that the problems will be corrected.

- *D* -- The deadline scheduler which always schedules the process with the earliest deadline. The deadline scheduler is included due to its known ability to meet all deadlines in a non-overloaded single processor, although it is not generally used in real-time schedulers.

- *SL* -- The slack time scheduler which schedules the process with the smallest slack time (or laxity). In our simulations, the slack time is defined as the time to deadline minus the expected execution time. Similarly with *D*, this algorithm is optimal in its ability to meet deadlines in non-overloaded single processors with deterministic execution times, but has other undesirable side effects. As a variant of the *D* scheduler, it is included for comparison with the other algorithms.

- *R* -- The random scheduler which assigns a random priority to each process when it is requested. The scheduler executes the process in the run queue with the

highest priority. Thus, each time a process is requested it will have a different priority. This scheduler is included to provide a measure of the effect of removing all knowledge the process set from the scheduling decision. While this algorithm, of course, is not used in existing real-time systems, it is effectively the scheduling algorithm used in certain bus contention protocols, such as the Ethernet.

- *FIFO* -- The first-in-first-out scheduler which schedules the process which entered the run queue earliest. While there are no scheduling performance measures for which this algorithm performs optimally, it is used to promote "fairness" in scheduling message handling in many real-time systems.

In the experiments described here, all of these algorithms are used to schedule a set of real-time processes under varying load conditions, applying execution time distribution knowledge of varying accuracy, and with varying numbers of processors, evaluating them with respect to the total value achieved over the simulation interval.

Before the simulations could be started, however, we performed a small set of experiments to determine the length of the simulation period to be used. The simulation must extend long enough to provide confidence in the resulting value, but must be short enough to complete the set of simulations in a reasonable time. We note that the value achieved in the very early portions of the simulation are non-representative of the algorithm performance, since it takes up to one or two seconds for enough processes to be requested to represent a "normal" load. Similarly, the value achieved at the end may not be fully representative, since the requests stop at a predetermined time, but the simulation continues until the queue is empty (either by process completion or abortion). In an effort to determine a sufficient simulation length, a set of identical runs were made of different lengths, varying from an elapsed time of 5 seconds to 60 seconds in 5 second increments. Figure 7-6 shows the results of these runs, using 40 processes and 20 processes in two series, plotting the fraction of the upper bound value achieved against the run time in seconds.

From these runs, we decided to somewhat arbitrarily pick a simulation time of 30 seconds, providing for enough process requests to overshadow the early and late value accumulation discrepancies, noting that the experiments indicate that there is no apparent systematic discrepancy due to the use of a short or long run time in the range covered. The few percent variation visible in Figure 7-6 are apparently due primarily to the normal variation of the average load.

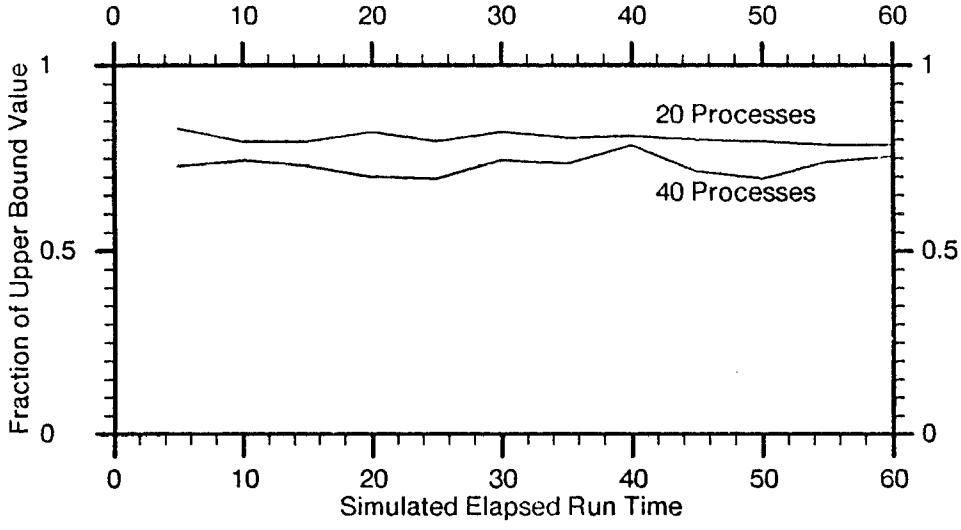

Figure 7-6: Run Time Evaluation.

7.4.1. Load Variation Tests

The first set of evaluation simulations was designed to compare the performance of a real-time system under varying load conditions using our time-driven scheduler and the other scheduling algorithms previously described. These results illustrate the fundamental result which constituted our principal scheduling objective; we expected to produce a real-time scheduling algorithm which could perform effectively and consistently in the presence of either normal loads or overloads. We require that this consistency be evident, not only under wide variations of process load, but also in the presence of an arbitrary set of value functions describing varying time constraints for the processes involved.

In these experiments, we first make a straightforward set of executions using a set of from 4 to 40 processes, representing loads ranging from about 15% to about 245%, drawing all the processes from the set identified *load1n* (set 1 in Figure 7-1). We used only a single processor, and repeated each of the runs 10 times, each time with a different request set, but with each request set processed identically by all the algorithms tested, averaging the results over the 10 runs. It is important to understand that the load averages given are averaged over the entire run; even the lowest load level contains a few periods of transient overload within its execution interval.

We repeated this entire set of runs four more times, drawing all processes from *load2n*,

load3n, *load4n* (see Figure 7-1), and from a uniform mixture of all four of these sets, respectively. For all five of these experiments, we plotted the fraction of the upper bound value produced by each simulation against the average percent load, and we present this information in Figures 7-7, 7-8, 7-9, 7-10, and 7-11, respectively.

Each curve plotted in these figures consists of a line connecting points representing the mean load factor and mean value fraction for the set of 10 runs made with a particular process load. Actually, of course, each of these points should have been plotted as a small ellipse, representing the uncertainty of the mean of these samples. As a part of the information produced by the simulator during these experiments is the 2σ confidence intervals for each of these points; we elected not to show them on the figures due to the excessive complexity that would result. The confidence ellipses were quite constant, however, with a width of about 8%-9% of process load, and a height of about 5% of the total value. These intervals are sufficiently small that the trends observable on the figures are not adversely affected, although exact conclusions about the relative placement of points which are very close together should not be drawn from the graphs. For example, on Figure 7-7, the relationship of the BE and VD algorithms at a load of about 170% cannot be determined from the graph; the mean load was measured as 225% with a 2σ confidence interval of $\pm 7.7\div$, the BE value fraction was measured as .713 with a 2σ confidence interval of $\pm .032$, and the VD value fraction was measured as .720 with a 2σ confidence interval of $\pm .029$. These confidence intervals confirm the apparent consistency of total value achievable by the use of value functions for real-time scheduling.

The first three sets of runs, shown in Figures 7-7, 7-8, and 7-9, show considerable similarity in overall shape, indicating that all of the schedulers performed similarly on the different value functions represented (we note that each of these value functions has a constant value prior to its critical times, and differ only in the nature of their value loss after the critical time). Each scheduler shows that its performance is consistently good with these three value functions at low load (15%), as expected, since the scheduling decisions are usually trivial at these loads. Not surprisingly, the random scheduler (R) consistently performs among the poorest in each of these runs, due to its use of no process or system knowledge in its decision making, but at such low load levels, even random scheduling performance is acceptable.

As the load increases, however, and notably well before 100% average load is reached, the schedulers diverge widely in performance, with the worst performance (usually FIFO, SL, or

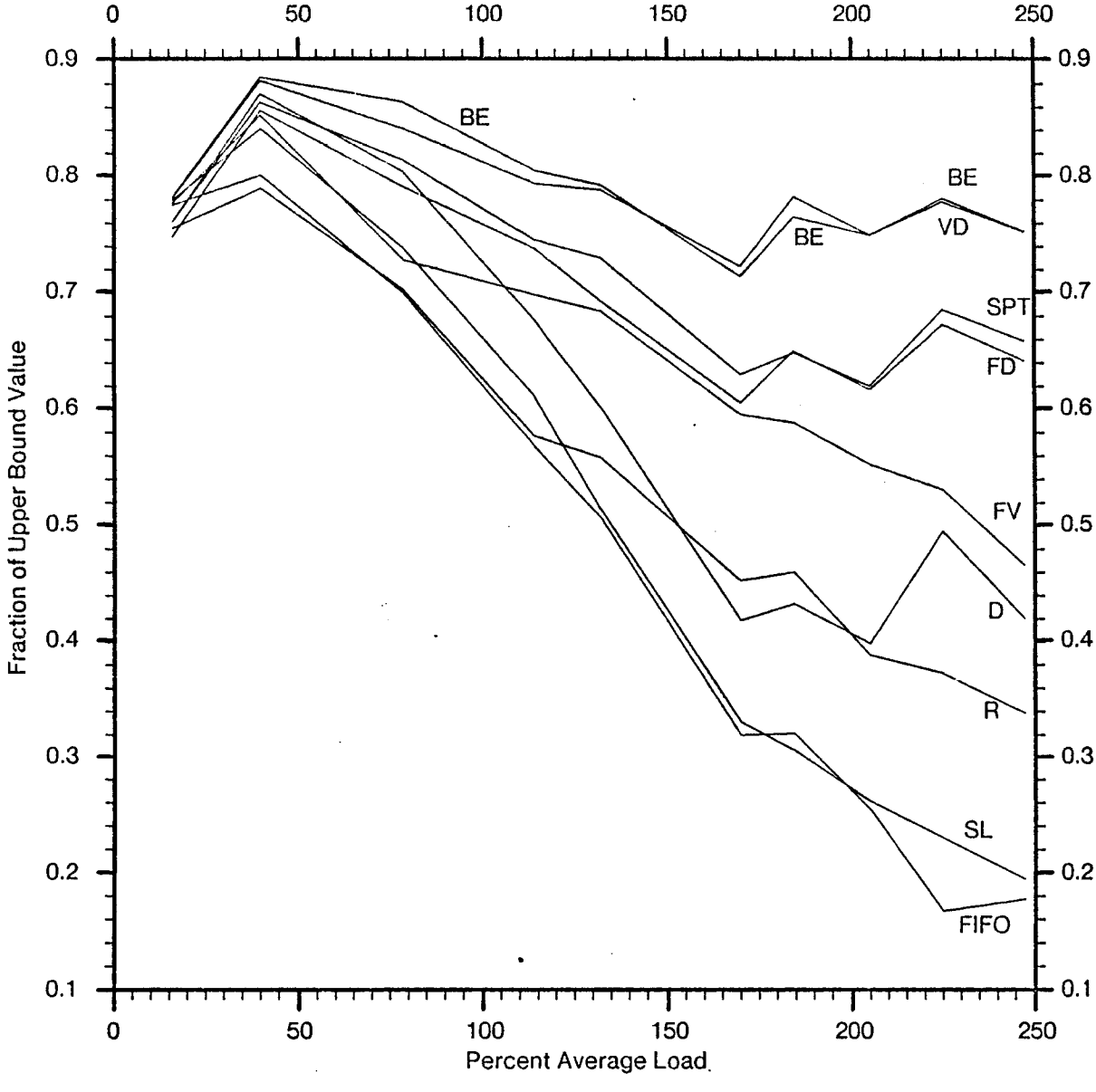

Figure 7-7: Scheduling Algorithm Evaluation with Varying Load Levels (load1n)

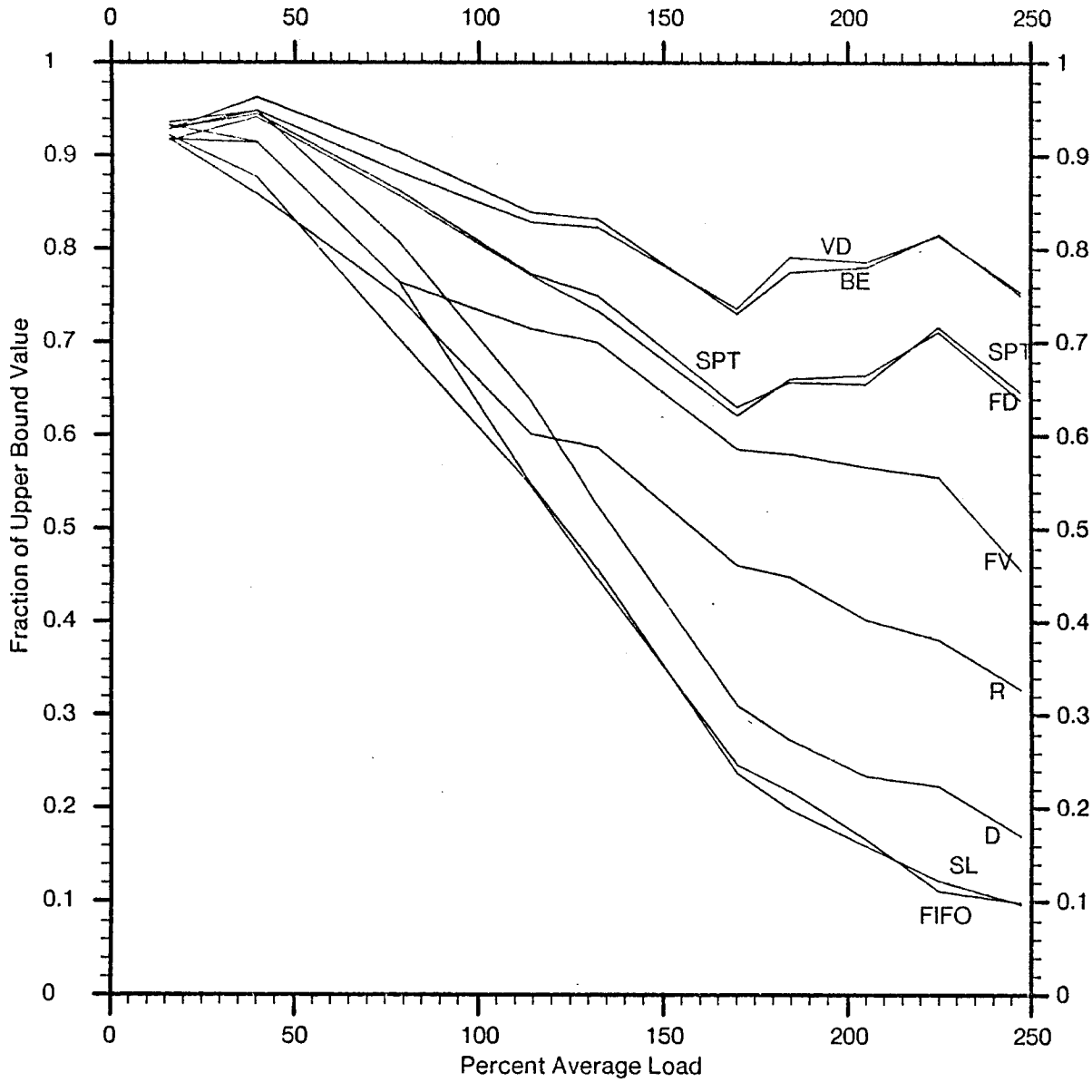

Figure 7-8: Scheduling Algorithm Evaluation with Varying Load Levels (load2n)

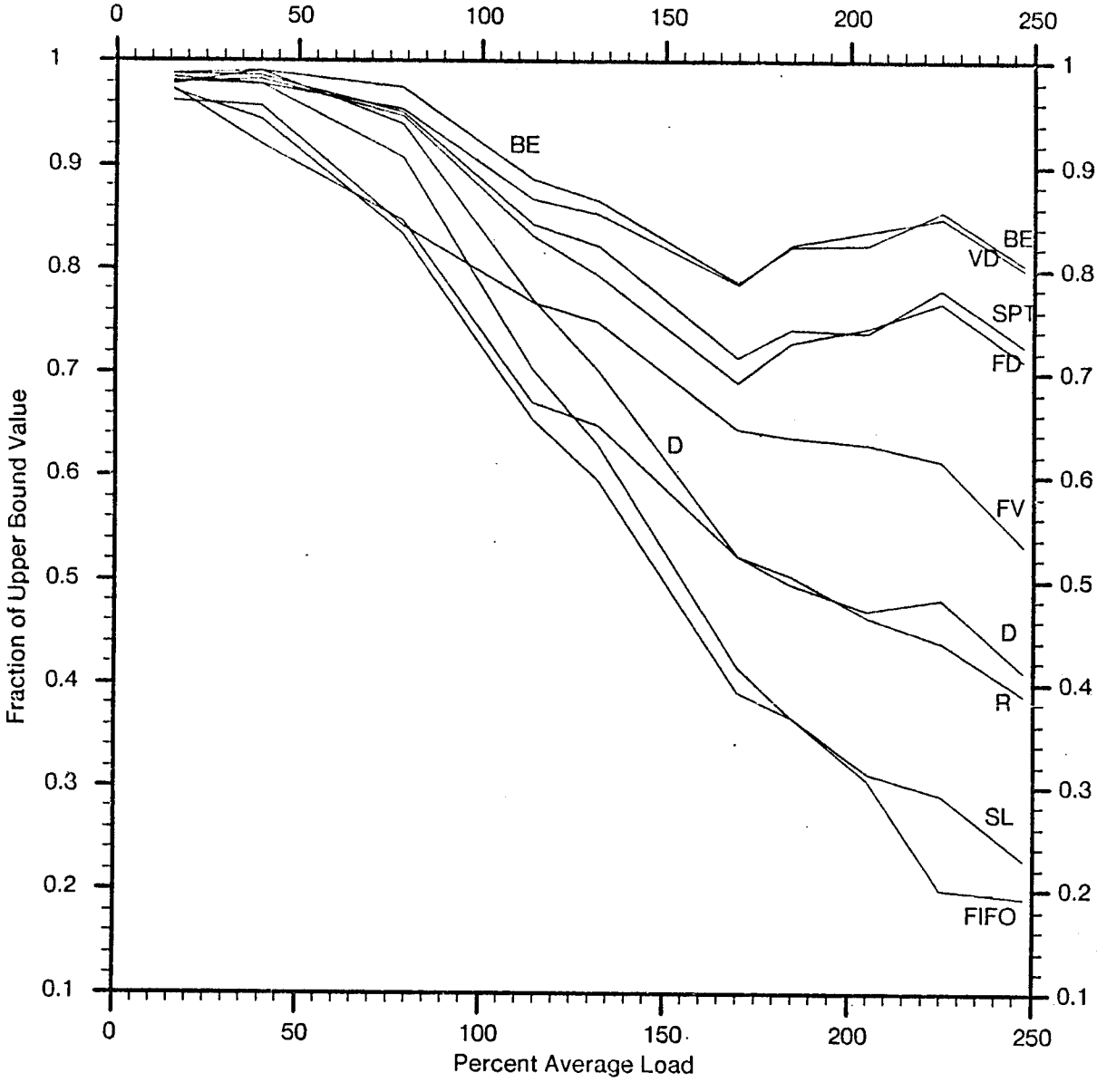

Figure 7-9: Scheduling Algorithm Evaluation with Varying Load Levels (load3n)

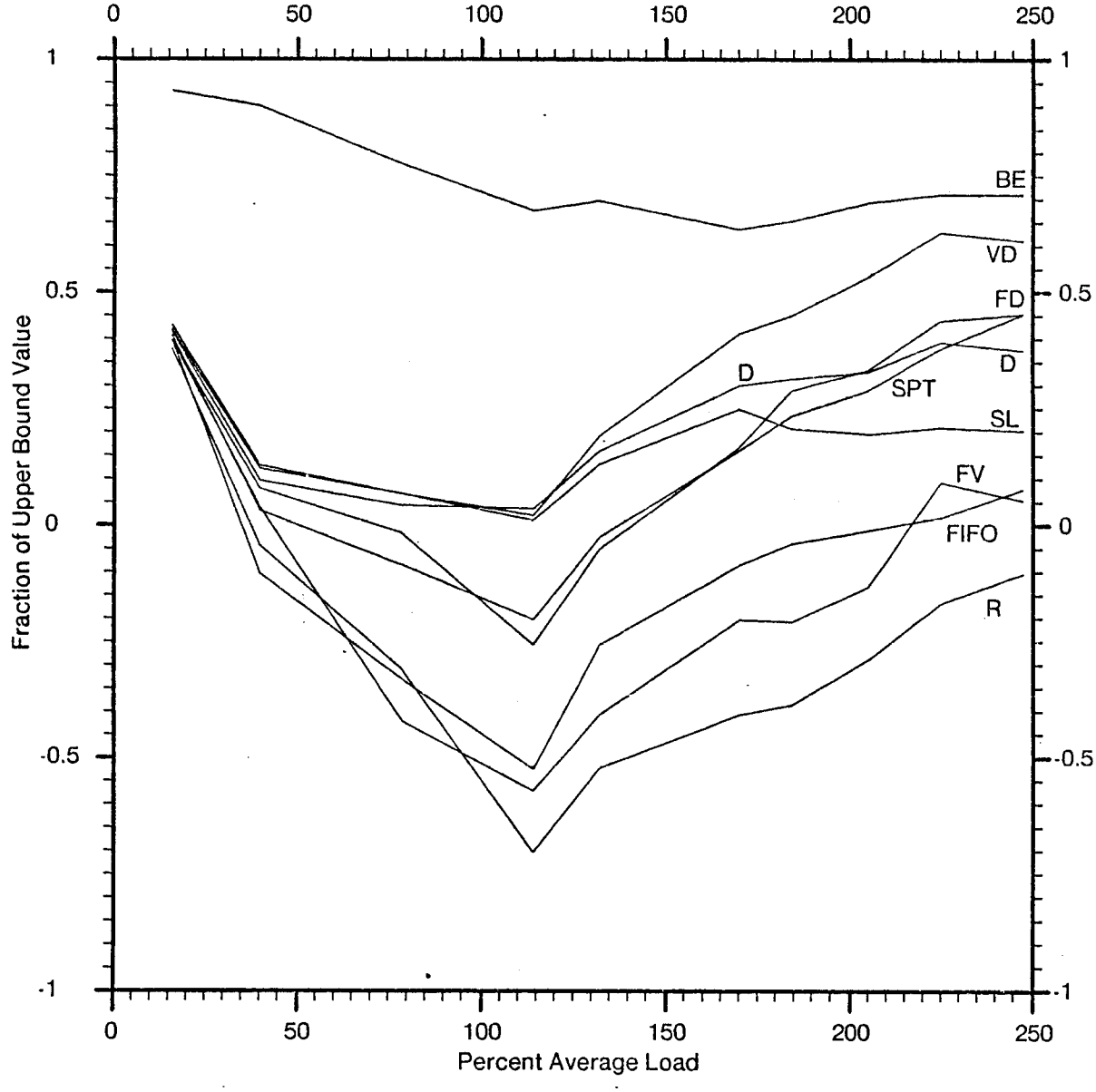

Figure 7-10: Scheduling Algorithm Evaluation with Varying Load Levels (load4n)

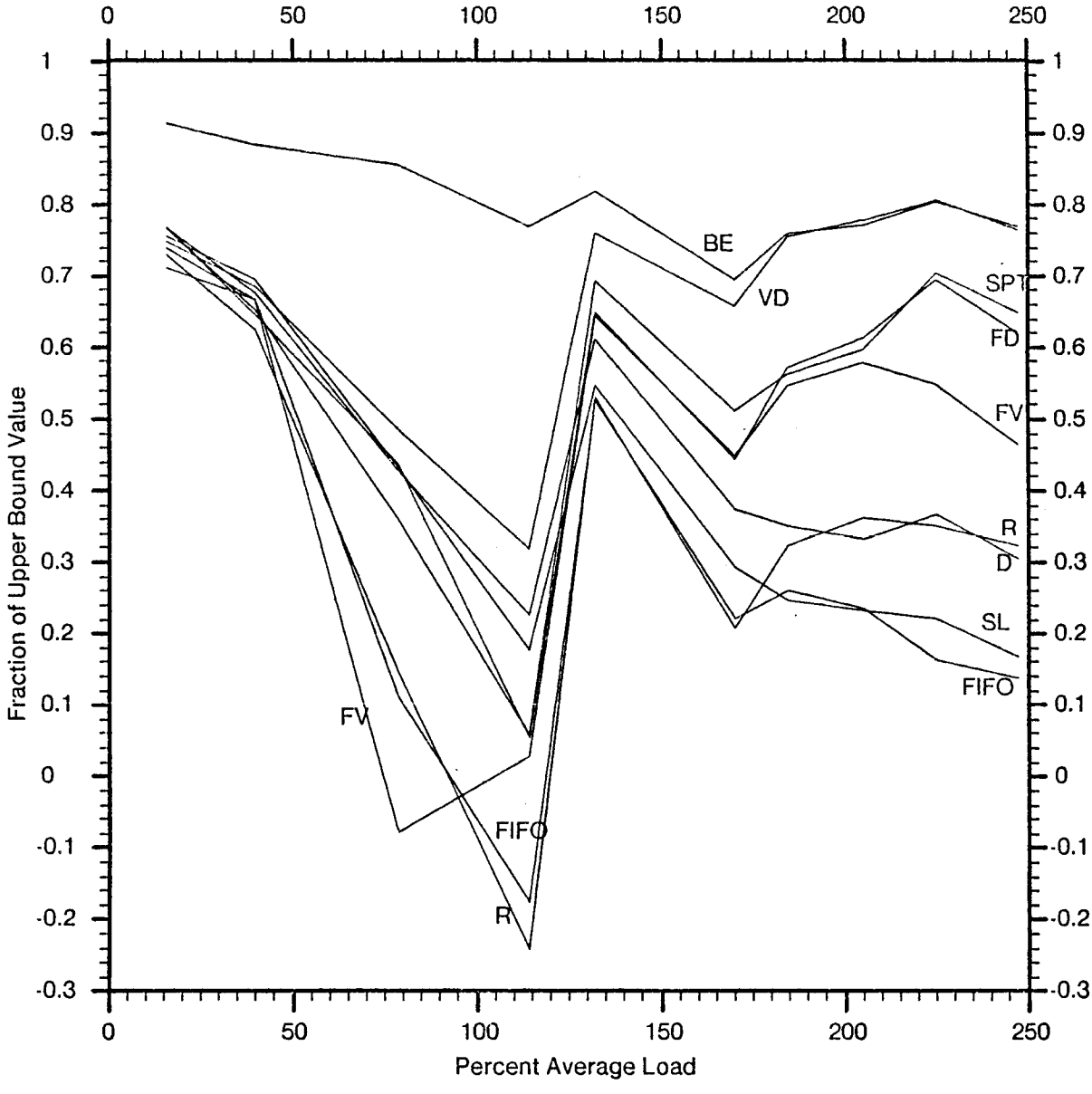

Figure 7-11: Scheduling Algorithm Evaluation with Varying Load Levels (Mixed Value Functions)

R) differing from the best by over 20% of the upper bound value by the time the system is 100% loaded. As the load increases to almost 250% average load, the divergence increases to 60 -70% of the upper bound value, with the FIFO and slack time (SL) consistently showing the poorest total value. As discussed later in this chapter, a low value signifies that when a scheduling decision must be made, *the wrong process is selected for execution*, preventing processes with high importance from being completed within their time constraints. Thus, the schedulers at the low end of this spread would violate the fundamental design goal of a real-time system in that the processor will frequently be assigned to low-importance processes rather than high-importance processes when contention exists. This represents one of the fundamental reasons for the unpredictability of most existing real-time systems in the event of overload.

The best performing schedulers in these runs is consistently our best-effort time-driven scheduler (BE) and the pure value density scheduler (VD). These schedulers take both the value function and the expected execution time into account, although VD ignores rising values (not present in Figures 7-7, 7-8, and 7-9) and processes the most "value-effective" processes first, possibly aborting lower importance processes which could have been completed. This effect becomes less important as the load increases, since it becomes less likely that such low importance processes could have completed anyway, accounting for its near parity in these runs with the BE scheduler at high load levels. On the other hand, the BE scheduler, attempting to process low importance processes if it believes that both high and low importance processes can be completed, occasionally finds that it must abort a process when a very important process unexpectedly arrives. This effect, also, is less important at high load levels, but generally results in an advantage for BE at low to medium load levels.

We note the perhaps surprising results of the shortest processing time (SPT) scheduler, since it is not usually considered an effective real-time scheduling algorithm. This is less surprising when one observes that SPT is provably optimal in minimizing mean lateness (in a single processor with known execution times) [Baker 74], and does not share with the deadline (D) and slack time (SL) schedulers their propensity for favoring processes which have no hope of meeting their deadlines over processes which can still meet their deadlines in an overload.

The two fixed priority schedulers, (FD) and (FV), representing scheduling algorithms which are frequently used in actual real-time systems performed acceptably at low load levels, but

dropped significantly at higher load levels. The FV scheduler, particularly, performed poorly, even though it correctly takes process importance into account. This result will, no doubt, prove surprising among many practitioners in real-time operating systems design, among whom this author is included, who have implicitly assumed that the most important processes should usually receive the highest priority in real-time systems.

Research on the use of the deadline scheduler (D) has predicted that it should perform well on underloaded systems, and this seems to be confirmed in this experimentation, but its inconsistency as described above rules it out in an overload situation, as it drops to the performance of the random (R) scheduler or below at high overload conditions.

Figure 7-10 shows quite a different side of the schedulers' performance. Here, all value functions are rising quadratically prior to the critical times, and dropping quadratically following them, showing the effects of ignoring knowledge of early time constraints on scheduling performance. Here, the BE algorithm alone shows a highly consistent value performance; the other algorithms perform poorly at low load levels, improving somewhat at higher loads since the processes will naturally be executed later, on average, at high load levels. In existing real-time systems, such time constraints are typically not handled at all by the operating system, but are handled by the application with the attendant weakness that the system state is not considered in the scheduling decision process. This is one of the significant strengths of the BE algorithm; it performs consistently over every value function and load tested. It is interesting that in this case the performance of FV and R decrease significantly relative to the other algorithms.

Figure 7-11 completes this picture by illustrating the effect of combining these value functions in a single process load. This load is much more realistic, and the consistently high value achieved by the BE algorithm in these runs is exactly the result for which we were striving.

7.4.2. Effect of Run-Time Knowledge

Knowledge of the execution time of each process is used in scheduling decisions by the BE, VD, SPT, and SL algorithms; an important question which we need to investigate is their sensitivity to this knowledge. In particular, we describe here four sets of experiments designed to show this sensitivity. These experiments are intended to demonstrate the effects of changing:

1. Execution time variance.

2. Execution time means.

3. Tight vs. loose time constraints.

4. Execution time distribution (in which the scheduler knows the distribution).

5. Execution time distribution (in which the scheduler does not know the distribution).

7.4.2.1. Execution Time Variance

For this set of experiments, a load consisting of 24 processes drawn equally from load1n, load2n, load3n, and load4n as defined in Figure 7-1 was generated, but the standard deviation of the execution time was varied in the range of 5% to 100% of the mean execution time for each process. The execution times were normally distributed for these experiments. Each such load was executed 10 times with independently drawn request times, and each such request time sequence was identically processed by all nine of the scheduling algorithms defined above.

The purpose of these tests was to see the effect of variations in the accuracy of the execution time knowledge on the quality of the resulting scheduling decisions. Our expectation would be that the scheduler should be able to make significantly better decisions in the presence of more predictable execution times, and that this effect would be most pronounced for those schedulers which explicitly utilize the execution time for decision making. A graph of the results of these experiments is shown in Figure 7-12, in which the fraction of upper bound value generated is plotted against the execution time standard deviation expressed as a fraction of the mean run time for each process.

As expected, we observe an almost linear decrease in value as the standard deviation increases. We note that this effect is more pronounced for some schedulers than others; the drop for BE seems to be less than that for VD and SPT, for example, while the drop is significantly less pronounced for D, SL, and FIFO. Expecting that the presence of processes with rising value functions might be affecting these results, we reran the experiments using only the step value functions defined in load1n from Figure 7-1, to isolate the effect of including processes with rising value functions. We expected that including load4n value functions would have a significant effect because the reduced certainty of the run-time would make it more difficult to predict the termination time of the process, making it more likely that

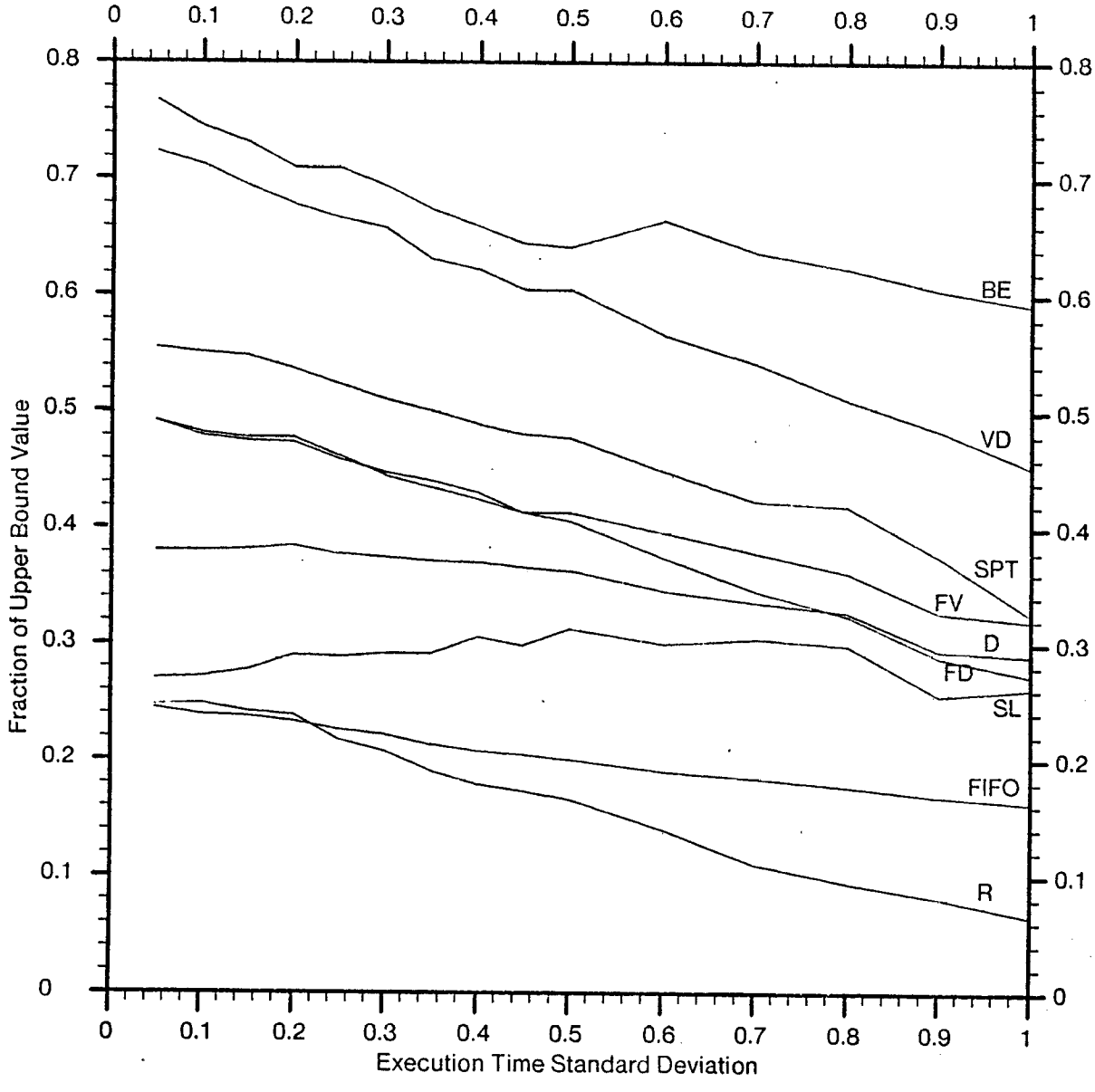

Figure 7-12: Scheduling Algorithm Evaluation with Varying Standard Deviation (Various Value Functions)

the relatively narrow peak value could not be achieved. These runs are shown in Figure 7-13, with the same scales used in Figure 7-12.

As shown in previous experiments, we find that exclusion of rising value functions causes the BE and VD schedulers to perform quite similarly, but we also note that the schedulers which do not use the expected execution time explicitly in their decision making (i.e., FD, FV, R, D, FIFO) show little effect due to the change of execution time standard deviation. From this comparison, we can see that the presence of processes with rising value functions has apparently caused the sensitivity to uncertainty in execution time shown by most of the schedulers in Figure 7-12 to be increased.

Indeed, the slack time (SL) scheduler shows an improvement in performance as the uncertainty increases; investigation into its decisions indicates that this was caused by significantly reduced preemptions. The SL scheduler performance typically suffers with respect to the deadline (D) scheduler because it tends to preempt running processes with waiting processes too frequently. This occurs because the slack time for a process decreases when it is *not* executing, but not when it *is* executing, and SL schedules the process with the smallest slack time. This effect decreases with increasing execution time uncertainty since the computation of expected remaining execution time (See Chapter 6) will produce a smaller change for a running process when the distribution has a larger standard deviation.

7.4.2.2. Execution Time Means

Given that we understand the sensitivity of the BE scheduler to the execution time variance, we would like to determine its sensitivity to our knowledge of the execution time itself. In the preceding simulations, the BE scheduler knew the actual mean execution time for the population from which the process execution times were drawn, allowing the scheduler to make well-founded decisions. In this section, we report a set of experiments in which the BE scheduler uses a biased knowledge of the execution time means to make the same decisions.

This set of experiments used the same five loads which were used in the load tests described in Section 7.4.1, consisting of the four loads described in Figure 7-1 and a uniform mixture of these four loads. In each case, 24 processes were used, resulting in an average load of about 160%. These executions were repeated with the assumed mean execution time for each process set to a multiple of the actual mean execution time (i.e., a multiple of 0.5

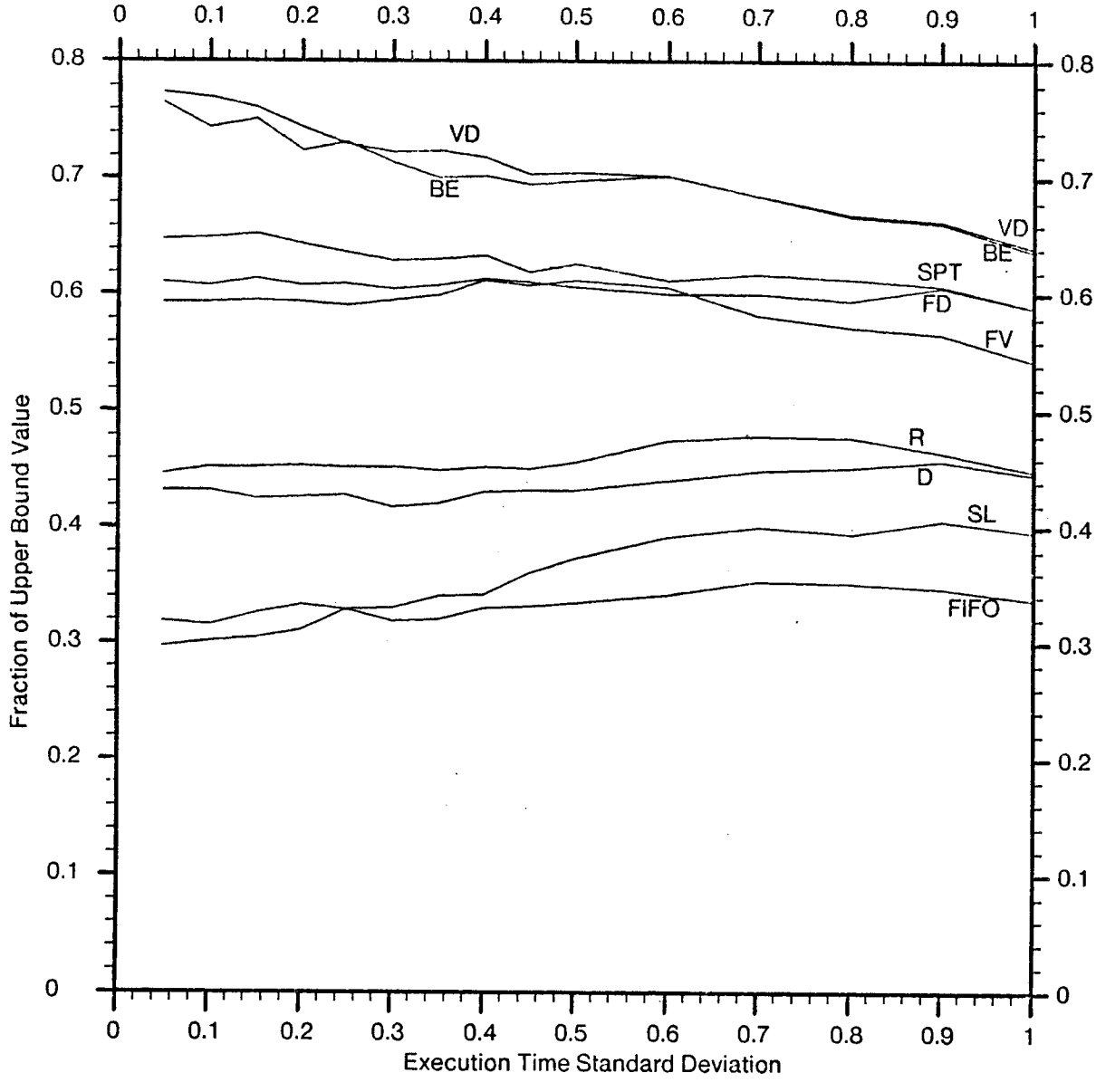

Figure 7-13: Scheduling Algorithm Evaluation with Varying Standard Deviation (Step Value Functions)

indicates that the assumed mean execution time is half that of the actual execution time for each process). As before, each set of processes was re-executed 10 times with different sets of requests used for each run, and the results were averaged, with the average total value returned by the scheduler plotted vs. the assumed mean execution time multiple in Figure 7-14.

We expect, of course, that the use of incorrect execution time means will degrade the performance of the scheduler, and this is indeed shown in Figure 7-14. Specifically, however, the sensitivity is dependent on the type of value functions present in the load. For loads with constant values prior to the critical times, we note that the knowledge of the mean for this load level is not particularly critical, as long as the assumed mean is at least as large as the actual mean. Thus, for loads with no rising value functions, we might conclude that it is not necessary to measure the mean execution time accurately to make good scheduling decisions. However, an investigation of the mechanism producing this effect shows that the use of progressively larger assumed mean execution times is causing the scheduler to increasingly detect an overflow condition, forcing it to remove load, and therefore produce a schedule which will be dominated by value density. As shown in our previous results, scheduling by the value density (exemplified by the VD algorithm) will generally produce a high total value for processes with these value functions, but, as discussed in Section 7.4.3, we will frequently unnecessarily drop processes whose time constraints could have been met.

In the case of loads containing processes with rising value functions, it is not surprising to see that the total value achievable is closely related to the knowledge of execution time means. In the load types used for this set of executions, the total value drops to about 65% of its peak level before leveling off due to the progressively frequent removal of load having removed all competing load, ensuring completion of the high value density processes. The width of the peak and the amount of the drop-off seems to be related to the shape of the value functions.

7.4.2.3. Tight vs. Loose Time Constraints

Clearly, a key performance indicator for a real-time system is its ability to meet timing constraints imposed by the application. In this subsection, we report on the execution of a set of experiments in which we test the ability of the BE scheduler, and the others with which we are comparing it, to meet varying time constraints. In this set of tests, we again use a load of 24 processes drawn from a uniform mixture of the four value functions described in Figure 7-1

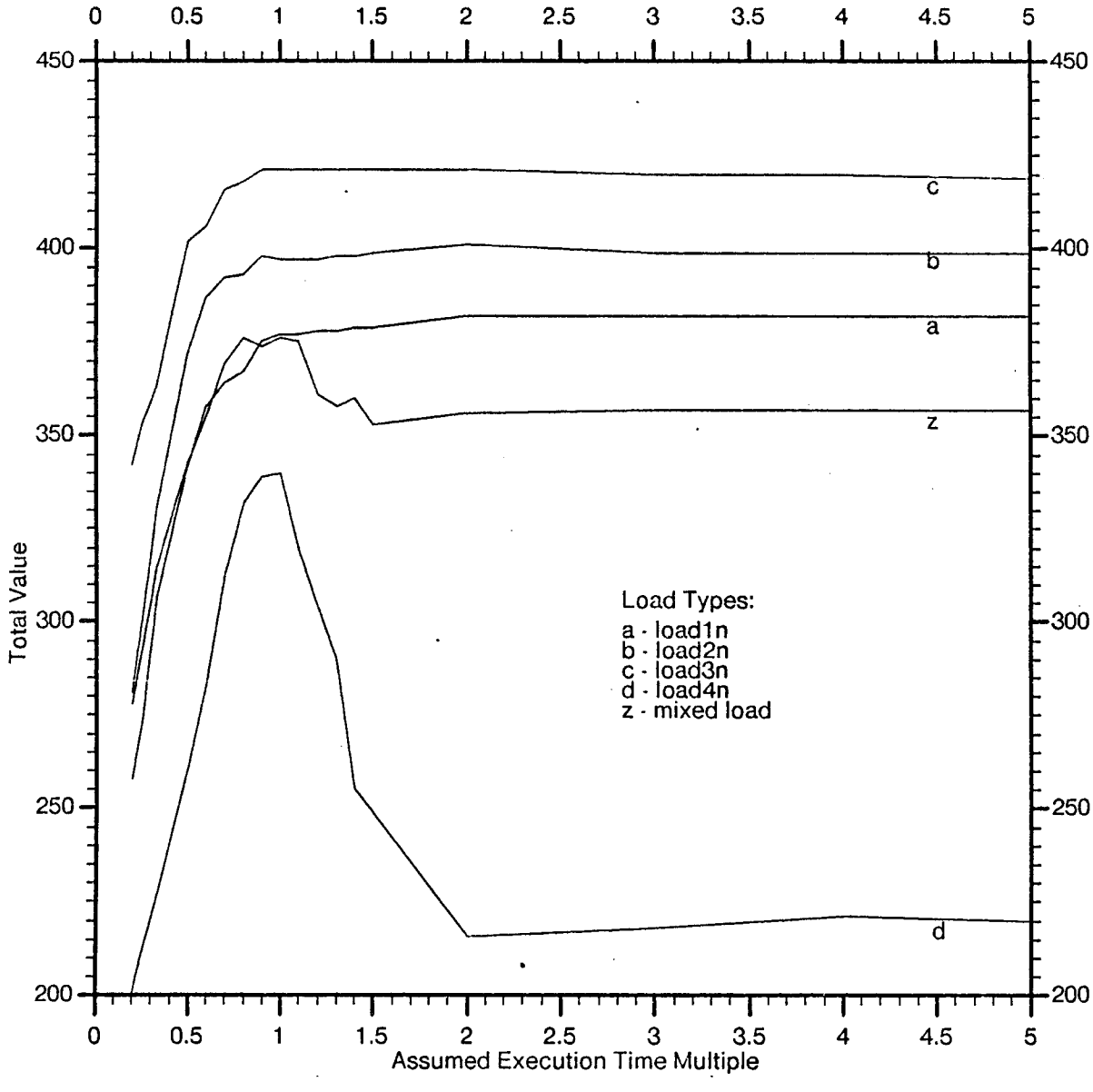

Figure 7-14: Scheduling Algorithm Evaluation with Varying Execution Time Mean Knowledge

(load1n, load2n, load3n, load4n), and vary the time constraints from one to the next. The time constraints are expressed as a percentage of the process mean execution time; a typical constraint specification consists of two numbers (e.g., 250 and 50). This specification controls the time interval from the request time to the critical time, so this example would mean that this interval would be drawn from a normally distributed set with a mean of 250% of the mean execution time, and a standard deviation of 50% of the mean execution time. For example, if a process has been created with a mean execution time of 100 ms., its request-critical time interval would be chosen randomly from a distribution with a mean of 250 ms., and a standard deviation of 50 ms. This value is determined once for each process during simulator initialization and remains fixed throughout the remainder of the execution for that process.

It is evident from this description that it could be possible for the request-critical time interval could be smaller than the mean execution time. In this event, it is quite likely the case that the process could never be schedulable to complete during this interval. Depending on the value function, this could mean that the process should always be aborted. It is important to remember, however, that the actual execution time for a given request could be more or less than the mean, so such a process might sometimes be schedulable, but not at other times.

In these experiments, we vary the constraint specification from 1.0 (i.e., 100%) to 5.0 (i.e., 500%), and present the results in Figure 7-15 as a graph of the fraction of the value upper bound versus the constraint specification. The standard deviation of the set from which the actual request-critical time interval is determined is 50% of the mean execution time in all cases.

Again in this graph, we note the consistency demonstrated by the BE scheduler under varying conditions of time constraints. We expect, of course, that the BE scheduler value should increase as the time constraints are loosened (i.e., to the right of the graph), but the BE scheduler demonstrates this without also showing a great sensitivity to this factor. In fact, close investigation of the actual runs involved show that it would have demonstrated much less sensitivity to this parameter (i.e., would have been considerably higher on the left) if it weren't for the large number of processes which were unschedulable due to their time constraints.

The interesting observation from Figure 7-15 is the poor performance of all the other

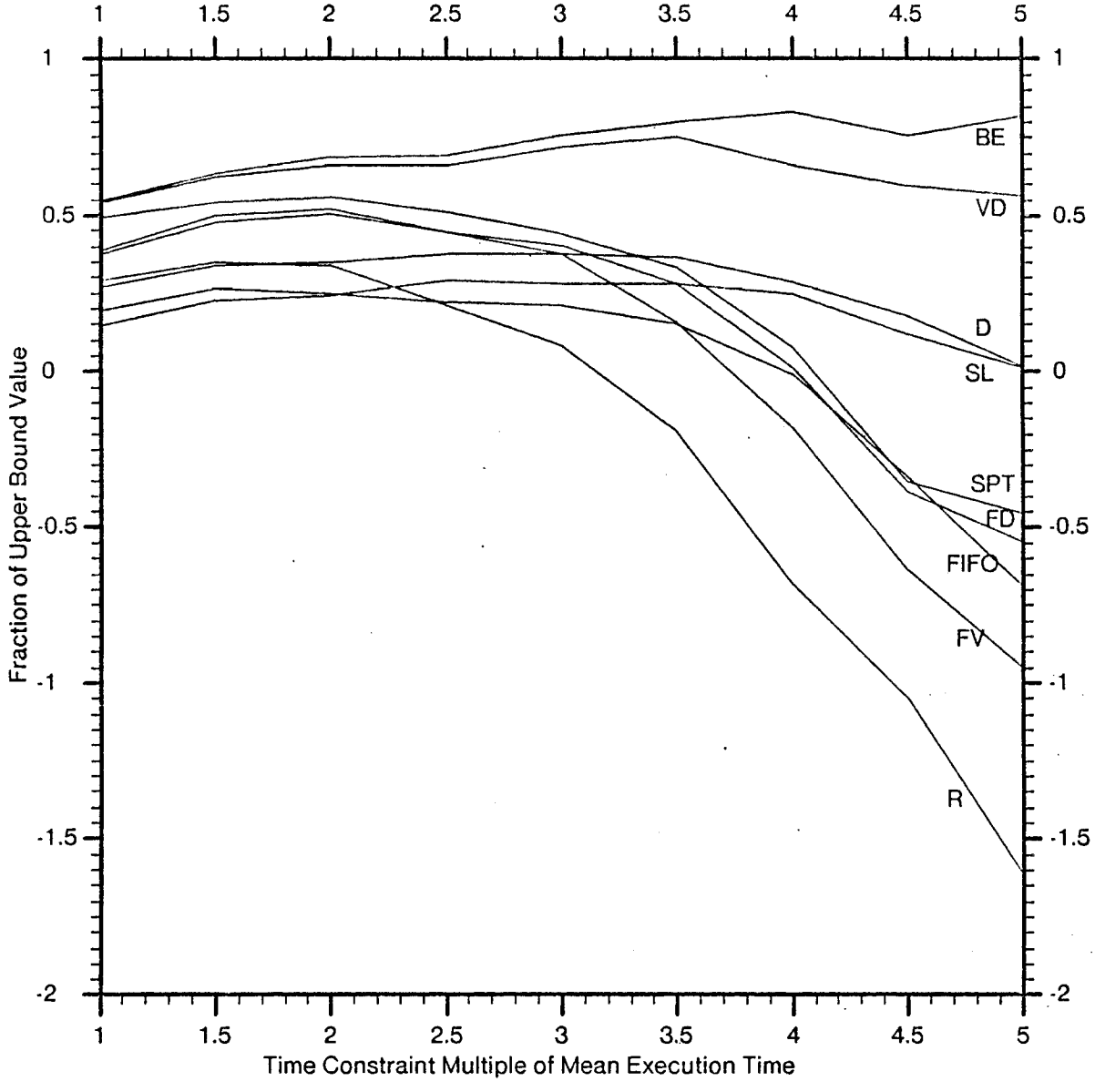

Figure 7-15: Scheduling Algorithm Evaluation with Varying Time Constraints (Various Value Functions)

schedulers (except the VD scheduler), at extremely loose time constraints. Upon closer observation, we can see that this is due to the presence of 6 processes with rising value functions, for which a high value can be achieved only by completing them near to their peak values at their critical times. This conclusion is confirmed by the small peak in their values between about 1.5 and 2.0, apparently due to the fact that the lateness due to the process load is causing the processes to frequently complete at about their critical times. This conclusion is also supported by Figure 7-16 in which the same processes were executed but with all processes drawn from the load1n set from Figure 7-1. Here, we see the expected approximately monotonically increasing curves as the time constraints loosen.

7.4.2.4. Known Execution Time Distribution

Having investigated the performance of the various scheduling algorithms in the presence of varying execution time knowledge (i.e., various execution time standard deviations), we turn to another aspect of the same problem. In this set of experiments, we vary only the execution time distribution, ensuring that the BE algorithm knows the distribution (for a discussion of the case in which the assumed distribution is not the same as the actual execution time distribution, see Section 7.4.2.5). Our simulator handles four execution time distributions (discussed in detail in Section 6.3):

- Normal
- Lognormal
- Exponential
- Bimodal

For each of these distributions, we ran two sets of simulations, one with a uniform combination of the four value functions described in Figure 7-1, and the other with only the step value functions, both sets containing 24 processes. The average load for all these runs is similar, and represents about a 175% average load.

The results of these two sets of runs showing the differences in execution performance under different execution time distributions are in Figures 7-17 and 7-18, each plotting the fraction of the upper bound value achieved for each of the four distributions, and for each of the schedulers. The principal effect shown by these runs is that all of the schedulers are sensitive to the execution time knowledge to some degree. We note that changing between the normal and lognormal distributions produces nearly identical performance for all the

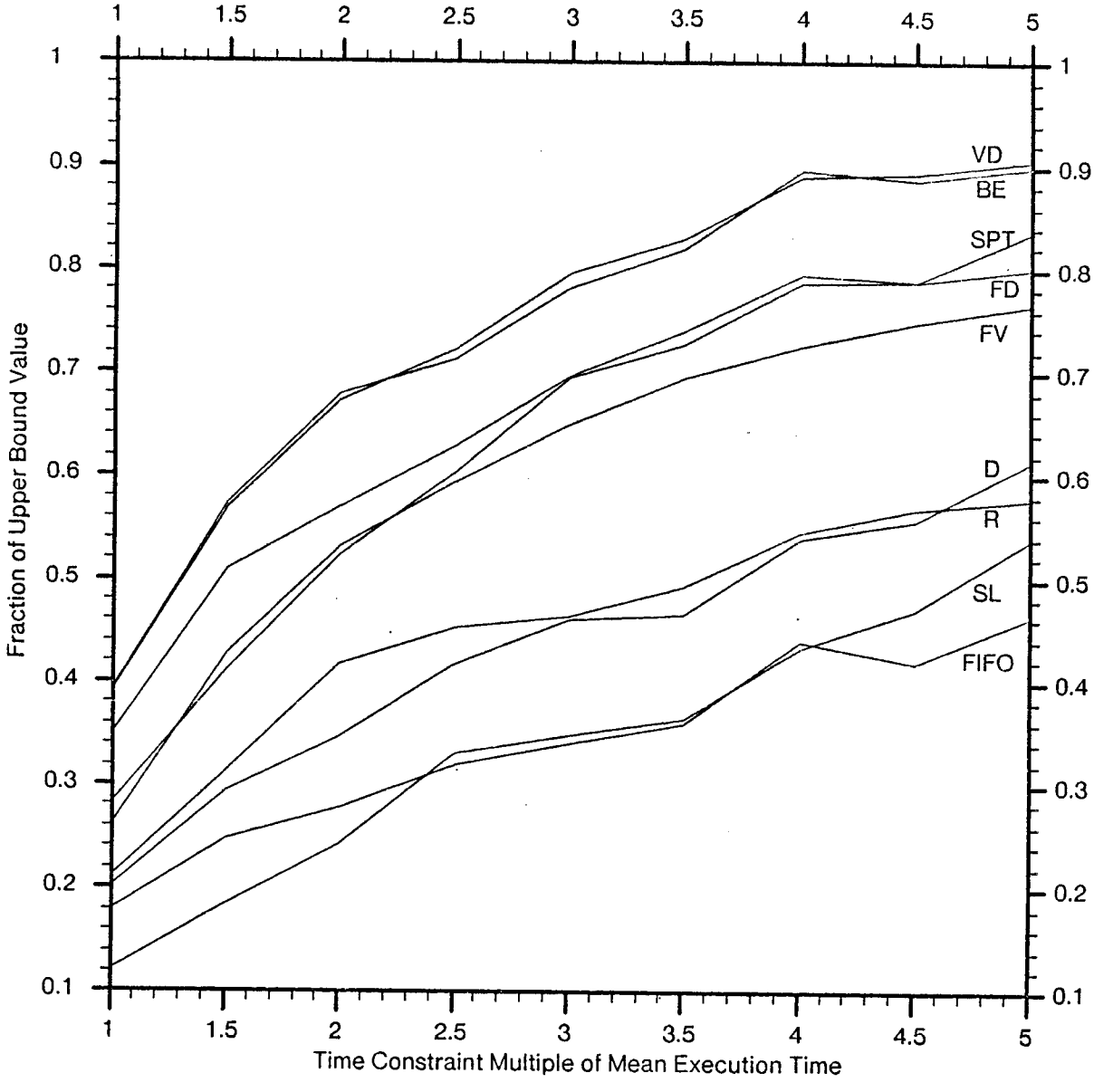

Figure 7-16: Scheduling Algorithm Evaluation with Varying Time Constraints (Step Value Functions)

algorithms, but changing to an exponential or bimodal distribution produces a larger variation in performance. There is evidence that this demonstrated change in performance for the exponential distribution is because its execution time variance is significantly different from that used by the normal and lognormal distributions. The variance for the exponential distribution with mean m is m^2, which is somewhat larger than the settings used by the normal and lognormal distributions, resulting in a somewhat lower total value. For the BE and VD schedulers, particularly, the inability of the scheduler to estimate the remaining execution time for a process under execution (due to the "memoryless" property of the exponential distribution" degrades its ability to make the scheduling decisions, although both of these schedulers still significantly outperform the other schedulers.

For the bimodal distribution, the variance also differs from that of the other distributions, but unlike the exponential distribution, the BE scheduler is able to make reasonable decreasing estimates of the remaining execution time as execution progresses, so the performance of the BE scheduler is still about the same as for the normal and lognormal distributions. As shown in Figure 7-17, the VD scheduler suffers for its inability to handle rising value functions with the bimodal distribution. The variance for the bimodal distribution is affected by the interaction between the two normal distributions from which it is generated.

In any case, the change in performance of the BE scheduler with respect to the choice of distribution other than for the exponential distribution, is small, indicating that the BE scheduler is able to effectively make scheduling decisions using whichever distribution is present. Thus, we conclude that the choice of distribution is less important than the quality of the execution time knowledge, as long as the distribution is known to the scheduler.

7.4.2.5. Unknown Execution Time Distribution

In the cases in which the actual execution time distribution differs from the distribution assumed by the scheduler, we would generally expect that the quality of the scheduling decisions for schedulers explicitly using execution time knowledge would suffer significantly. To investigate this aspect of the performance of the schedulers, we have executed a set of simulations in which each of the four distributions supported by the simulator are assumed by the scheduler (and by the expected remaining computation time computation), while the actual execution time distribution was different, using all of the other three distributions.

In these runs, we assume that the mean and variance of the execution time is known

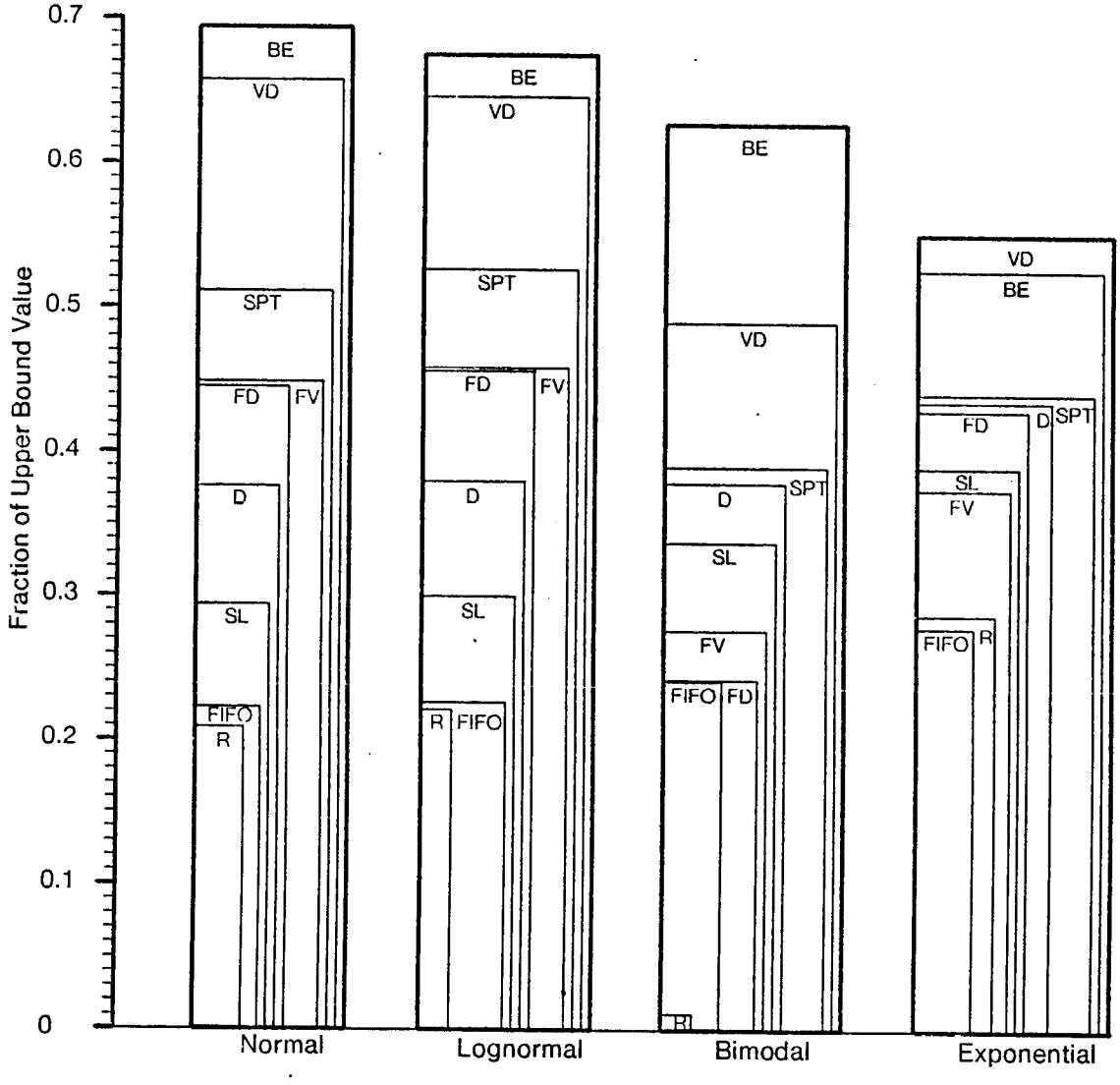

Figure 7-17: Scheduling Algorithm Evaluation with Varying Known Execution Time Distributions (Various Value Functions)

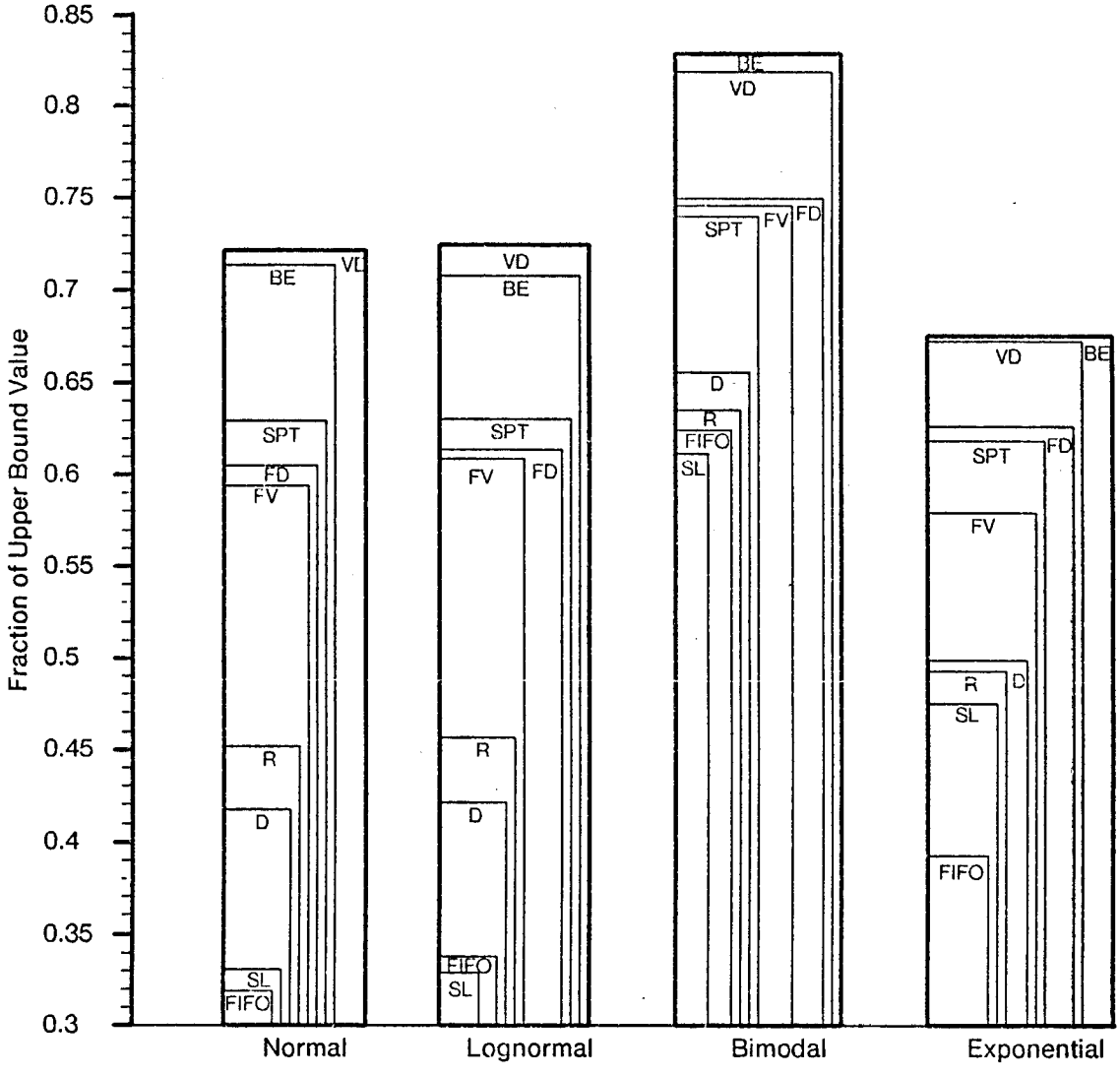

Figure 7-18: Scheduling Algorithm Evaluation with Varying Known Execution Time Distributions (Step Value Functions)

accurately; relating this assumption to an actual system, such a condition implies that the operating system has measured the mean execution time and variance over a long series of executions of each process. We assume, however, that the scheduler is not aware of the actual distribution of these times, and will thus make some incorrect scheduling decisions. Figure 7-19 contains four plots of such runs, each of the four assuming a different distribution, but actually using each of the distributions and a mixture of value functions; Figure 7-20 contains a similar set of runs, using only step value functions. Each plot is shown in the form of a histogram showing the fraction of the upper bound value achieved by the algorithm when the execution times are drawn first from its assumed distribution, then from each of the other three distributions.

Five of the algorithms tested do not use the execution time in their decisions (i.e., FD, FV, R, D, and FIFO), and therefore do not show any variation in the correctness of the execution time distribution assumption, so they were omitted from these figures. The remaining schedulers (i.e., BE, VD, SPT, and SL), as expected from the similarity in the shape of their density functions, show that the assumption of a lognormal distribution when the actual distribution is normal, or the reverse, has little effect on the scheduling performance.

From the point of view of decreased scheduling performance, it is clear that the most drastic change in performance is due to having an actual bimodal execution time distribution when the normal or bimodal distributions is assumed. This is not unexpected, since the probability density function is so different for this distribution, and for any process, there exists a significant probability that more execution time remains when the assumed distribution indicates that the process should have completed. This effect holds regardless of the use of mixed or step value functions.

The presence of exponentially distributed execution times when normal or bimodal distributions are assumed similarly presents a problem for the scheduling performance, but not to the same extent, since the density curve numerically favors a large number of execution times shorter than the mean (whose scheduling times will be met), but with a few extraordinarily long times, which will merely be aborted prior to their completion. As with the known distribution cases, the performance degradation results from the inability of the scheduler to predict the execution time of a partially completed process.

In applying these results to the design of an actual scheduler, it is clear that the better the scheduler's ability to predict the execution time, the better the predictability of its performance will be.

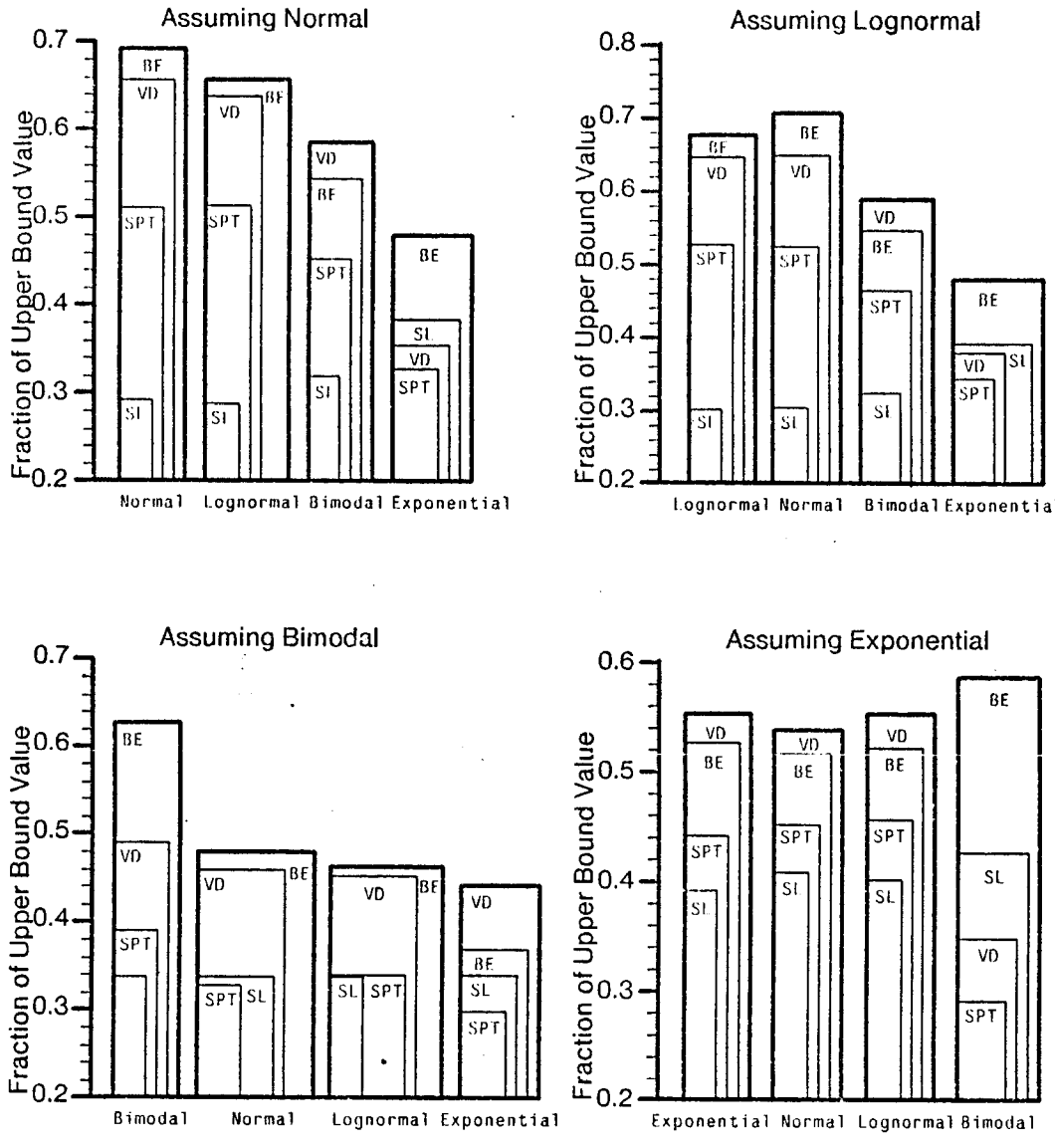

Figure 7-19: Scheduling Algorithm Evaluation with Varying Unknown Execution Time Distributions (Various Value Functions)

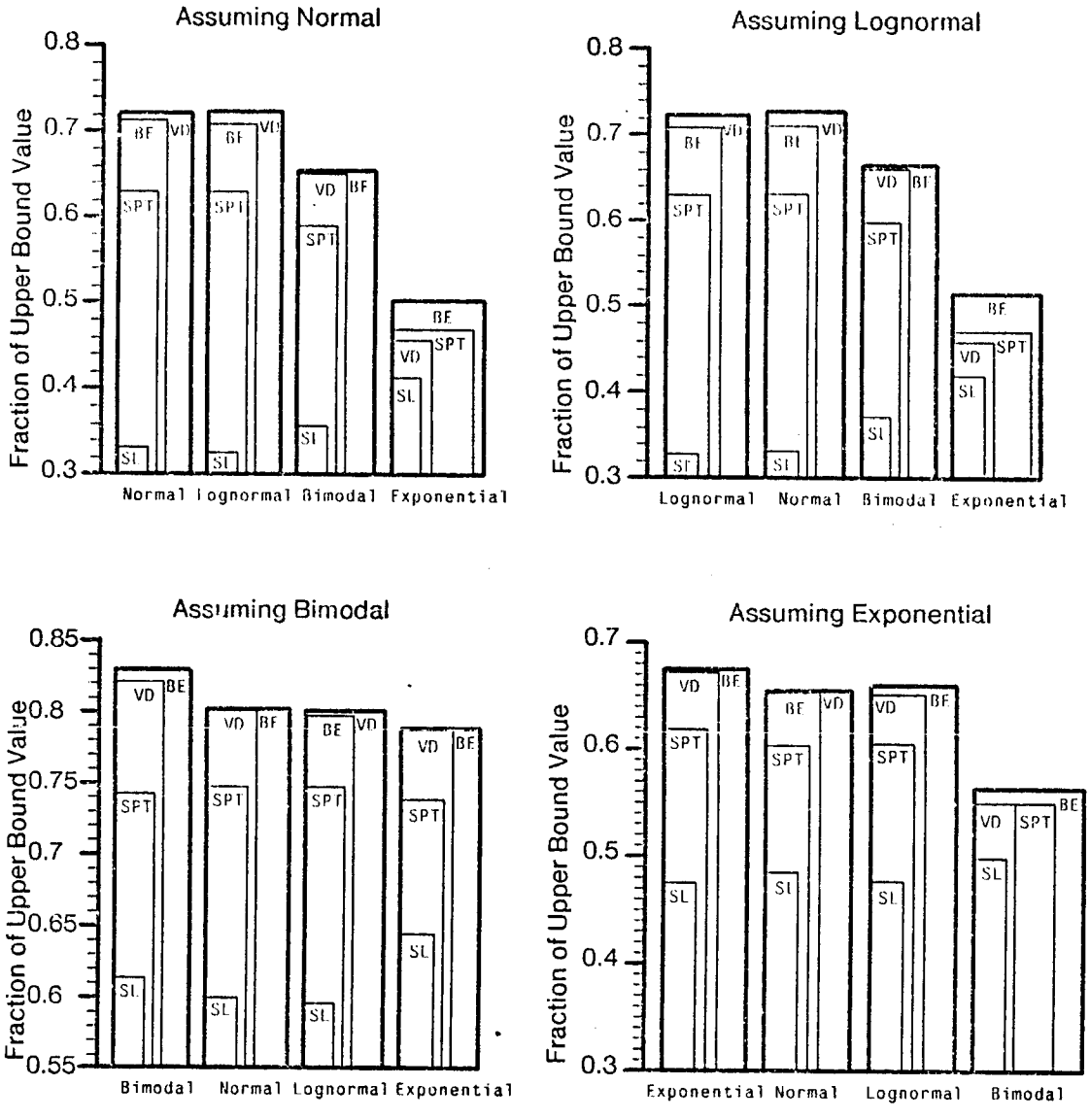

Figure 7-20: Scheduling Algorithm Evaluation with Varying Unknown Execution Time Distributions (Step Value Functions)

7.4.3. Individual Process Scheduling Evaluation

The preceding sections have concentrated on the performance of the BE scheduler with respect to statistical measures over many executions with large numbers of processes. It is also interesting to observe the actual performance of this scheduler from the point of view of the individual processes being scheduled.

In an overloaded real-time system, we would expect a time-driven scheduler such as the BE scheduler to consistently favor the execution of *important* processes over processes of lesser importance. The definition of importance, in this context, involves the amplitude of the value functions for each process. We have therefore graphically illustrated this value for each of the processes in one of the simulations described above.

In this case, the BE scheduler is scheduling a set of 36 processes drawn uniformly from the load1n set described in Figure 7-1, resulting in an average load of about 225%. For each of these 36 processes, we have produced a small graph which we have grouped together in Figures 7-21, 7-22, and 7-23. The vertical axis of each of these graphs is scaled the same, and represents the amplitude of the entire set of value functions, with the highest at the top, and a value of 0 on the bottom. The horizontal axis (the time axis) is scaled individually for each process, with the left edge representing the request time, and the vertical line within each graph representing the request time. In each graph, a horizontal line is drawn for each actual execution of the process, using a broken line if it was preempted, showing when in the request-critical time interval its execution took place. Requests resulting in a process abortion are represented by a small x to the right of the execution line, but if the process was never started for that request, the x is alone. Thus, the number of x's on the graph indicates the number of aborted requests for that process. Also plotted on the graph is the value function itself, showing its importance as the amplitude of the value function, and the time constraint as its shape. In addition, we have annotated each graph with its process ID number, its mean execution time (m), and the time from its request to its critical time(d). For processes whose value functions never rose above 0, the graph is blank.

This execution is a heavily overloaded case, so we expect to find that many processes were aborted, which is certainly confirmed by these figures. For example, Process 2, with a relatively low value, was never even started, but was aborted every time it was requested. Even such relatively high value processes as 4 and 12 were occasionally not completed due to the incidence of higher value processes using the available resources. It should be noted

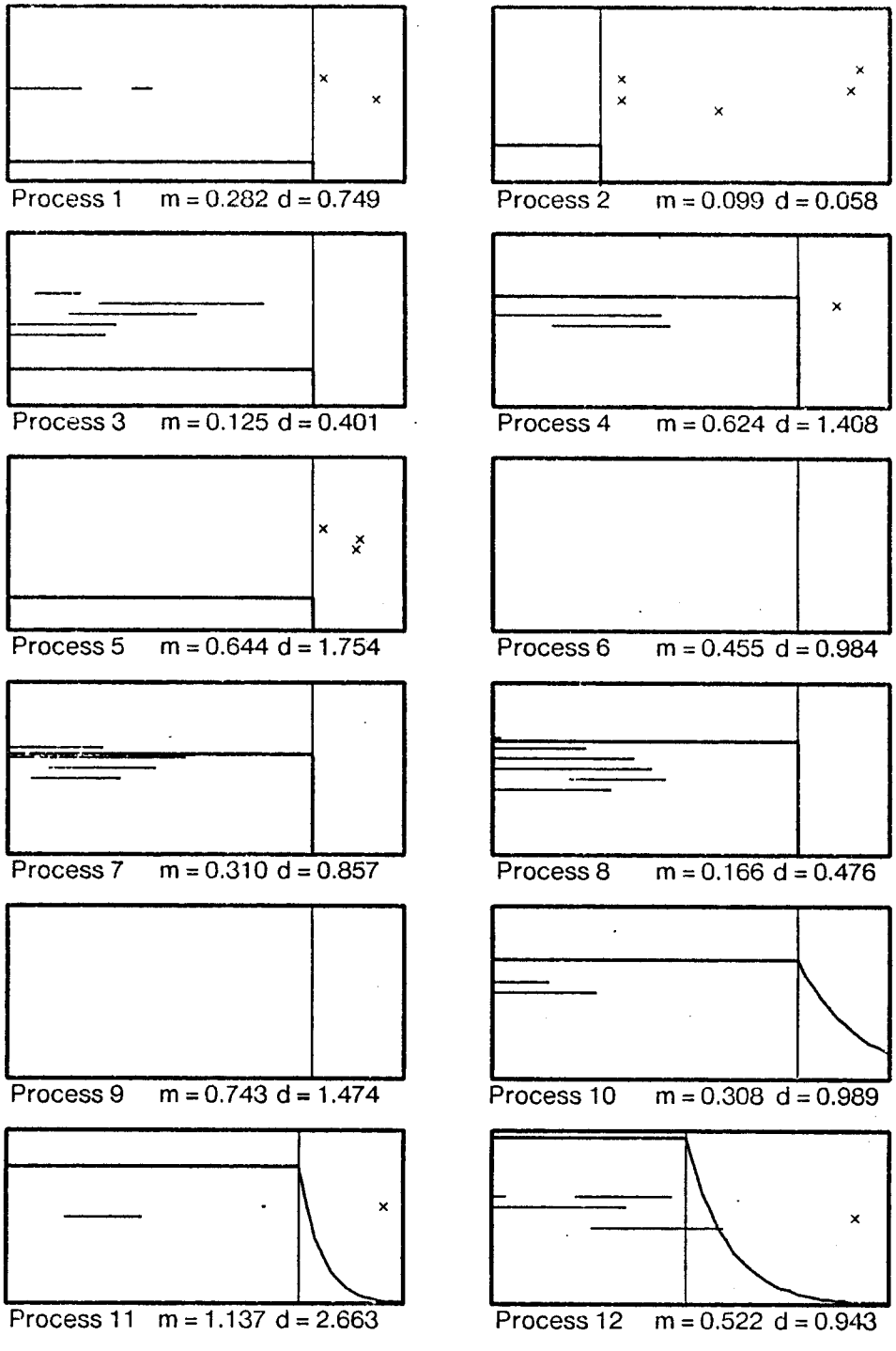

Figure 7-21: Individual Process Scheduling Performance (Part 1).

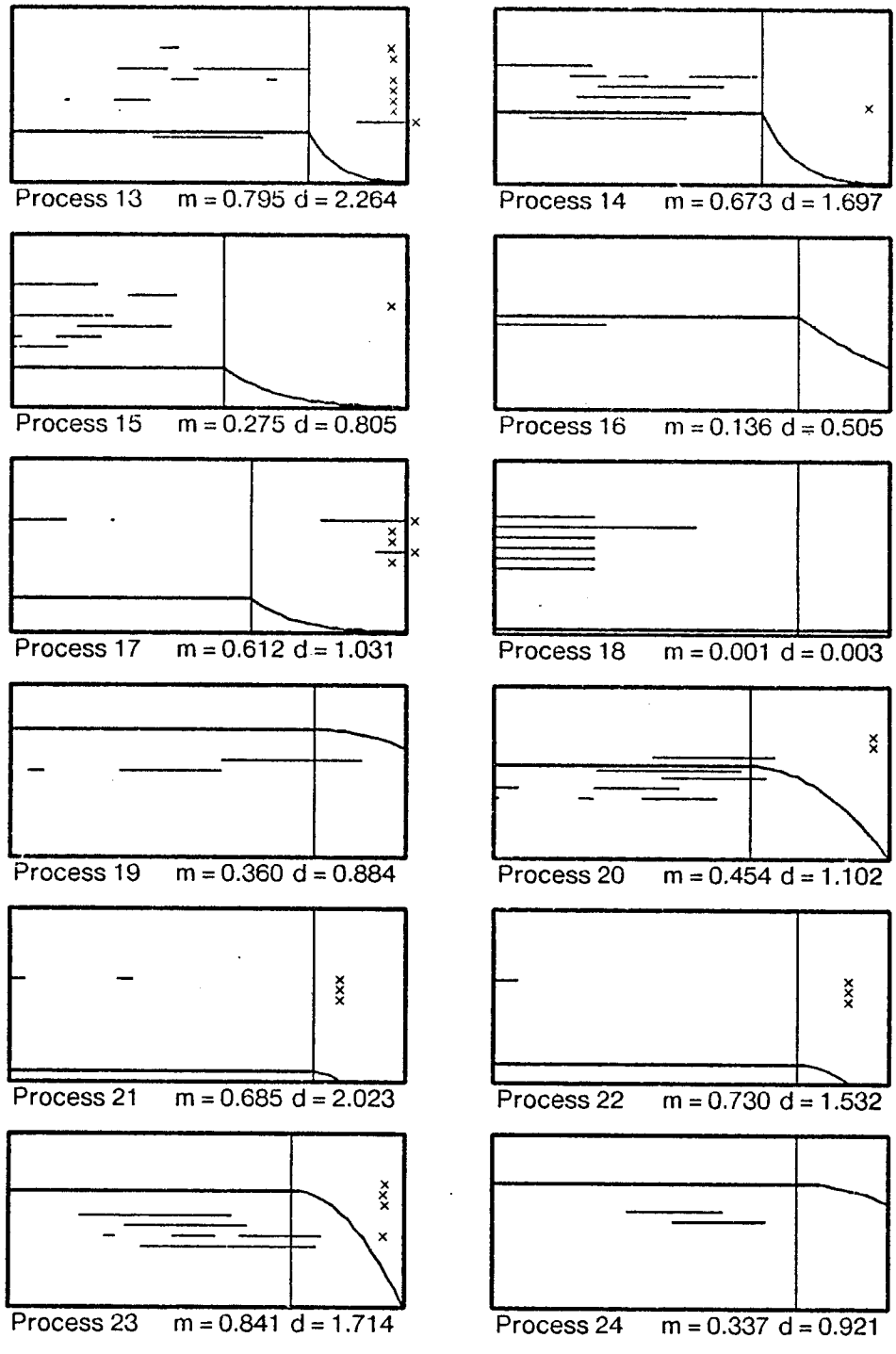

Figure 7-22: Individual Process Scheduling Performance (Part 2)

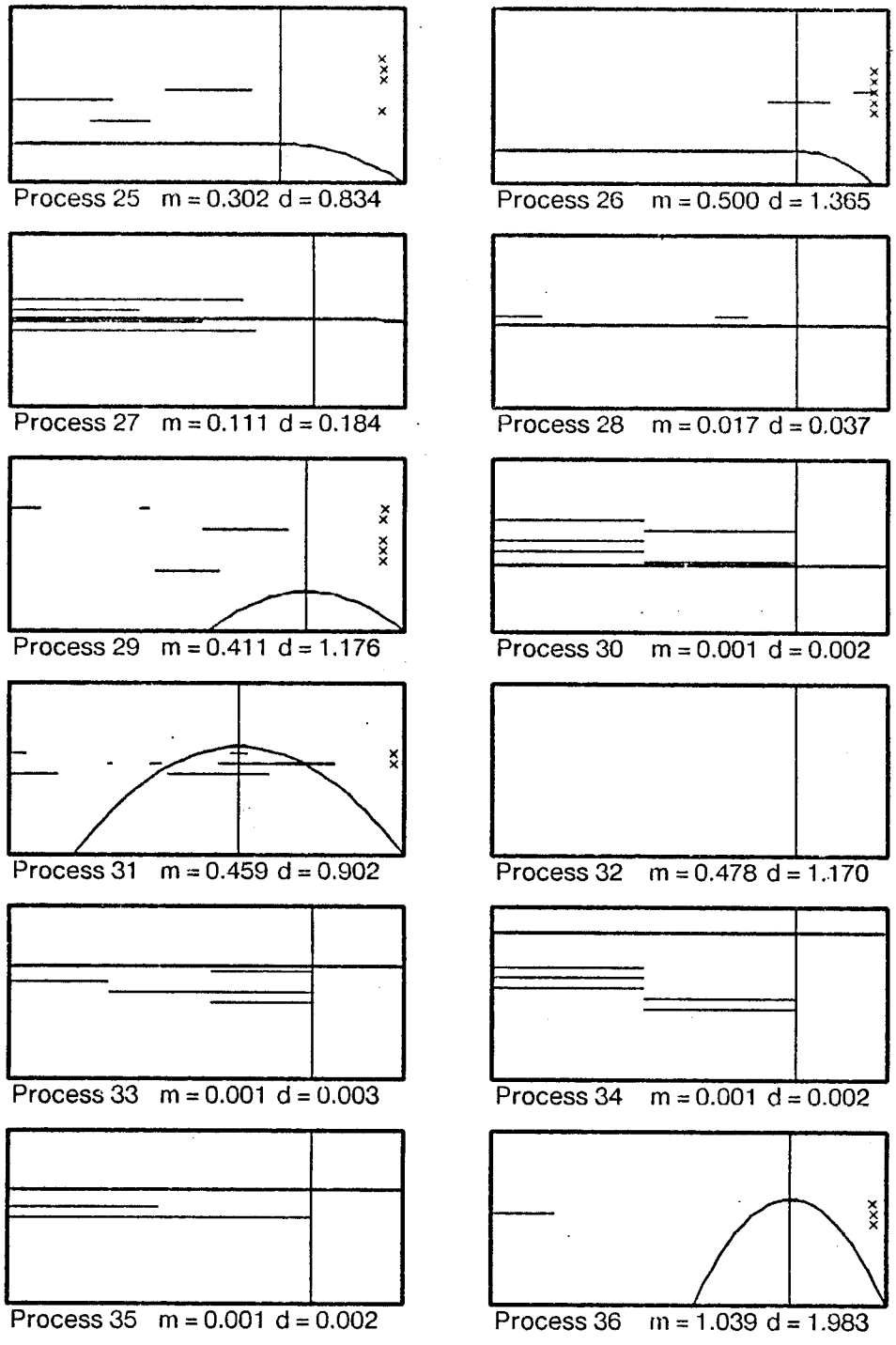

Figure 7-23: Individual Process Scheduling Performance (Part 3).

that these graphs include processes such as 6 whose value function never became positive, so it was never executed. Such a situation might correspond to a large real-time system with certain low-value processes which should not be using system resources during certain system modes.

The most important effect which can be seen in these figures is the overall consistency with which the higher valued processes were completed within their time constraints in spite of the extreme overload condition. The processes which were aborted were almost always the ones with low value, keeping the overall value to the system at a uniformly high level. In this simulation, all the maximum values range from a low of -2.7 for process 9 to a high of 12.7 for process 12. This range is normally distributed with a mean of 5.0; in many systems it would be expected that certain critical processes would have maximum values perhaps several orders of magnitude above the rest, ensuring that a scheduler such as our BE scheduler would always meet their time constraints.

7.4.4. Decision Cost Evaluation

Since the BE scheduler is intended for use in a real-time system, it is important to have an understanding of its expected execution cost. We have noted before that its worst case computational complexity is $O(n^2)$ (see Section 5.4.5), but this is not completely descriptive of its expected performance in an actual system, since an operational version of the BE scheduler can be designed to bound the size of the run queue (n), and the expected number of times that a process must be removed due to a detected overload (the portion of the algorithm which gives rise to the square term) should be small.

To provide some evaluation of this cost, we have measured the performance of the BE scheduler during the runs described to this point, counting each pass of the scheduler through each of its linear parts for each process and each decision made. In fact, referring to the BE algorithm description given in Figure 7-24, we have counted the number of times the algorithm reached each of the critical points named in the listing:

- C0 -- Number of schedules computed.
- C1 -- Number of times the ready time is computed.
- C2 -- Number of processes looked at by scheduler.
- C3 -- Number of times the pre-execution probability is computed by the scheduler.

- C4 -- Number of processes added to the available queue.
- C5 -- Number of times an overload needed to be fixed.
- C6 -- Number of processes checked to remove an overload.
- C7 -- (Not shown in Figure 7-24) Number of times the expected value was computed.

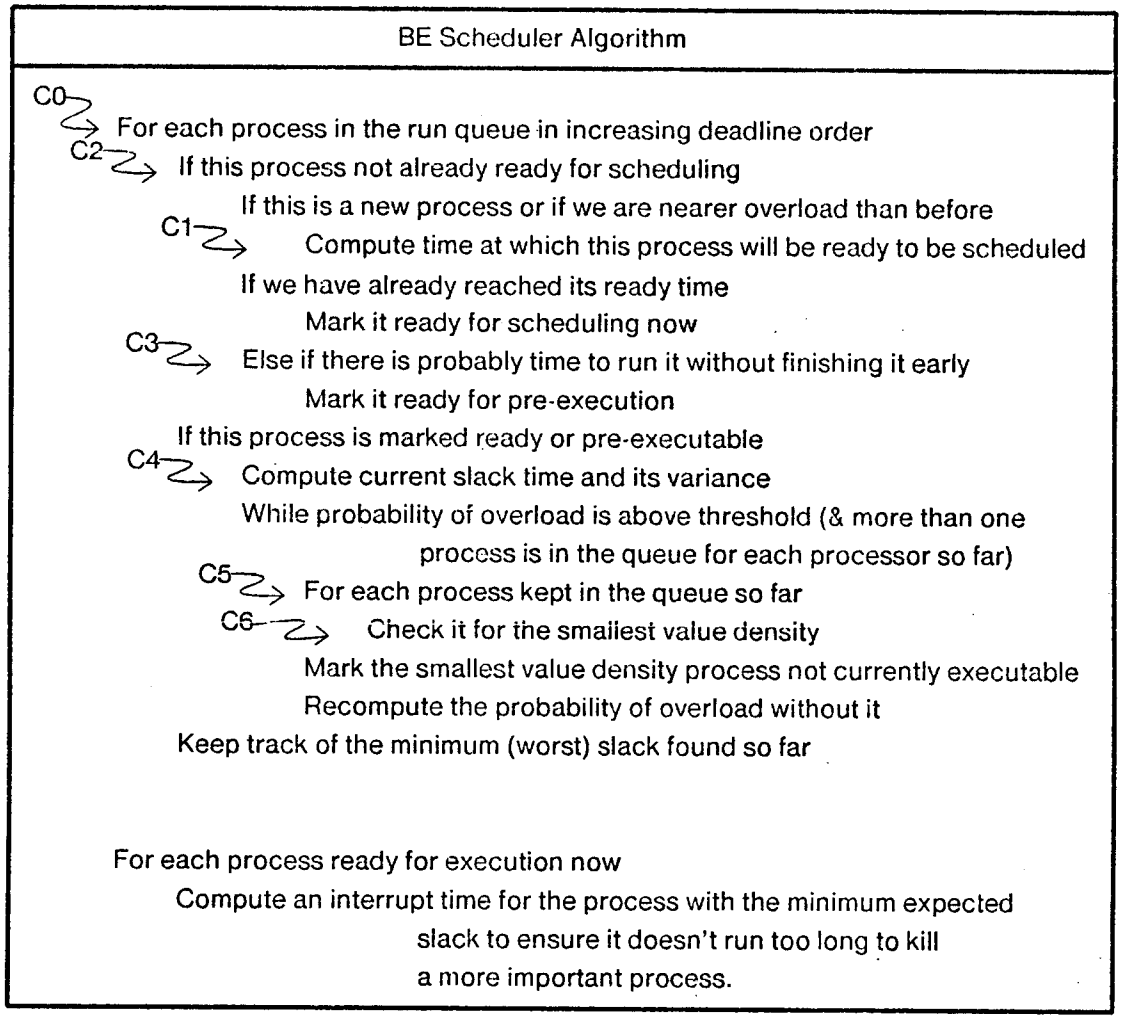

Figure 7-24: Instrumented BE Scheduler Algorithm

C7 is counted in the routine which computes the expected value, since this computation is not repeated for processes for which it is already available for the same scheduling decision.

The actual cost for making a decision is a function of the number of times these parts of the

algorithm are reached and the actual execution cost for a processor to complete each of these computations. Based on experience with similar computations on existing real-time systems, and assuming that an actual scheduler would use efficient approximations to the expected execution time and expected value computations, a set of approximate costs (in μseconds) for a hypothetical 1 MIPS processor was defined. These estimated costs represent rough engineering estimates whose correct value is not critical, but we are instead primarily interested in their relationships to each other. We will not be evaluating the actual cost of this algorithm precisely, but will rather be interested in the cost growth with increasing load. For these computations, we estimated costs to be:

- T_0 Time to compute ready time — 100 μseconds
- T_1 Time to check pre-execution probability — 20 μseconds
- T_2 Time to put a process in the execution list — 100 μseconds
- T_3 Time to remove a process from the execution list — 100 μseconds
- T_4 Time to check a process for minimum value density — 25 μseconds
- T_5 Time to compute its expected value — 400 μseconds

Clearly, the architecture of the processor will significantly affect these times, but we expect them to provide a useful evaluation of the simulator execution times.

Using these values and the counts from the simulations, we produced a set of plots of the estimated decision cost in μseconds versus average percent load for each of the runs described in Section 7.4.1. We include here two of these graphs:

- Scheduling decision cost per process (Figure 7-25), which is the overall scheduling cost for the simulation divided by the total number of processes present over all scheduling decisions.
- Scheduling decision cost per decision (Figure 7-26), which is the overall scheduling cost for the simulation divided by the number of times a new schedule was constructed.

In each of these graphs, we show 5 separate curves for each value function run described in Section 7.4.1, with each curve representing a value function type from Figure 7-1. Although there is a considerable amount of variability on the curves which include processes with rising value functions, the general shape of the curves shown in Figure 7-25 is significant. As load increases from about 15% to about 250%, the mean cost for each process grows

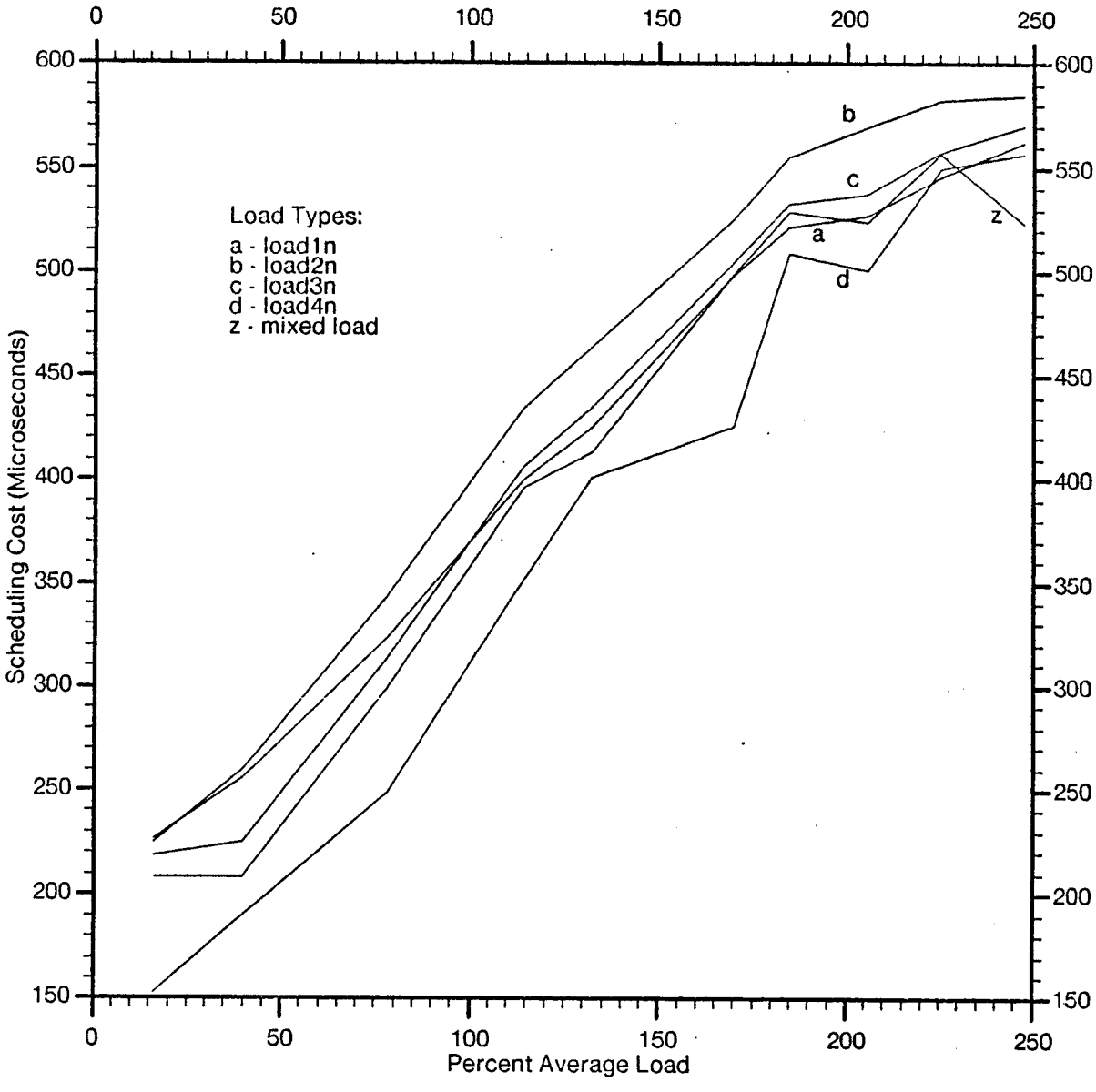

Figure 7-25: Scheduling Algorithm Cost per Process vs. Average Percent Load

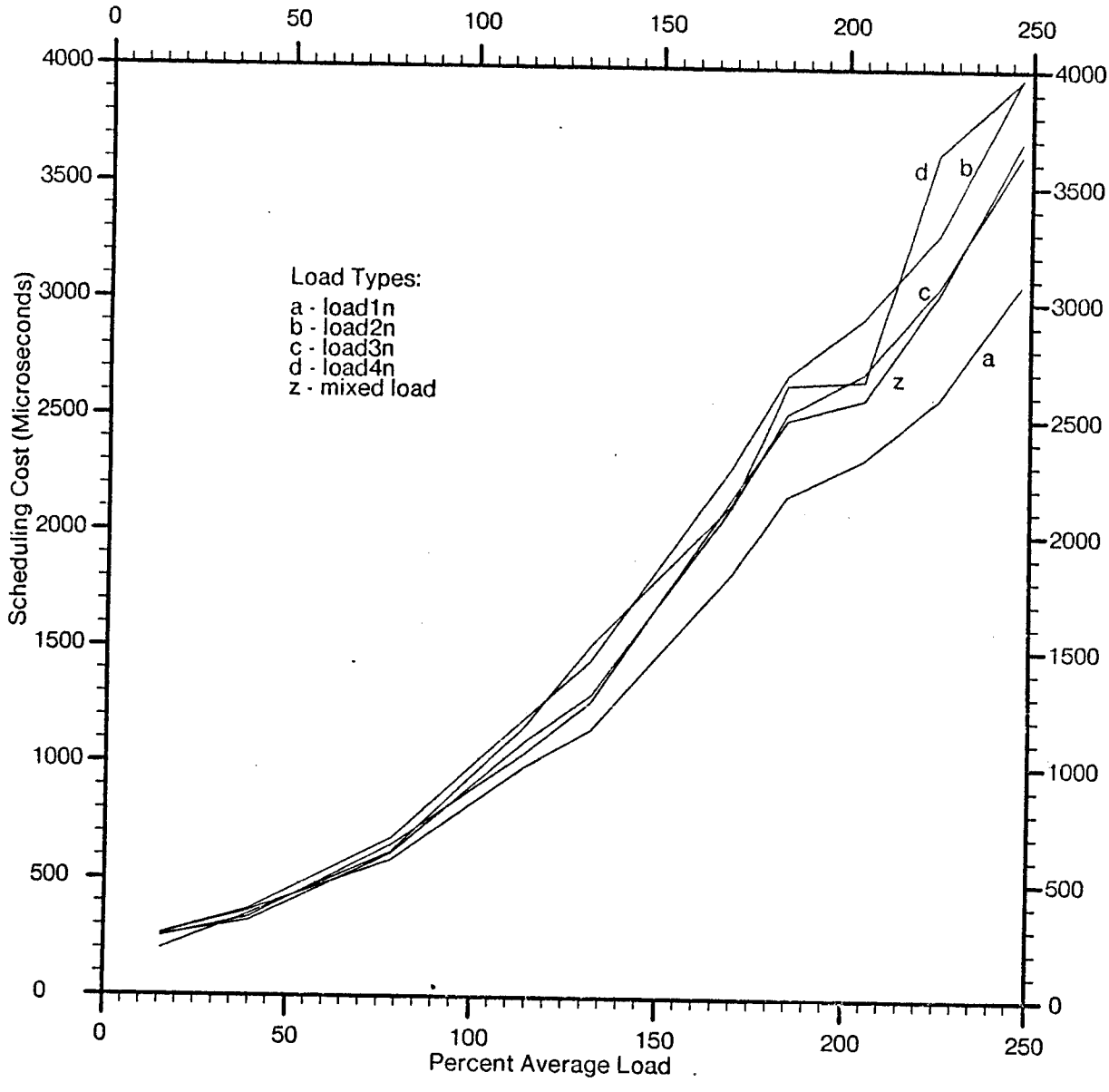

Figure 7-26: Scheduling Algorithm Cost per Scheduling Decision vs. Average Percent Load

approximately linearly once a load of about 50% has been exceeded. Although it is not shown on this graph, the algorithm is a simple linear one if no overload exists (see Section 5. This graph provides empirical evidence for the $O(n^2)$ complexity of this scheduler.

Similarly, Figure 7-26 gives evidence that despite its time complexity, the growth in cost over the load range studied here is not explosive. Obviously, the cost of each of the steps taken in the algorithm must be carefully controlled in an actual implementation, but the results, especially if a separate non-application processor is used for scheduling, lead us to believe that such a scheduler can be feasibly constructed.

The number of times a scheduling decision needed to be made varied with the number of processes were requested. It was found that when processes without rising value functions were used, the number of times the scheduler was invoked averaged about 1.8 times the number of requests, while for simulations exclusively with processes with rising value functions, the scheduler was invoked about 3.4 times the number of requests. This information confirms our expectation regarding the timing of scheduling decisions for various value function and load conditions.

7.4.5. Multiple Processors

Having described the performance of the BE scheduler for a single processor, we are interested in showing its performance in the presence of more than one processor, subject to the constraint that a single run queue be used for scheduling all the processors. This model corresponds to a shared memory multiprocessor, and we assume that the configuration is symmetrical with respect to processor speed.

Each of the schedulers tested in the experiments described above were implemented as *list* schedulers as described in [Ruschitzka 77], with each scheduling call resulting in a list of executable processors ordered by the sequence which the scheduler currently believes to be best at that time. Each of the first n processes on this list are then assigned to the n processors, then all of the processors continue until their assigned process terminates, is aborted, or until the scheduler believes that another process should be executed preemptively.

The assignment of the first n processes to the available processors constitutes a heuristic which will very frequently, but not always, result in the highest value to the system. For a more

detailed discussion of the potential problems with this approach, see Section 7.6. We expect, however, that the performance of this heuristic will provide a robust schedule usually providing both a very high value (at least for the BE scheduler) and consistent performance with reasonable utilization of the available processors.

As stated in Chapter 1, we expect that this research will be later extended within the Archons project to a distributed, real-time system, with each node consisting of a shared memory multiprocessor, and a scheduler similar to that described here used to schedule processes within each node. In such a system, we expect that each node will consist of a processor dedicated to operating system functions, accompanied by one or more processors executing application processes. In such a system, we would expect that the number of such processors in a node will be fairly small (probably less than 5), so the experimentation performed here on multiprocessor scheduling has been limited to 5 processors.

Our principal goals for the multiprocessor scheduling are:

- Consistently high value for a set of processes as the number of processors are increased. For this comparison, we look for the value to increase monotonically as the number of processors increases, asymptotically reaching a value representing the completion of all processes at their maximum values

- Consistently high value for a load which is at the same level per processor for each additional processor. For example, given a load running at an average load of about 150% on one processor, we would like to achieve close to the same total value with twice the load on two processors, three times the load on three processors, etc. This goal indicates that the scheduler is able to schedule each of the processors equally well, spreading the load at each scheduling decision to all processors.

Corresponding to these goals, we make two sets of experimental runs. The first set, resulting in Figure 7-27, contains an identical set of processes for each of five runs with 1 to 5 processors. This set was run twice; once using only step value functions, and the other with a uniformly mixed set of value functions, and was chosen to place an average of about 113% load on a single processor, proportionately less on the multiprocessor configurations. Thus, the single processor was slightly overloaded on the average, while the multiprocessor cases were underloaded on the average (although they still contained transient overloads within the request interval). Figure 7-27 consists of two graphs of the two iterations, each plotting the fraction of upper bound value achieved against the number of processors, and showing the results for all schedulers.

The second set, resulting in Figure 7-28 consists of a set of five runs with 1 to 5 processors, each running at about 113% average load. Thus, the load in each case as seen by each processor is about the same, although the total load with 5 processors was 5 times as great on an absolute basis. This set, also, was run twice, once with only step value functions, and the other with a uniformly mixed set of value functions. Figure 7-28 is similar to Figure 7-27, consisting of two graphs, each plotting the fraction of upper bound value achieved against the number of processors, showing the results for all schedulers.

Interpreting these graphs is rather straightforward. In Figure 7-27, we clearly see the monotonically increasing value as the number of processors increases, asymptotically approaching an optimum value as the number of scheduling conflicts decreases. This effect is apparent for all of the schedulers when only step value functions are present, but is true only for the BE algorithm when mixed functions are used. In the step function case, we note that the optimum value is approximated somewhat earlier by those algorithms explicitly using knowledge of the expected execution time (i.e., BE, VD, SPT, D, SL) than for the other schedulers. In Figure 7-28, we note the very consistent performance of the BE scheduler, regardless of the nature of the value functions, with somewhat more variation with number of processors shown by the other schedulers. Since none of these schedulers is explicitly taking the existence of multiprocessors into account in its scheduling decisions, we conclude that the consistent performance of the BE scheduler is due to its previously demonstrated performance consistency regardless of value function mix or overall load.

7.5. Critical Decisions Analysis

In this section, we perform a detailed examination of some actual scheduling decision situations encountered in the runs described in Section 7.4. Our goal here is to see some of the critical situations in which the scheduling decision produces a higher or lower total value in a real-time environment. Here, we view two actual situations which are representative of the decisions made by the BE and other schedulers, and indicate the reasons for the consistently satisfactory behavior of the BE scheduler in the presence of processes with different value functions.

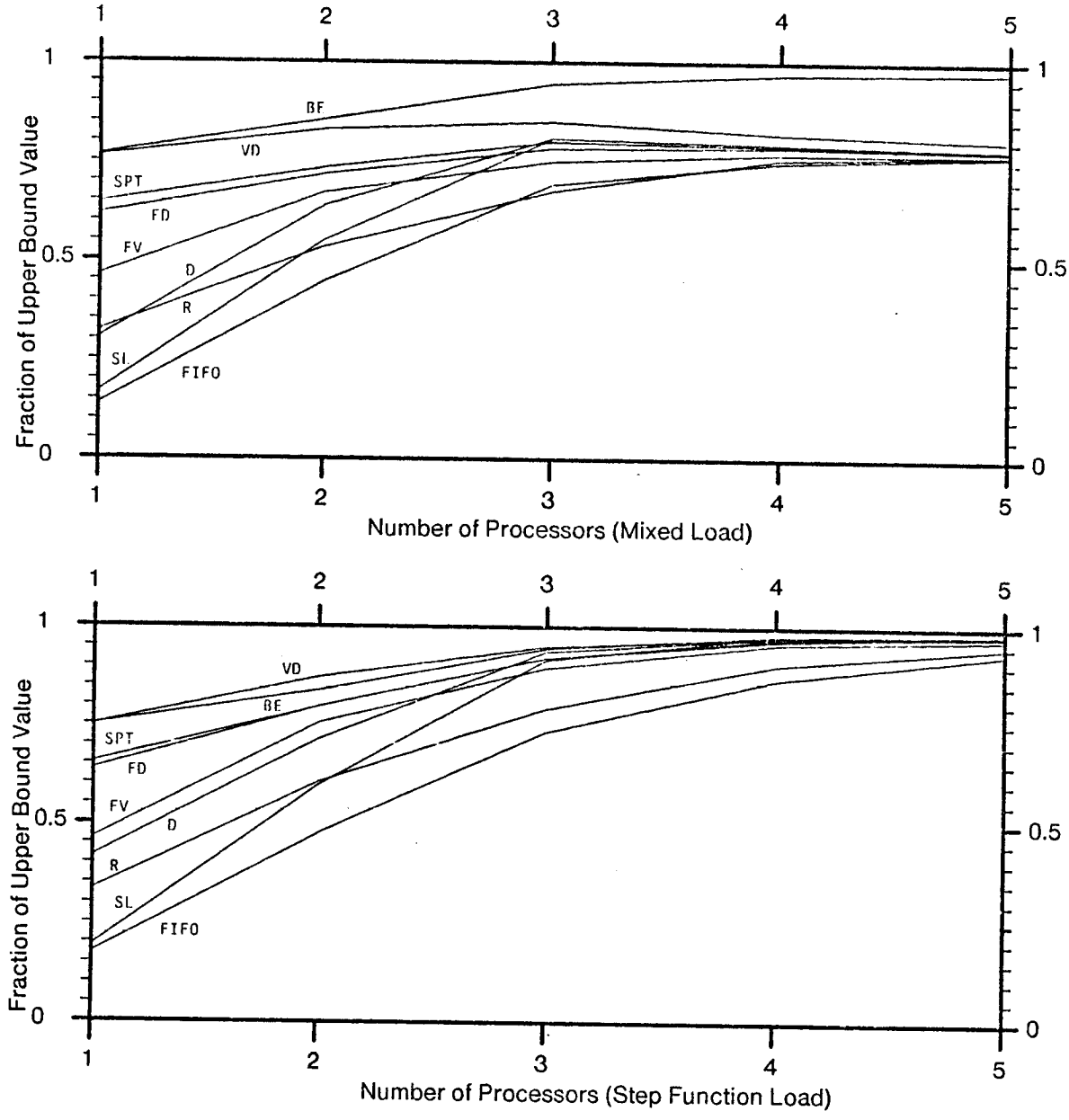

Figure 7-27: Multiple Processor Scheduling with Constant Load

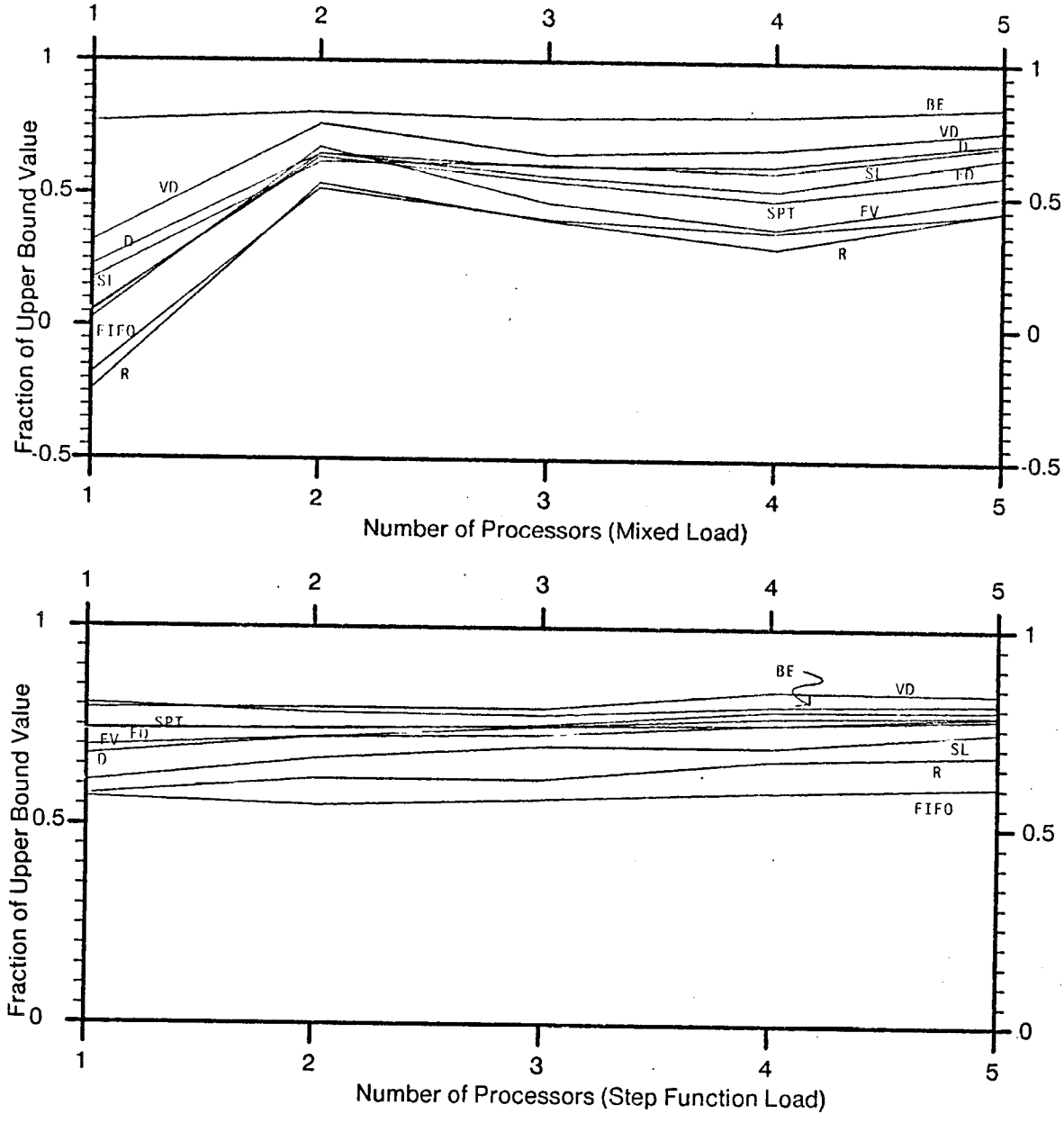

Figure 7-28: Multiple Processor Scheduling with Proportional Load

7.5.1. Greediness

There are several important differences in the decision making behavior between the BE scheduler and the others included in our experimentation. Of particular interest is the similarity of the BE and VD schedulers which, under some conditions, produce comparable total values. In the decision discussed here, we see the effects of their differences.

In this decision, we have a queue consisting of these processes:

Process ID	Expected Run Time	Time to Deadline	Peak Value	Expected Value Density	Deadline Priority
3	.115	.389	2.7	22.93	4
13	.617	.515	3.8	3.35	23
19	.355	.884	9.8	27.04	9
22a	.720	1.432	1.4	1.80	15
22b	.720	1.485	1.4	1.79	15
14	.663	1.686	5.4	7.83	17
11	1.121	2.582	10.5	9.19	27

The BE scheduler, as described in Chapter 5, starts with a deadline schedule (the order of this list), then attempts to remove any overload. In this case, an 80% probability of overload is detected when process 13 is considered, forcing either process 3 or 13 to be removed. Based on their value density, process 13 is dropped. The probability of overload is not high again until process 22b (the "b" means that this is the second instance of process 22 to be requested in this run queue) is considered, at which an overload probability of 87% is detected. At this point, process 22b is dropped, and the processing is repeated until the deadline ordered, non-overloaded list 3, 19, 11 is generated, resulting in executing process 3. Note, however, that process 3 is not the most important process in the list (indicated by the peak value shown), but the BE scheduler believes that it can complete 3 in time to still process 19 and 11, so it does not remove it.

The VD scheduler, on the other hand, greedily picks process 19, since it has the highest value density, thus dooming process 3 to be aborted, since its deadline is too near to allow it to be completed after 19. The BE algorithm would have made this same decision if the overload had required choosing between 3 and 19, but such a decision was not needed in this and many similar cases.

Process 3 was picked at this point not only by BE, but also by D, SPT, and FD, but their decision making was flawed in that none of them noticed the impending overload, and would

have made this same decision regardless of the overload. Only the VD scheduler would have picked 19; the SL and FIFO schedulers would have picked 13, resulting in the probable loss of process 19.

Decisions of this type occur frequently in overload conditions, but as the overload worsens, the difference between the BE and VD schedulers diminishes, since the computed value density becomes a more important decision factor than the deadline order. The criticality of these decisions becomes most evident in the presence of load "spikes" in which only a few processes must be aborted in order to consistently execute the most important processes.

7.5.2. Rising Value Function

In this case, a process (process 37) is requested with a rising value function which peaks 1.125 seconds after being requested. We describe the run queue and sequence of BE scheduling decisions made at each decision time, printing an asterisk by the process scheduled at that time:

Time	Process ID	Expected Run Time	Time to Deadline	Completion Time	Completion Value
22.819	21	.675	.570		
	13*	.373	.628		
	37	.449	1.125		
	11	1.121	2.486		
22.824	21	.675	.564		
	13	.369	.622		
	19*	.355	.884	23.237	9.7
	37	.449	1.120		
	11	1.121	2.481		
23.237	21	.675	.152		
	13	.369	.210		
	37*	.449	.707		
	11	1.121	2.068		
23.422	21	.675	-.033		
	13*	.369	.025		
	37	.274	.522		
	11	1.121	1.883		
23.503	21	.675	-.114	aborted	0.0
	13	.291	-.056		
	37*	.274	.441	23.715	6.4
	11	1.121	1.802		

Here, we note that process 37 arrives in a non-overloaded situation, as long as process 21 is

ignored; since 21 has already been delayed until it is unlikely that a reasonable value could be obtained (note that 21 has an exponential value decay). The BE scheduler has determined that 37 could be pre-executed for a period of time if time is available, but because of the lack of overload, sticks with the simple deadline schedule by executing 13 (which has a quadratic value decay, prolonging its useful life by .3 or .4 seconds). A short time later, at time 22.824, process 19 arrives with such a high value density, that its competition with 13 causes 13 to be preempted. Process 19 executes, completing at time 23.237 with a very high value. By this time, process 13 is competing with 37, and 37 has a higher value density, so 37 is scheduled. This is really too early for 37, so it is removed at time 23.422 to prevent its premature completion, returning the processor to 13. At time 23.503, control is returned to 37, permitting it to complete at time 23.715 with nearly its peak value of 7.3.

This sequence has significantly enhanced the value of 37 over, for example, the VD scheduler, which would have begun 37 as soon as 19 completed, finishing it much too early for a high value. Situations such as this one account for the significantly higher value attained by the BE scheduler in the presence of rising value functions.

7.6. Algorithm Areas of Weakness

The BE scheduler makes its decisions according to a set of heuristics, and as with any heuristic-based decision maker, will not always make an optimal decision. In this section, we consider situations in which sub-optimal decisions may be made, assessing the significance of these decisions in a real-time environment.

7.6.1. Non-Determinism

This research is based on a stochastic paradigm in which it is not possible to absolutely predict the processing of any particular application function, but rather it has been our goal to make significant improvements to the overall probability of completing real-time processes at a time which will produce a high value for the system.

There are many situations in which a real-time system must not only achieve a high value with high probability, but in which a particular process must be "guaranteed" completion at a particular time (e.g., processing critical sensor data which will be overwritten if processing is delayed). The inability of the scheduling techniques described here to provide such a guarantee can be considered a weakness in the application of these ideas to certain real-time system domains.

We characterize applications containing processes requiring such "guarantees" as highly cyclic applications, in which all or almost all system inputs and outputs occur at regular intervals, and hence the processing load is highly predictable. Such systems, including applications such as sonar processing or industrial process control, provide inputs at regular intervals and produce outputs at similar intervals, and only infrequently process aperiodic interrupt-driven inputs.

While our inability to guarantee processing access to such systems might be a drawback, we note that a designer using a scheduler such as the BE scheduler described here has a number of options in setting up the value functions which can alleviate this problem. The use of the simple deadline schedule which remains intact in the absence of overload ensures that a set of periodic processes with deadlines which do not themselves cause overload will be scheduled to meet their deadlines. This predictability is essentially the same as is currently provided in many existing real-time systems, which provide for such repetitive processing with fixed time slots, except that the deadline-driven schedule permits utilization of the unused time in each time slot currently wasted in such systems. Given the absence of overload due to aperiodic load, the BE scheduler will perform exactly as well as the deadline scheduler, providing optimal time-constraint processing in such a single processor environment. Even in the presence of an overload, careful use of extremely high value functions for such a set of mutually non-interfering processes can essentially "guarantee" their completion at the expense of processes with lower value functions.

The aperiodic inputs may occur as a result of two kinds of events:

- Highly important "emergency" events.
- Low importance "background" events.

7.6.1.1. Emergency Event Scheduling.

Existing real-time systems with high periodic processing loads suffer from exactly the problems described in Chapter 1; performance remains completely predictable until an unexpected (and, it is to be hoped, unusual) flood of aperiodic activity occurs, possibly as a result of a massive failure or emergency in the application. With such scheduling techniques as time-slot scheduling, all guarantees are gone, and the response to such a situation is completely unpredictable; such an occasion is, however, exactly the wrong time for this unpredictable behavior to occur, since appropriate emergency response is critical.

Using a BE scheduler, the designer can evaluate the processing requirements of each process, including both the normal periodic processes and the emergency interrupt processes, specifying their response time requirements and their importance in the single value function (see Chapter 4 for a discussion of the use of value functions for determining system policy) for each process. Thus, in the absence of the aperiodic processes, the ability of the non-overloaded system to meet time constraints can be guaranteed, while the predictable performance in reaction to an emergency overload will also be attainable, naturally at the cost of missing a few of the "normal" periodic time constraints.

7.6.1.2. Background Event Scheduling

The other form of aperiodic events occurring in a predominately periodic real-time system is relatively low-importance events such as operator switch depressions or I/O device completion. In a time slot driven system, these are typically handled by assigning all of them to a time slot which checks for its occurrence each cycle and handles it then. If, as is usually the case, no such event has occurred, the slot is wasted.

Using a BE scheduler, the designer can simply assign to each such event a value function whose shape reflects its time constraints, and whose amplitude reflects the importance of the event relative to the periodic processing and other events. Assuming a relatively low importance is assigned, the scheduler would naturally ensure the completion of the more important periodic processing, completing the event-driven processes as time is available. Thus, while "guarantees" are not possible in such a system, the use of value functions to define time constraints and importance should ensure that all non-overload time constraints are met.

7.6.2. Planning vs. Greediness

The reader will have observed that for many types of value functions and for some load levels, the VD scheduler performs as well as, or occasionally slightly better than, the BE scheduler. This results from the fact that at low load levels, the choice of scheduler is relatively unimportant as long as they both make scheduling decisions using value function data, while at very high load levels, the BE scheduler will essentially schedule by value density, having eliminated all low value density processes.

The fact that the VD scheduler occasionally outperforms BE, however, might come as a surprise from this analysis. This results from the relative performance of a "greedy" algorithm (i.e., VD) and a forward-looking algorithm (i.e., BE). Consider a request sequence as follows:

Request Time	Process ID	Value Density	Expected Run Time	Time to Deadline
0	1	5	2	3
	2	10	3	6
4	3	20	4	5

At time 0, the BE scheduler will execute process 1, since it expects that both processes can make their deadlines, while the VD scheduler will execute process 2, causing 1 to be aborted. Neither, of course, foresees the impending arrival of process 3. When process 3 is requested, the BE scheduler has been executing process 2 for 1 time unit, but now detects an overload and switches to process 3, since process 2 has a lower value density. Thus, when BE completes process 3, it has achieved a total value of 25 from completing processes 1 and 3. VD, however, completed process 2, sat idle for 1 time unit, then completed process 3, achieving a total value of 30.

This effect is due to the greediness of the VD scheduler. BE is assuming that the probability of a high value density/low slack time process being requested is low, so it expects to achieve a high total value by completing all processes as long as an overload does not occur. The VD scheduler, however, is pessimistic, and assumes that if it doesn't take the value as soon as possible, the opportunity will pass by. The difference, then, in the total value to be achieved depends on the probability with which the scheduler's assumptions about future requests are correct.

In general, we would expect that in an actual implementation, the use of forward planning would be preferable to a greedy approach to scheduling, but the architect of the system would need to understand the expected pattern of requests to determine the best approach for that application.

7.6.3. Multiple Processor Anomalies

In Section 7.4.5, we described the approach used by these schedulers to provide scheduling for a shared-memory multiprocessor. This method has been shown empirically to produce good results for a multiprocessor, but there are cases in which the schedule is non-optimum. Consider the following list of processes output by the BE scheduler in a two processor system, reflecting the order in which these processes should be executed to meet their time constraints (of course, the BE scheduler has removed the overload, if any, from this sequence).

Process ID	Expected Run Time	Time to Deadline
1	2	4
2	3	6
3	6	7

This schedule, assuming that the expected run times are correct, could be met by assigning processes 1 and 3 to the two processors, then assigning 2 to the available processor when process 1 completes. However, the method we used would have assigned processes 1 and 2 to the two processors, thus forcing process 3's deadline to be missed.

While this effect does occur, our simulation showed that the probability of its occurrence is very low. Furthermore, we experimented with heuristics to correct such situations, and we found that because of the stochastic nature of the execution times, the resulting schedule modifications were not sufficiently accurate to result in a significant improvement, while the cost was rather high. It is this effect which causes the general problem of optimal multiprocessor scheduling to be so difficult.

7.7. Overall Algorithm Evaluation

In this chapter, we have attempted to demonstrate the performance of a scheduler, here called the BE scheduler, which explicitly uses the application specified time value function of a process to make scheduling decisions. The principal design goal for this scheduler was to consistently achieve a high total value from the scheduling of a set of processes under time constraints. The resulting time-driven scheduler has been demonstrated to produce such a value in the presence of:

- A wide variety of loads, including both transient and persistent overloads, and including very heavy overloads.
- A wide variety of value functions, including both rising and level value functions prior to the critical time, and including both homogeneous and mixed value functions.
- Either single processor configurations or multiple processor configurations.
- Varying amounts and quality of execution time knowledge.
- Varying execution time distributions and knowledge of that distribution.

Chapter 8
Observations and Conclusions

The problem of scheduling processes in a real-time system is a difficult one, and has been studied in a number of contexts for many years. Because of the difficulty of the problem, practitioners in this field have resorted to simple techniques, such as fixed priority schedulers and FIFO schedulers, in an attempt to produce acceptable systems, designing them to ensure that the available resources are adequate to provide service at all expected load levels. As a result, such systems are both extremely expensive and excessively unreliable when the processing load grows beyond the designers' expectations.

In this research, we established an ambitious set of goals with respect to real-time process scheduling:

1. To determine a way to make the process scheduling decisions in a real-time computer system which would explicitly consider process importance and time constraints in each decision. We expected that this effort would require creating and evaluating a set of heuristics with which time-dependent value functions could be used to allow a system to achieve a high user-defined value by scheduling processes at the correct times.

2. Using these heuristics, to create a scheduler for a shared-memory multiprocessor which will use these time-dependent value functions to schedule processes, making a "best effort" to maximize the overall value to the system.

3. To construct a flexible simulator with which to evaluate this scheduler in a fully preemptive, single queue, multiprocessing environment.

4. Using this simulator, to evaluate this scheduler according to three characteristics:

    - In comparison to the other scheduling algorithms in use, or proposed for use, in existing operating systems (whether real-time or not).

    - In comparison with the actual decisions made at critical scheduling times with the best decision which a human expert would be able to make.

    - In comparison with an upper bound on the value achievable with a particular set of processes in the time allotted to them.

5. To determine how such a scheduler could be used in an actual real-time system to fulfill user policy objectives.

This effort has resulted in significant progress against these goals:

1. We have created a scheduler which performs with a high degree of consistency over a wide range of time value functions and with any of four computation time distributions (i.e., Normal, Exponential, Lognormal, or Bimodal) as long as the value decreases in some fashion following the critical time. This scheduler makes scheduling decisions using a set of relatively simple heuristics, determining a "good" time to schedule each process, detecting an overload situation and responding to it by allocating the processor to those processes which will enhance the overall value to the system.

2. A simulator has been constructed that realistically provides an environment in which a preemptive multiprocessor scheduler can schedule a set of suitably defined processes over a finite period of time, measuring its performance according to a number of characteristics. This simulator provides multiple computation time distributions, the ability to define a processing load which is generally overloaded or underloaded, and the ability to define "spikes" in processing load which can substantially change the load characteristics for subsets of the execution interval. It allows processes to have either homogeneous or non-homogeneous sets of value functions, and can repeat a particular set of process requests for as many as nine different schedulers which have been designed to implement each of the commonly used scheduling algorithms.

3. A large number of evaluation executions of our scheduler against widely varying conditions of load, value function, processing time distributions, number of processors, and run-time knowledge, have been made, providing a large body of statistics demonstrating the performance of the scheduler. In addition, a large number of schedules have been evaluated manually, a few of which have been described here in detail, identifying critical scheduling decisions and evaluating them against the decisions which would have been made by the experienced human decision maker.

4. An analysis of the implications of such a scheduler with respect to the definition and implementation of a user-defined set of scheduling policies has been made with the goal of providing for the future use of this scheduling paradigm in actual real-time systems.

8.1. Contributions

While process scheduling has been widely considered for many years, long predating computers as we know them today, the computationally tractable scheduling problems have only marginal practical use, while the intractable ones, particularly in distributed environments, have been almost entirely untouched. The computational model used to describe the time-driven scheduling problem for this research bears a strong relationship to the actual situation in many existing real-time systems, so we expect to be able to implement this concept in an actual real-time environment such as ArchOS (the Archons operating system); therefore his work is expected to result in practical approaches to the problem of real-time time-driven scheduling in both a single processor and, by extension, in a distributed environment. The use of continuous value functions to control this scheduling will, we believe, greatly improve our ability to define and implement practical scheduling policies for real-time systems.

Our specific emphasis on real-time time-driven scheduling in the presence of insufficient resources is critical to a successful real-time system. Existing real-time operating systems do not manage critical times at all, but rely on the abundance of resources to ensure that critical times are met. In virtually all such systems, deadlines are occasionally missed, due to such problems as unanticipated load, design error, or partial system failure, and such systems frequently produce anomalous and occasionally disastrous behavior in these situations. Our contribution in this area is expected to lead to a significant improvement in our ability to design, implement, and predict the behavior of such systems.

Relying on resource abundance to ensure predictable, consistent behavior in the presence of heavy load situations has the effect of greatly increasing the cost of real-time systems. There are two parts to the computation of costs in constructing systems; recurring and non-recurring costs. Non-recurring costs (e.g., system architecture, software development, initial test), while important, form only a small part of the total system life-cycle cost. The recurring cost (e.g., hardware production, hardware and software maintenance, spares) is the principal driving factor in this total cost. Designing systems to provide excessive resources (e.g., cpu performance, storage, communications bandwidth) drive up both cost factors, but especially the recurring costs, and any method which will make it possible to moderate the waste of these resources will help keep system costs under control.

One of the reasons for the high cost of real-time systems is the cost of producing and

maintaining the complex software required for such systems. Because of the rather limited capability of existing real-time operating systems to schedule processes to meet time constraints, the burden of managing time constraints has been assumed by the application software, thereby significantly adding to the complexity of the software and the cost of producing and maintaining it. Attendant with this increased complexity is the secondary effect that decisions made at the application level cannot usually take the overall system performance into account, rather they are tailored to the specific part of the application involved, possibly at the expense of the other parts of the application. The use of user-defined policy in the form of value functions allows these scheduling decisions to be made within the operating system, where decisions can be made from the larger system perspective, and removed from the application, providing greater simplicity and robustness during application software development and maintenance.

8.2. Extensions

There are a number of possible extensions to this work which will be the subject of future effort after the completion of this project, both by this author and others. Some of these are:

1. Exploration of the issues of best effort decisions in a decentralized system. This research has concentrated on the scheduling problem at a single node, but it seems clear that this work should be extended to be part of an overall process management capability in a distributed environment. For example, the scheduler defined in this research produces not only a list of processes which are immediately schedulable, but also a list of processes which could be scheduled now if load conditions permitted, but whose value density is too low. These processes could be candidates for process migration, a significant part of a distributed process management capability.

2. The list produced by the scheduler also identifies processes which, although not to be immediately executed, are expected to be executed in the near future, which could be used as part of a heuristic for a virtual memory paging manager. This list of processes which will, and will not, be needing storage in the near future, along with estimates of when the space will be needed could be used to ensure that the needed resources are available at the appropriate times. This could also, perhaps, be applied to other resource management decisions, such as files and communications resources.

3. Although we have informally considered some of the software engineering issues involved in the use of value functions in actual real-time systems development, there is much more to be done here. An extension of this research should include further investigation and experimentation with the separation of value function concerns among the participants in a real-time system project, determining the effects of controlling scheduling policy using value functions at different levels of control.

4. This research has concentrated on the scheduling of processes which are assumed to be ready for execution, and whose value functions are known at scheduling time. Important extensions of this work will involve considering processes whose requests are known to be imminent, such as periodic processes, and explicitly handling precedence and consistency constraints among processes.

References

[Allen 78] A. O. Allen.
Probability, Statistics, and Queueing Theory.
Academic Press, 1978.

[Baker 74] Baker, K. R.
Introduction to Sequencing and Scheduling.
John Wiley & Sons, Inc., 1974.

[Bernstein 71] Bernstein, A. J.; Sharp, J. C.
A Policy-Driven Scheduler for a Time-Sharing System.
Communications of the ACM 14(2):74-78, 1971.

[DeGroot 70] DeGroot, M. H.
Optimal Statistical Decisions.
McGraw-Hill, 1970.

[Erman 80] Erman, L. D.; Hayes-Roth, F.; Lesser, V. R.; Reddy, D. R.
The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty.
ACM Computing Surveys 12(2):213-253, June, 1980.

[Graves 81] Graves, S. C.
A Review of Production Scheduling.
Operations Research 29(4):646-675, July-August, 1981.

[Jeffrey 84] Jeffrey, R. C.
The Logic of Decision, Second Edition.
University of Chicago Press, Chicago and London, 1984.

[Jensen 75] 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.

[Jensen 81] Jensen, E. D.
Distributed Control,
In B. W. Lampson, M. Paul, and H. J. Siegert, *Distributed Systems - Architecture and Implementation*, pages 175-190. Springer-Verlag, 1981.

[Jensen 82] Jensen, E. D.
Decentralized Executive Control of Computers.
In *Proceedings of the Third International Conference on Distributed Computing Systems*, pages 31-35. IEEE, October, 1982.

[Jensen 84] Jensen, E. D.
ArchOS: A Physically Dispersed Operating System.
IEEE Distributed Processing Technical Committee Newsletter, June, 1984.

[Jensen 85] Jensen, E. D.; Locke, C. D.; Tokuda, H.
A Time-Driven Scheduling Model for Real-Time Operating Systems.
In *Proceedings of the Real-Time Systems Symposium*, IEEE, December 3-5, 1985.

[Kain 84] Kain, R. Y.
Performance Bounds on Multiprocessor Schedules.
In *Proceedings of the Seventeenth Annual Hawaii International Conference on System Sciences*, pages 495-503. University of Hawaii and the University of Southwestern Louisiana, 1984.

[Karp 72] Karp, R. M.
Reducibility among Combinatorial Problems,
In Miller, R. E.; Thatcher, J. W., *Complexity of Computer Computations*, pages 397-411. Plenum Press, New York, 1972.

[Knuth 69] Knuth, D. E.
The Art of Computer Programming, Volume 2: Seminumerical Algorithms.
Addison Wesley, 1969.

[Levin 75] Levin, R.; Cohen, E.; Corwin, W.; Pollack, F.; Wulf, W.
Policy/Mechanism Separation in HYDRA.
In *Proceedings of the Fifth Symposium on Operating Systems Principles*, pages 132-140. November, 1975.

[Liu 73] Liu, C. L.; Layland, J. W.
Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.
Journal of the Association for Computing Machinery 20(1):46-61, January, 1973.

[Locke 85] Locke, C. D.
Best-Effort Decision Making for Real-Time Scheduling.
Ph.D. Thesis Proposal, Carnegie-Mellon University, Computer Science Department.

[Mok 78] Mok, A. K.; Dertouzos, M. L.
Multiprocessor Scheduling in a Hard Real-Time Environment.
In *Proceedings of the Seventh Texas Conference on Computing Systems*, pages 5-1 - 5-12. November, 1978.

[Nunnikhoven 77] Nunnikhoven, T. S.; Emmons, H.
Scheduling on Parallel Machines to Minimize Two Criteria Related to Job Tardiness.
American Institute of Industrial Engineers 9:288-296, 1977.

[Pinedo 83] Pinedo, M.
Stochastic Scheduling with Release Dates and Due Dates.
Operations Research 31(3):559-572, May-June, 1983.

[Ruschitzka 77] Ruschitzka, M.
A Unifying Approach to Scheduling.
Communications of the ACM 20(7):469-477, July, 1977.

[Ruschitzka 78] Ruschitzka, M.
An Analytical Treatment of Policy Function Schedulers.
Operations Research 26(5):845-863, September-October, 1978.

[Ruschitzka 82] Ruschitzka, M.
The Performance of Job Classes with Distinct Policy Functions.
Journal of the Association for Computing Machinery 29(2):514-526, April, 1982.

[Sahni 76] Sahni, S. K.
Algorithms for Scheduling Independent Tasks.
Journal of the Association for Computing Machinery 23(1):116-127, January, 1976.

[Sahni 84] Sahni, S.
Scheduling Multipipeline and Multiprocessor Computers.
In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 333-337. IEEE, August, 1984.

[Stankovic 84] Stankovic, J. A.; Ramamritham, K.; Cheng, S.
Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems.
IEEE Transactions on Computers C-34(12), December, 1984.

[Stankovic 85] Stankovic, J. A.
An Application of Bayesian Decision Theory to Decentralized Control of Job Scheduling.
IEEE Transactions on Computers C-34(2):117-130, February, 1985.

[Wendorf 83] Wendorf, J. W.
Concurrency of Operating System and Application Processing.
Ph.D. Thesis Proposal, Carnegie-Mellon University, Computer Science Department.

[Wulf 81] Wulf, W. A.; Levin, R.; Harbison, S. P.
HYDRA/C.mmp: An Experimental Computer System.
McGraw-Hill, Inc., 1981.