Function-Based Scheduling

W. Timothy Strayer

Computer Science Report No. TR-91-35
December 6, 1991

Function-Based Scheduling

W. Timothy Strayer

Department of Computer Science
University of Virginia
Charlottesville, Virginia
wts4x@virginia.edu

1. Introduction

Scheduling theory maintains that there are fundamental similarities in task sequencing problems which transcend the character of the particular tasks to be ordered or the resources to be used. Tasks arrive, require some amount of resources, and depart. A *scheduling policy* is chosen for a system such that tasks within the system are serviced at particular times for particular durations. A *scheduling algorithm* implements the scheduling policy by specifying the method for sequencing the tasks. An algorithm applied to a particular task set produces a *schedule*; scheduling theory studies how we can analyze these scheduling algorithms.

One of the objects of analyzing scheduling algorithms is to determine if the algorithm will always meet the constraints of the task set. Often this question is too difficult to answer for a general task set; if so, the task set is typically restricted. Unfortunately scheduling algorithms quickly become non-polynomial in their complexity if these restrictions on task constraints are relaxed. Consequently, traditional scheduling algorithms tend to rely on one or a few specific task attributes or system characteristics for establishing a ranking criteria. To simplify the scheduling algorithm and therefore its analysis, all tasks are typically ranked according to the same criteria.

1.1. Function-Based Scheduling

Consider a model for scheduling tasks where each task has an associated function which profiles the task's importance over time. Assume that all of these functions are normalized so that comparing the values of any two tasks' functions at a particular point in time would indicate which of the two tasks is more important to the system. At any point in time the tasks in the system can be ranked according to the values of their functions, and thus according to their importance to the system at that moment in time.

If tasks were assigned functions which profiled their importance to meeting the system goal, and the system ensured that at every point in time the most important task in the system was receiving service, then the resulting schedule of the tasks would necessarily be the *optimal schedule*. Stated alternatively, the system would always be doing the best it could under the circumstances. Note that this does not imply that the schedule will provide all of the work required by all of the tasks, or that the schedule will meet the goal of the system; however, it does state that if the goal can be met, this schedule will meet it. These functions associated with each task we term *importance functions*, and the scheduling model within which these functions serve we term the *importance abstraction*.

There is a spectrum of forms for expressing scheduling policies. Traditionally the scheduling policies are expressed using algorithms. The importance abstraction represents another point in this spectrum, where the "algorithm" is simplified and universal, and expressiveness derives from using functions to describe particular scheduling policies. While these two methods are of equal power in terms of the policies which can be expressed, the importance abstraction offers a straightforward approach to implementing a scheduling policy. Importance functions are particularly expressive, as each task is described by a function tailored to that task. These functions include as parameters those attributes and characteristics upon which the task's importance to the system is based.

Consequently, schedules are produced by considering what conditions make an individual task important rather than trying to find an algorithm whose criteria fits all cases for a task set. Since the principle component of the importance abstraction is a set of functions rather than an algorithm, the analysis of the scheduling of tasks benefits from the maturity of the analysis of functions.

1.2. Issues

There are three issues to be addressed concerning the usefulness of the importance abstraction—expressiveness, analyzability, and implementability.

The first issue considers expressiveness. The importance abstraction can emulate the “traditional” scheduling algorithms by creating functions such that the schedule for a given task set produced by the importance abstraction is identical to that produced by the algorithm. Consider the nearest deadline first algorithm, for example. For any two tasks *i* and *j* active at time *t*, with deadlines d_i and d_j, the importance functions $I_i(t)$ and $I_j(t)$ are constructed such that

$$d_i < d_j \Rightarrow I_i(t) > I_j(t) \qquad \text{(Eq 1)}$$

Since algorithms choose tasks according to specific conditions, we may, in general, construct a set of importance functions for a given task set such that a task within that set will become most important precisely when the algorithm would choose that task for service.

The concept of profiling a task’s importance over time is intuitive, and using functions to express this importance is more “natural” than using equivalent algorithms. By using functions we can more easily encompass a wider range of parameters than can an algorithm, and we can more easily tailor importance profiles to individual tasks. These may

be possible in an algorithmic domain, but designing an algorithm to achieve this generality may prove more cumbersome than a function-based approach.

The second issue considers the analyzability of the importance abstraction. By expressing the scheduling problem in a functional domain rather than an algorithmic one we gain the tools supplied by mathematics. We may invoke proof techniques that are more extensive and mature than those used in proving properties of algorithms. If we can phrase our scheduling questions in terms of functional mathematics, then we may apply the machinery of the mathematics to help answer these questions. Scheduling algorithms may be as expressive as using the importance abstraction, but as the complexity of the algorithm increases, the analysis of the algorithm becomes exceedingly difficult. Very complex and subtle scheduling problems can be expressed using importance functions and, since we are dealing with functions, functional mathematics can be applied to help provide analysis.

The third issue considers the implementability of such a framework within a real system. Clearly it is impossible to evaluate each task's function at every moment in time to ensure that the most important task can be identified. However, it is sufficient to ensure that the most important task can be identified at every point in time; the importance functions require evaluation only when a new most important task must be chosen. If it is possible to identify when the evaluations must take place, it may be possible to implement this scheme in a cost-efficient manner. It may be the case that some restrictions must be placed on the functions so that the scheme may be efficiently implementable. If this is so, it is important to discover how such restrictions affect the expressiveness of the importance abstraction.

1.3. Contribution

This report introduces the *importance abstraction* as a framework for implementing scheduling policies. The importance abstraction is a function-based approach for

describing the scheduling policy. There have been previous function-based approaches, in particular those of [BERN71], [RUSC77] and [JENS85]; the importance abstraction extends this previous work and makes contributions to scheduling theory in several ways:

1. the importance abstraction provides a universal framework for the expression of traditional and novel scheduling policies
2. the importance abstraction employs a standard and very simple scheduling algorithm, thus shifting the burden of expression of the scheduling problem from the algorithm to the function which represent the tasks
3. since the scheduling problem is expressed in terms of a set of functions, analysis using mathematical techniques is appropriate
4. while conceived primarily as a theoretical tool for modeling scheduling policies, several aspects of the importance abstraction suggest that an efficient implementation of a scheduler based on functions is possible for a wide class of policies.

2. Background

Rate monotonic theory ([LIU73]) provides rich analytical results for scheduling algorithms designed to meet deadline for a periodic task set. Included in these results are a simple tests for feasibility of a task set based on its aggregate utilization of the processor. Much work in real-time scheduling centers around this algorithm, both in exploiting the results and in seeking solutions to the various deficiencies to the basic algorithm. We present a brief overview, first because rate monotonic theory is pervasive within scheduling theory, and second because we revisit some of these results in our analysis section.

The importance abstraction has a function-based scheduling framework. We survey seminal work in using functions to aid in scheduling decisions. Typically the functions return a value which represents some aspect of the task's worth, such as *priority* and *value* (the importance functions return values which represent *importance* of a task). Functions provide flexibility in expressing this task's worth over the time that the task is active. The importance abstraction extends this work by also using the functional representation of the task's importance to perform analysis on the nature of the schedules produced.

2.1. Rate Monotonic Theory

In 1973, Liu and Layland introduced *rate monotonic* scheduling theory ([LIU73]) as a method for scheduling many periodic tasks on a single processor such that the scheduling algorithm used to do this was optimal. Dhall and Liu extended this work into the multiprocessor environment in [DHAL78]. The following discussion is drawn largely from Sha and Goodenough ([SHA90]), who present an excellent overview of the theory and recent extensions which include aperiodic and sporadic tasks, as well as non-independent task relationships.

Rate monotonic scheduling theory in essence ensures that as long as the processor utilization of all tasks lies below a certain bound and appropriate scheduling algorithms are used, all tasks will meet their deadlines without the programmer knowing exactly when any given task will be running. Given a set of independent periodic tasks which are preemptable, the rate monotonic scheduling algorithm gives each task a fixed priority and assigns higher priorities to tasks with shorter periods. All tasks are *preemptable* in that whenever there is a request for a task that is of higher priority than the one currently being executed, the running task is immediately interrupted and the newly requested task is started. A task set is said to be *schedulable* if all its deadlines are met (i.e., all periodic tasks finish execution before the end of their periods).

Any independent periodic task set may be subjected to a test to determine if that task set is schedulable regardless of when each individual task is started ([LIU73]). Let C_i be the execution time for task τ_i, T_i be the period for task τ_i and n be the cardinality of the task set. For a statically assigned priority algorithm (the rate-monotonic algorithm, where priority is defined as the inverse of the task period), the following must be true for the task set T to be feasible:

$$\frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} \leq n\,(2^{1/n} - 1) \qquad \text{(Eq 2)}$$

If the utilization (computation time over period) of all of the tasks is below the bound prescribed, then the tasks are guaranteed to be schedulable if they are scheduled according to the rate monotonic algorithm. This bound converges to *ln* 2, or about 70% utilization of processor capacity, as the number of tasks goes to infinity. This algorithm is shown to be optimal among all fixed priority scheduling algorithms on periodic task sets.

Liu and Layland also show that a variation on this, the deadline driven scheduling algorithm, can provide 100% processor utilization on task sets where the priority can be assigned dynamically. This variation is also optimal among all algorithms where priority assignment may be made during the run of the system. In the deadline driven scheduling algorithm, the priorities are assigned according to which task's deadline is nearest rather than by period length.

In [SHA90], Sha and Goodenough discuss the use of rate monotonic theory for real-time scheduling in the Ada tasking model. However, there are certain drawbacks to the unabridged rate monotonic scheduling policy, namely that (1) a task's period is not inherently related to how critical it is to the system, even though priority is assigned by period length; (2) synchronization of a lower priority can indefinitely delay a higher priority task when tasks share data or communications; and (3) there is no clear way to treat aperiodic tasks in this policy designed for periodic task sets. *Period transformation*, *priority inheritance* and *priority ceiling protocols*, and the *deferrable server protocol* address each of these issues respectively.

2.1.1. Period Transformation

One major problem with rate monotonic scheduling is that the priorities are assigned according to the period of the task rather than according to its criticality to the system. When all tasks can be scheduled without fear of some task exceeding its execution time, then no criticality measure need be placed on the tasks. However, execution times are necessarily stochastic, and scheduling is usually done with worst-case estimates which may be significantly longer than the average execution time. When tasks exceed their estimated execution times, a transient overload occurs which may cause some tasks to miss their deadlines. Yet if tasks are prioritized according to their periods, some critical tasks may miss their deadlines if their periods are too long.

The *period transformation* technique ([SHA86]) is used to ensure that highly critical tasks are treated with higher priorities even if they have longer periods. The priority of a critical task can be raised by treating it like a task with a shorter period. The technique is to divide both the period and the worst-case execution time by some constant. Now the task looks like its period is shorter, but the total utilization is not affected. Execution is suspended after each execution time until the next "period" arrives.

This technique is designed, therefore, to decouple the criticality of a task from its period, while maintaining the benefits of the rate monotonic algorithm. If the tasks can be partitioned into critical and non-critical task sets, where the critical tasks are defined to be those which must receive service during a transient overload condition, then a period transformation can be applied to the critical tasks with the longest periods. Without period transformation, the longest period tasks would be subject to missed deadlines since they have low priority. The set of critical tasks, therefore, are period transformed until the longest period of the critical set is shorter than the shortest period of the non-critical set. Now all non-critical tasks will miss a deadline before the first critical task will.

2.1.2. Priority Inheritance and Priority Ceiling Protocols

Priority inversion is defined as the phenomena of a task of higher priority being forced to wait on the completion of a task of lower priority. In certain cases the priority inversion can be unbounded. The *priority inheritance protocol* attempts to limit the amount of priority inversion by allowing a server task to inherit the priority of its highest priority *client* ([SHA87]). A central theorem in priority inheritance specifies a sufficient worst-case condition that characterizes the rate-monotonic schedulability of a given set of periodic tasks. The *priority ceiling protocol* minimizes the blocking of high priority tasks by guaranteeing that such a task will be blocked by at most one critical region of any lower priority task ([GOOD88], [LOCK88]).

The *priority ceiling* of a critical region is defined to be the highest priority of all the tasks that may lock on that region. When a new task attempts to secure that region, it will be suspended unless its priority is higher than the priority ceilings of all regions currently locked by tasks other than this one. If the task is suspended, then the task that holds the lock on the region with the highest priority ceiling is said to be blocking this task, and hence inherits the priority of this task.

2.1.3. Deferrable Server

Current systems with hard real-time periodic tasks handle aperiodic tasks either by servicing them in background or by polling periodically for aperiodic tasks. If an aperiodic task is serviced in the background, it must wait until all periodic tasks have been serviced. If an aperiodic task arrives just after the polling time, the task must wait until the next polling time. In both of these cases the response time for aperiodic tasks suffers unnecessarily due to naive treatment of the task set.

The Deferrable Server algorithm ([LEHO87], [SPRU88]) is designed to provide aperiodic tasks with a low response time without jeopardizing the periodic tasks. A new periodic task with highest priority is created to service the aperiodic tasks such that all tasks, including this aperiodic server, are guaranteed to meet their deadlines by the rate monotonic theory. Any aperiodic tasks are serviced at this highest priority as soon as they arrive as long as there is computation time left for this aperiodic server. When there are no aperiodic tasks, the computation time of the server is deferred until one arrives. The computation time of this server is replenished each period. Thus, the response time for aperiodic tasks is minimized while the schedulability of the hard real-time periodic tasks is maintained.

2.2. Survey of Function-Based Scheduling Techniques

Bernstein and Sharp ([BERN71]) recognized that service given to a class of tasks could be controlled using a function such that various service profiles could be effected as the tasks grew older. Priority in this scheme was related to the difference between the function's projected service and the service actually attained. Ruschitzka and Fabry ([RUSC77]) used functions to describe the priority of a task directly. Within this model, various scheduling algorithms could be emulated by using an appropriate priority function. Jensen *et al* ([JENS85]) used a function to profile a task's value to the system for completing at that time. The value functions did not directly drive the scheduling decisions in Jensen's model; rather they were used mostly as a metric for comparing the performance of other scheduling algorithms.

Below we survey these three techniques for using functions for making scheduling decisions.

2.2.1. Policy Functions

Bernstein and Sharp, in [BERN71], theorized that a scheduling algorithm which keeps track of the resource count of each task and orders the tasks according to how far a task is from the expected resource count at that task's age would provide the specified level of service for each task. They defined a *policy function* as a function which characterizes a class of tasks by specifying the amount of service those tasks should receive as a function of time. Each class of tasks within a system is characterized by a function that specifies the amount of service a task within a class should receive as a function of time. The shape of the policy function will control the type of service received by that class of tasks. The notion of priority corresponds to the difference between the service promised to the task by the policy function and the service actually received by the task. Consequently, the priority of a task changes at a constant rate while awaiting service and at another rate determined by the shape of the policy function while receiving service. The tasks which are most delinquent are therefore the highest priority tasks.

Since the shape of the policy function ultimately determines a task's priority, different scheduling policies may be implemented using the same basic scheduling algorithm by simply changing the policy functions of the tasks. Bernstein and Sharp consider piecewise functions as the policy functions for various classes of tasks. One such function proposed uses a curved portion in a region starting with the task's activation to give a task a limited amount of rapid service, followed by a linear portion for a more constant rate of service.

Ruschitzka and Fabry ([RUSC77]) extend the notion using functions for scheduling by introducing the *universal scheduling system* (USS) as a generalized scheduling framework to support arbitrary scheduling algorithms. There are three parts to the specification of a scheduling algorithm within the USS: the *decision mode*, the *priority*

function, and an *arbitration rule*. The *decision mode* specifies how often scheduling decisions are made. The *priority function* is an arbitrary function of task and system parameters that determine the task's priority at the time of evaluation. The *arbitration rule* is used to break ties between tasks of the same priority. Ruschitzka and Fabry suggest that a scheduling algorithm can be emulated by appropriately specifying the decision mode, priority function, and arbitration rule such that the USS will make exactly the same scheduling decision at exactly the same time as would the algorithm.

Four decision modes are identified; each decision mode is progressively more general than the last, and each previous mode is a proper subset of the next. The four modes are *nonpreemptive*, *quantum-oriented*, *preemptive*, and *processor sharing*. The *nonpreemptive* mode allows decisions only after the task currently being serviced is complete or when an arriving task finds the server empty. The *quantum-oriented* decision mode makes decisions every fixed quantum of time unless a task completes or a newly arriving task finds the server empty before the time of the next decision. If the quantum is infinitely long, then the quantum-oriented mode reverts to the nonpreemptive mode. If decisions are also made when any new task arrives, regardless of whether the server is busy, then the decision mode is *preemptive*. Allowing the quantum size to become zero, and hence ensuring that decisions are made continuously, defines the decision mode called *processor sharing*. Processor sharing is the most general decision mode, encompassing each of the other decision modes—the instant one task becomes higher priority than a task being serviced, the new higher priority task takes its place in the server. This mode is called processor sharing since the group of highest priority tasks, while they all remain highest priority, will effectively share the server (processor) even if there are more tasks to service than the server can handle; each task is receiving service concurrently with all other equally highest priority tasks.

The priority function is an arbitrary function of task and system parameters. The priority of a given task is defined as the value of the priority function applied to the current values of the parameters. Ruschitzka and Fabry suggest that these parameters may include the memory requirements, the attained service time, the total service time, external priorities, timeliness, and system load. A priority function is defined for a scheduling algorithm such that, when the algorithm chooses a particular task for service, the priority function applied to that task will return the highest priority among all tasks.

The arbitration rule specifies how to resolve conflicts among jobs with equal highest priority. Ruschitzka and Fabry note that the advantage to specifying the arbitration rule, as well as the decision mode, is that this specification simplifies the priority function. Neither the decision mode nor arbitration rule is necessary since the priority function can be made to implement the various decision modes and arbitration rules.

Ruschitzka and Fabry continue by noting that a large class of scheduling algorithms can be defined by a priority function of only three arguments: the task's attained processing time, the current time, and the task's processing time requirement. Furthermore, an algorithm is called *time-invariant* if the difference between the priorities of any two tasks does not change as long as neither task receives service. Included in this class of algorithms is the policy-driven scheduling algorithms of [BERN71]. Ruschitzka and Fabry extend the work of Bernstein and Sharp by noting that, in general, time-invariant priorities are characterized by a policy function of an arbitrary number of arguments.

2.2.2. Time-Driven Scheduling

The primary notion in *time-driven scheduling* ([JENS85], [LOCK86], [TOKU87], [WEND88]) is that the distinguishing characteristic of a real-time system is the concept that the value a task has to the system is dependent upon when that task completes. Each task

has associated with it a *value function* $V_i(t)$ which returns the value to the system for completing task i at time t. The optimal schedule, therefore, arranges the tasks such that they complete at times which maximize the sum of their values to the system. Jensen *et al* use this value sum as a metric for comparing the effectiveness of conventional scheduling algorithms.

It was observed in [JENS85] that task scheduling in real-time systems almost always uses some simple algorithm, like fixed priority, first in first out, or round robin. Often the time-criticalness of the tasks is represented by a point in time called a *deadline*. Attempting to meet deadlines via fixed priority scheduling algorithms leads to rounds of testing and adjustment of the priorities, and results in a particularly fragile system. Assigning higher priorities to important tasks does not reflect the time-constrained characteristic of the tasks. Assigning higher priorities to tasks with nearer deadlines does not reflect the differences in importance among tasks.

The tasks with associated value functions do not employ the explicit use of a deadline. Rather, the existence and importance of deadlines depend on the nature of the value function. A *critical time* for a task is represented by a discontinuity in the task's value function. In this way the concept of hard and soft deadlines is replaced by a step function whose shape reflects the urgency of completing before a certain time.

Jensen *et al* create an environment in which various scheduling algorithms can be evaluated through the use of a simulator. For tractability reasons the value functions are limited to having two parts, one prior to and one after the critical time, each consisting of the following five-constant form:

$$V_i(t) = K_1 + K_2 t - K_3 t^2 + K_4 e^{-K_5 t} \qquad \text{(Eq 3)}$$

This form allow value functions which are constant, linear, quadratic, exponential, or a linear combination of any of these.

Jensen *et al* report the simulation of several classical algorithms on a task set using various shapes for the value functions. These algorithms included shortest estimated completion time first, earliest deadline first, least slack time first, first in first out, random order, and a fixed priority where the priority was equal to the highest value that the value function could attain. Two experimental algorithms were also evaluated. The first used a *value density* (value at the projected completion time over the task length). The second algorithm used a nearest deadline first algorithm, shedding the tasks with the lowest value densities during overload. Four shapes of value functions, shown in Figure 1, were used in separate executions to compute the total value generated by each of the scheduling algorithms. The results showed that the second experimental algorithm outperformed all others tested; this algorithm, called the Best-Effort Heuristic, is the focus of Locke's work in [LOCK86].

The implementation issues of time-driven scheduling, especially using the best-effort heuristic, are explored in [TOKU87] and [WEND88]. It was concluded that the high computational overhead of best-effort time-driven scheduling made implementation impractical on a uniprocessor system. More reasonable performance could be gained by using a dedicated processor for only scheduling decisions.

3. Importance Abstraction

The *importance abstraction* is a framework within which we can describe scheduling policies by focusing on the importances of the tasks within a system. The system has a goal and the tasks within the system are processed with the intent of meeting the system goal. A task within the system is viewed as "important" to the system *vis-a-vis*

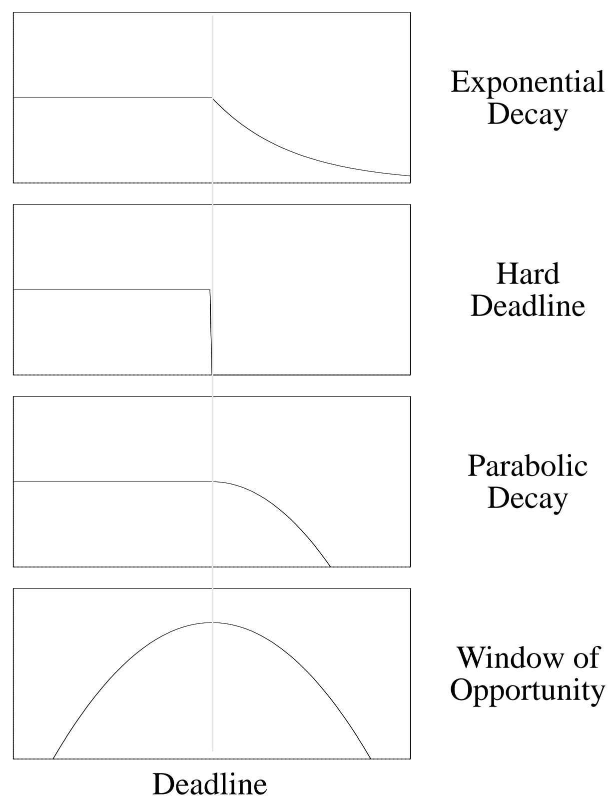

Figure 1 —Value Function Shapes

how that task can contribute to accomplishing the system goal. As the system progresses and its state changes, various tasks become more or less important to the system. The importance abstraction is a framework for expressing those conditions under which tasks within a system become important to the system.

The importance abstraction includes within its framework sets of importance functions that describe the tasks within a system, and a scheduler which uses the importance functions to determine which tasks should receive service. By using this abstraction to consider scheduling problems, we shift the emphasis from the analysis of the scheduling algorithm to the analysis of a set of functions.

3.1. System Model

We define a *system* as any entity with the following components: a set of inputs into and a set of outputs from this entity, a processor, and a set of tasks to be processed, as shown in Figure 2. The system “communicates” with the world outside of it through its inputs and outputs. The system reacts to inputs by changing the system state. The outputs from the system reflect, to some extent, these and other state changes, and are the means by which the system may affect the outside world. Since a system is designed to accomplish some goal, it is only through these outputs that the degree to which the goal is accomplished can be gauged by an outside observer. The system makes choices about when and what tasks to process such that the system can move toward accomplishing its goal.

Since the system is designed to accomplish some goal, each task within the system somehow contributes to accomplishing the system’s goal under system conditions and task attributes which are specific to that task. As these particular conditions arise within the system, the task becomes “important” to the system. At any particular point in time there exists a ranking of the tasks according to how important each task is to accomplishing the system goal. If, at that moment, a “most important” task exists, then the system could best move toward accomplishing the goal by performing that task at that moment. As conditions

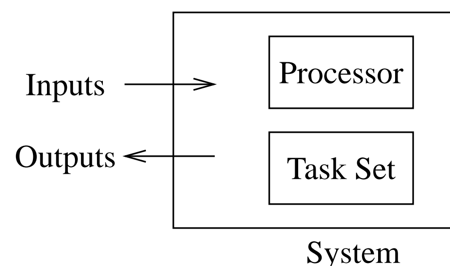

Figure 2 — The System Model

change, the importances of the tasks may change, and a new task may become “most important.”

Just as the state of the system changes with time as work is performed on the tasks and inputs are received, the composition of the task set also changes with time. At the system start time, when the system begins at some initial state, there exists an initial task set. As work is done on tasks within this set, the attributes of the tasks within the set change—in particular, the “work remaining” attribute of the task currently receiving service is decreasing. The membership of the task set also changes over time. Some tasks complete and are removed from the task set. Other tasks may simply outlive their usefulness and be removed from the task set. Still others arrive and join the task set. Consequently, we can think of a snapshot of the task set as being a “state,” and the act of scheduling and servicing the tasks within the task set moves the task set from one state to another.

3.2. Importance Functions

If the importance of a task could be quantified, it could be expressed as a function over time to profile a task’s importance to the system. Since the importance of a task depends upon the conditions of the system and the attributes of that task, these conditions and attributes must be the parameters to the function. If we can identify each possible task in the system, and under what conditions that task will become important to the system, we could associate with each task a function that reflects the task’s importance.

Consider a task set where each task in the set has associated with it a function, called an *importance function*, which includes all of the conditions and circumstances under which the task is important to the system. Let the function return a value which ranks that task among all other active tasks competing for a system resource according to how

important it is that the task be given that resource at that moment in time. The importance abstraction uses sets of importance functions as a representation of the task set with respect to how the tasks within the task set should be ordered for service in order to accomplish some system goal.

3.2.1. Sets of Importance Functions

Assume that, for a given system, there is an importance function associated with each task in the task set. Let I_T be a set of importance functions for the task set T. The set I_T embodies those attributes and constraints of the tasks in T and system parameters considered important to accomplish a particular goal; therefore, we can consider the set of importance functions as representatives of the tasks, and use these functions when asking questions regarding scheduling.

We can consider a *universe* of all sets of importance functions for the task set T, $U = \{I_T\}$, where each member of the universe imposes a schedule which will meet some particular system goal. Not every member of the universe of importance function sets will meet the same system goal; rather, it is the system goal which partitions the universe into two sets: those sets of importance functions which impose schedules which meet the system goal, and those importance function sets for which the system goal cannot be guaranteed. Thus, given a goal G, the universe U can be partitioned into $U_G = \{I_T \ni G \text{ is satisfied}\}$ and $U_{\overline{G}} = \{I_T \ni G \text{ is not guaranteed}\}$.

3.2.2. The Defining Property of an Importance Function Set

Given a task set T within a particular system, and a goal G for that system, we seek the property P_G which defines the set U_G. We call this property a *defining property.* Since the goal G partitions the universe of importance function sets U into U_G and $U_{\overline{G}}$, the

defining property P_G reflects those qualities of the sets of U_G which (1) make each set a member of U_G, and (2) distinguishes each set from sets in $U_{\overline{G}}$.

Since each importance function set in U_G imposes a schedule which meets the goal G, the schedules are termed *equivalent*. The schedules, and indeed the importance function sets which impose them, are therefore members of an equivalence class. By discovering the defining property of an importance function set which causes that set to belong to U_G, we can determine if a given importance function set is a member of this equivalence class.

If the defining property holds for every member of the equivalence class and no others, that defining property represents the *necessary* and *sufficient* conditions on the set of importance functions for inclusion in the equivalence class. If a property holds for a subset of the equivalence class and no others, then the property is a sufficient condition for inclusion in the equivalence class, but not a necessary condition.

3.3. The Scheduler

When a set of importance functions has been associated with a task set, the tasks within that task set are scheduled according to the values of the importance functions. By definition, the *optimal schedule* is achieved when the scheduler chooses the most important task (task with the highest valued importance function) at every point in time. Thus, at every point in time the scheduler must evaluate the function $M{:}\{I_T\} \rightarrow T$, which takes the set of importance functions and returns a task. Without loss of generality assume that the tasks in T are indexed, in no particular order, so that a task is identified by its index. The function M evaluates each importance function in the set I_T and returns the task $i \in T$ whose importance function has the maximum value at that point in time. If, at some point in time, the scheduler finds that two or more tasks are most important simultaneously, the scheduler

will arbitrarily choose one of those tasks as the task to receive service, and will continue to allow that task to receive service until some other task becomes most important.

We can express the actions of the scheduler with some mathematical constructs. The boolean relation $(M(I_T) = i)$ returns the value 1 if the most important task at the time of evaluation is the task i, and the value 0 otherwise.[1] By using this boolean relation as a function of time, we can ask how long a specific task has been most important over a certain period. Let the value $w_i\big|_{t_1}^{t_2}$ represent the amount of work applied to the task i from time t_1 to time t_2 (Appendix A describes in detail the properties of this construct). The equation

$$w_i\big|_{t_1}^{t_2} = \int_{t_1}^{t_2} (M(I_T) = i)\, dt \qquad \text{(Eq 4)}$$

shows the relationship between the importance functions and the amount of work done to a particular task. This equation states that the amount of work received by task i over the period from t_1 to t_2 is equal to the amount of time that the task i has been most important from time t_1 to t_2. Note that if there are two or more tasks with equal importance at time t, the function M chooses one of these tasks arbitrarily.

4. Expressiveness of the Importance Abstraction

The importance abstraction is a framework for expressing scheduling policies. The actual scheduling algorithm is simple and universal: the scheduler chooses the most important task at every point in time. The most important task is found by evaluating the set of importance functions which profile the importance of each task. By using a function to profile the task importance, the scheduler considers the conditions under which an individual task becomes important without the scheduler or the scheduling algorithm

---

[1]. The convention of using a boolean expression within a set of parentheses to denote a function that returns 1 if the boolean expression is true and 0 if it is false is used in Graham, Knuth, and Patashnik's book **Concrete Mathematics** (1988); they attribute the convention to Iverson in the programming language APL.

maintaining the state of these conditions for each task. This shifts the description of the conditions for scheduling from the scheduler to the agents for the tasks, allowing the scheduler to remain simple and universal. Consequently, complex scheduling policies (e.g., those with many conditions on which task is to be scheduled at any particular time) are easily expressed in the importance abstraction while the same policies may prove difficult and cumbersome to express as algorithms.

Traditional scheduling policies are typically based on one or only a few task attributes. Consequently, the algorithms which implement these policies use these attributes when determining the schedule. For example, the nearest deadline first scheduling policy considers only task deadlines; the algorithm dictates that the task with the nearest deadline is always scheduled for service. These scheduling policies can be also be implemented within the importance abstraction by devising importance functions based on the task attributes considered by the algorithms. The importance functions emulate the algorithm if a task becomes most important exactly when the task would be scheduled by the algorithm. In this section we give several examples of traditional scheduling policies and show importance function sets that implement the policies by emulating the algorithms associated with the policies.

In addition to its ability to emulate the traditional scheduling algorithms, the importance abstraction provides the framework for implementing novel scheduling policies not intuitive when using algorithms. We offer an example of such a novel scheduling policy, and show how the policy can be expressed easily with importance functions.

4.1. Emulation of Traditional Scheduling Policies

An interesting and important aspect of the importance abstraction is the ability to *emulate* traditional scheduling policies within its framework. The importance abstraction is

said to emulate an arbitrary scheduling policy in that it makes exactly the same scheduling decisions at exactly the same time.[2] In the importance abstraction the act of scheduling always remains the same: choose the most important task to perform at each decision point; the various scheduling policies are actually implemented by defining appropriate importance functions. The importance functions must be defined in such a way that a task becomes most important at precisely the same instant as the conventional scheduling policy would have chosen it.

First Come First Served

In the First Come First Served (FCFS) scheduling policy, the scheduler chooses the oldest of the active tasks to serve. That is, it finds the min(a_i), where a_i is the arrival time for the task *i*. To emulate this policy, we define importance functions for each task such that the task's importance is monotonically increasing with its age. There is an infinite class of importance functions for which this is true; we offer the most obvious:

$$\forall\,(i \in T)\,, \qquad I_i(t) \;=\; \begin{cases} 0,\ \text{if}\ (t < a_i) \\ t - a_i,\ \text{if}\ (t \ge a_i) \end{cases} \tag{Eq 5}$$

Consider four tasks with arrival times as follows:

$$\begin{aligned} \text{task 1} &: a_1 = 0 \\ \text{task 2} &: a_2 = 2 \\ \text{task 3} &: a_3 = 3 \\ \text{task 4} &: a_4 = 4 \end{aligned} \tag{Eq 6}$$

Let each task be associated with an importance function as defined above. Assume that each task requires 3 time units to finish. Figure 3 shows the graph of importance value versus

[2]. The concept of creating a framework within which to emulate other scheduling policies was first presented by Ruschitska and Fabry in [RUSC77] with the Universal Scheduling System.

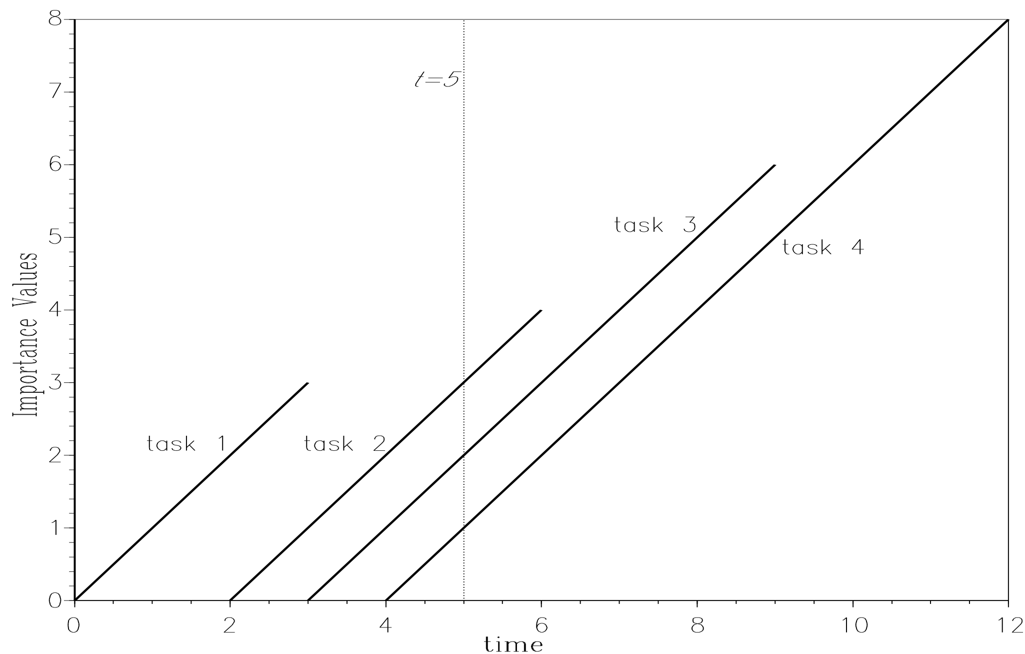

Figure 3 — Importance Function Values for FCFS Policy

time as each task gets older. Notice at time 5 there are 3 active tasks. The scheduler will always choose $M(I_T)$ which, for time 5, is $M(I_2(5)=3, I_3(5)=2, I_4(5)=1) = 2$, so task 2 is chosen.

Nearest Deadline First

Nearest Deadline First (NDF) is quite similar to FCFS in that we need a monotonically increasing function based on the nearness of a point in time; while FCFS uses arrival times, NDF uses deadlines as the basis for the importance functions. If the scheduler could chose the minimum of some set of values rather than the maximum, we could use the quantity $d_i - t$, where d_i is the task's deadline. Since the scheduler always chooses the most important, then we need a function which is monotonically increasing: the reciprocal of $d_i - t$ is such a function.

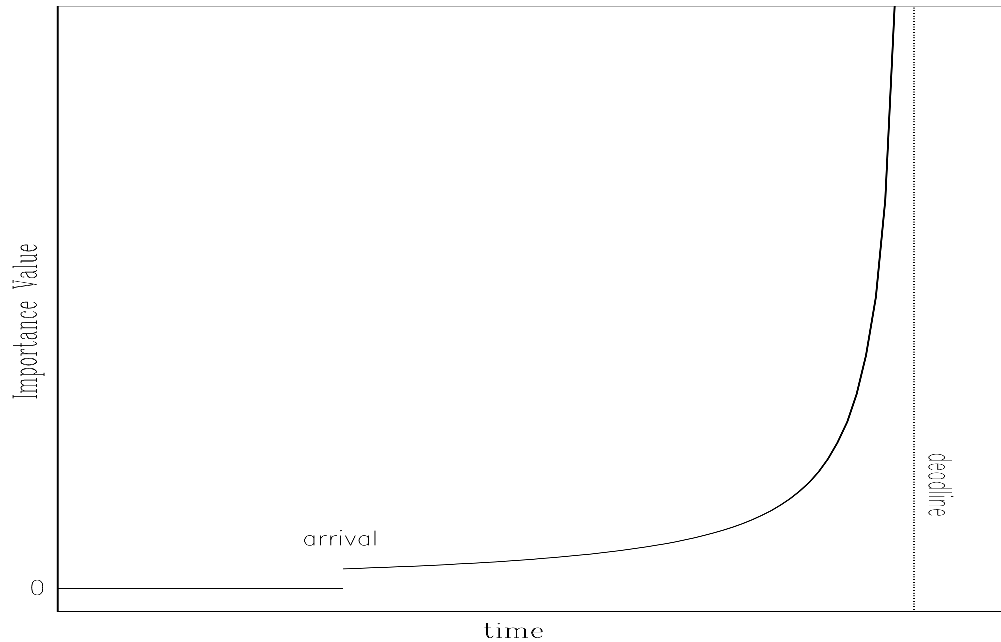

Figure 4 — Importance Function for an Nearest Deadline Task

Consider the following importance function definition for each task:

$$\forall (i \in T), \qquad I_i(t) = \begin{cases} 0, \text{ if } (t < a_i) \\ (d_i - t)^{-1}, \text{ if } (d_i > t \geq a_i) \\ 0, \text{ if } (d_i \leq t) \end{cases} \qquad \text{(Eq 7)}$$

Further, consider a task which arrives at time 3 and has a deadline of time 10. Figure 4 shows the graph of the importance values over time for this task. Notice that there are two discontinuities, one at the moment that the task becomes active and one at the moment it misses its deadline. Also notice that the task becomes infinitely important just as the deadline is reached.

Priority Driven

Tasks which are ranked by static priority are easily emulated by the importance abstraction by defining the importance functions as constant functions reflecting the relative ranking of the tasks. Any constant values will work as long as, for any two tasks *i* and *j* with priorities p_i and p_j, priorities equal to or greater than 0, the following always holds:

$$p_i > p_j \Rightarrow I_i(t) > I_j(t) \quad \text{(Eq 8)}$$

An example of such a function is:

$$\forall (i \in T), \qquad I_i(t) = \begin{cases} 0, \text{ if } (t < a_i) \\ p_i, \text{ if } (t \geq a_i) \end{cases} \quad \text{(Eq 9)}$$

Rate Monotonic

Rate monotonic theory applies to those tasks which are periodic in nature; that is, an instantiation of the task is activated exactly once per time period.The priority of a task is statically assigned to be the inverse of that task's period, T_i. An importance function set which emulates this is:

$$\forall (i \in T), \qquad I_i(t) = \begin{cases} \frac{1}{T_i}, \text{ if } (a_i \leq t \leq a_i + T_i) \\ 0, \text{ otherwise} \end{cases} \quad \text{(Eq 10)}$$

Least Slack Time

The least slack time policy chooses the task which has the least difference between its projected finish time and its deadline. Previously we have considered a deadline as the time by which the task must start. Here the deadline is the time by which the task must

finish. Slack time for task i is defined as $slack = d_i - l_i - t$, where l_i is the task length. The importance functions are easily given by replacing the d_i quantity in the deadline driven functions by the quantity $d_i - l_i$, thus:

$$\forall (i \in T), \qquad I_i(t) = \begin{cases} 0, \text{ if } (t < a_i) \\ (d_i - l_i - t)^{-1}, \text{ if } (d_i - l_i > t \geq a_i) \\ 0, \text{ if } (d_i - l_i \leq t) \end{cases} \qquad \text{(Eq 11)}$$

Round Robin

In round robin scheduling each of the n tasks is given an equal share of the processor in turn until all n tasks have received a share. Although the order of service is arbitrary, once established, the order is maintained for subsequent cycles through the task set until one or more tasks complete or one or more tasks join. In general the share of the processor, or *time slice*, may either be fixed, and hence the period of the cycle varies with the size of the task set, or the period itself is fixed, and hence the time slices vary according to the set size.

Consider a set of importance functions which take the form

$$I_i(t) = \sin(bt + c_i) + d \qquad \text{(Eq 12)}$$

where b determines the period, c_i is the offset for task i, and d, if greater than 1, shifts the function so that all values are positive. Let $d = 1$ and, for n tasks numbered 0 through $n-1$, let $c_i = (2\pi i)/n$. Figure 5 shows graphs of importance functions for four tasks. It is easily seen that each task is "most important" for precisely $1/n^{th}$ of the period, and that the order of service remains fixed.

4.2. Families of Importance Functions

Often it is instructive to show a function in its general form, as with lines ($y = mx+b$) and circles ($x^2+y^2 = b^2$). Since the importance abstraction is based on sets of functions,

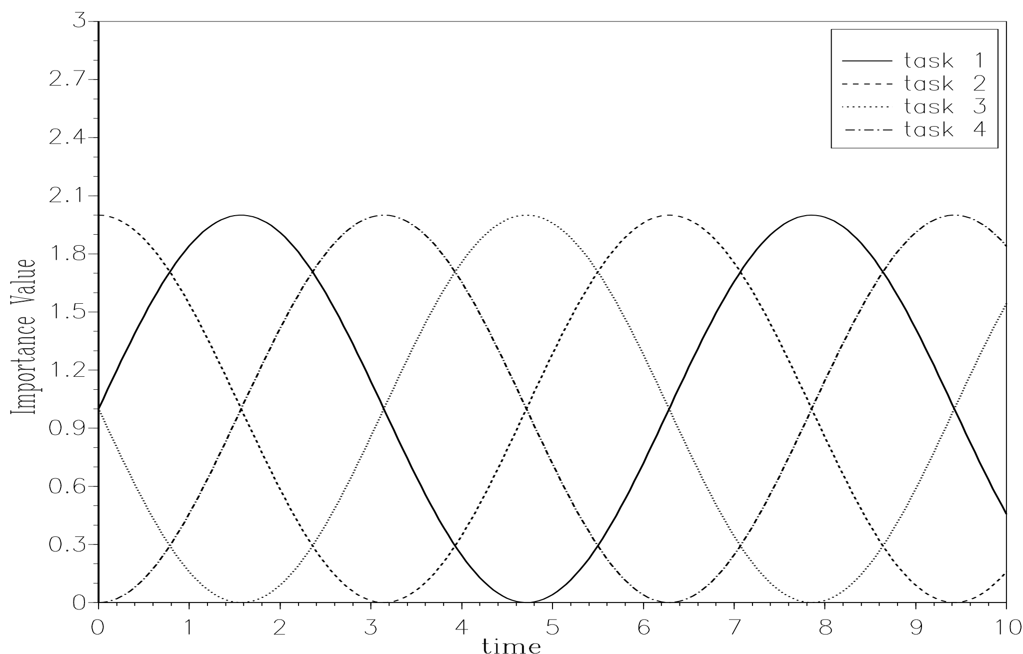

Figure 5 — Importance Function Values for Round Robin Policy

certain classes of functions, or *families*, can be expressed in general parametric forms where specific values are assigned according to the application. Jensen *et al*, in [JENS85], used a five parameter function to describe "value functions." It is claimed that the value of most interesting tasks can be profiled by

$$V_i(t) = K_1 + K_2 t - K_3 t^2 + K_4 e^{-K_5 t} \qquad \text{(Eq 13)}$$

where appropriate assignments of the constants K_i could produce value functions which are constant, linear, quadratic, exponential, or a linear combination of any of these.

The importance abstraction allows task importance to be profiled using functions of many forms. We have seen already how sets of functions can be used to emulate traditional scheduling policies. Yet, for most of these examples, we have given specific forms of the functions where the parameters are attributes of the task, like the task's deadline. By

examining the general forms of some of these functions, the families of functions available expand the expressiveness of the function form. For example, the nearest deadline first policy may be expressed as shown in (Eq 7); more generally, however, the form of the function could be given as

$$I_i(t) = \frac{\alpha_i}{\beta_i(d_i - t)} \quad \text{(Eq 14)}$$

where α_i and β_i are constants specific to task i. A set of importance functions based on this family may not necessarily provide nearest deadline first service, but rather exhibit additional properties, such as giving preference to meeting the deadlines of the most critical tasks.

4.3. Novel Policies using Importance Functions

The so-called traditional scheduling policies, and the algorithms which are used to express them, often arise from the requirements of the scheduling mechanisms. Many aspects of tasks and task sets, which should logically be expressed as scheduling parameters, are ignored, simplified, or encoded to make them useful to the traditional scheduling mechanisms. One of the most popular scheduling policies is priority ordering, where all aspects of the task are condensed into a single value. Another popular policy, rate monotonic, permits us to make static guarantees about the schedulability of a task set, but the task set must be expressed as a set of periodic tasks, even if this is inappropriate to do so. The importance abstraction, in addition to emulating traditional scheduling policies, allows us to focus on attributes, conditions, or other events that are not traditionally parameters of scheduling algorithms.

One such example of parameters which are difficult to consider in traditional scheduling models is continuously updated variables, as may be found in process control

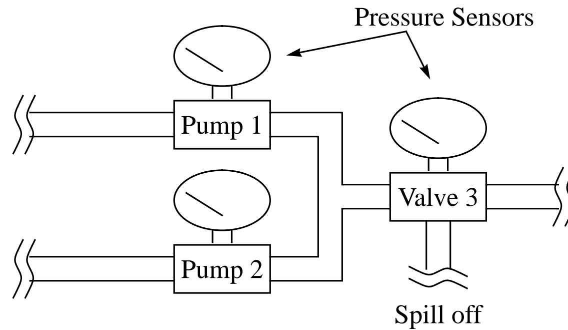

Figure 6 — Valve Configuration for a Process Control Example

applications. Values from sensors, for example, may affect the choice of tasks to process. For example, a sensor may monitor the pressure on a pump such that when the pressure at the pump deviates significantly from a normal value, the task controlling the pump becomes important.

Let V be such a continuously updated variable such that the value of V at time t is given by $V(t)$. We can include this variable within a task's importance function by composition:

$$I_i(t) = f_i(V(t)) . \qquad \text{(Eq 15)}$$

Consider a process control application where a single processor maintains two pumps and a valve in a configuration shown in Figure 6. Pump 1 must maintain $X \pm x$ units of pressure and Pump 2 must maintain $Y \pm y$ units of pressure to ensure a proper mixture flowing through to Valve 3. Valve 3 is a safety valve, protecting the machinery beyond by monitoring the pressure and spilling off any excess. If the pressure in Valve 3 builds to Z, Valve 3 must be made to divert some of the mixture to the spill off. Assume that each pump and valve is controlled by a separate task. The pressures at Pump 1 and Pump 2 are given by the continuously updated variables $P_1(t)$ and $P_2(t)$, respectively. The pressure on Valve

3 is the sum of the pressures produced by Pumps 1 and 2. The safety limit Z must be greater than or equal to $X + Y + x + y$.

We can define the importance functions for the three tasks which maintain these devices such that the task maintaining a pump or a valve will become important whenever that device needs attention. Pump 1 or Pump 2 needs attention when the pressure created deviates greater than x for Pump 1 or y for Pump 2. Valve 3 needs attention when the sum of the pressures from Pumps 1 and 2 approaches the limit Z. The importance functions below are designed to ensure that these tasks are invoked at the proper times.

$$\text{Define:} \quad \begin{aligned} \Delta_1(t) &= |P_1(t) - X| \\ \Delta_2(t) &= |P_2(t) - Y| \end{aligned} \qquad \text{(Eq 16)}$$

$$I_1(t) = \begin{cases} \Delta_1(t)x, & \text{if } \Delta_1(t) \le x \\ xy, & \text{otherwise} \end{cases} \qquad \text{(Eq 17)}$$

$$I_2(t) = \begin{cases} \Delta_2(t)y, & \text{if } \Delta_2(t) \le y \\ xy, & \text{otherwise} \end{cases} \qquad \text{(Eq 18)}$$

$$I_3(t) = \frac{(Z - X - Y - x - y)\ (xy)}{Z - P_1(t) - P_2(t)} \qquad \text{(Eq 19)}$$

The values $\Delta_1(t)$ and $\Delta_2(t)$ represent the deviations from the target pressures for Pump 1 and Pump 2 respectively. For an example where $X = 50$, $Y = 50$, $x = 5$, and $y = 7$, Figure 7 shows the graph of $I_1(t)$ and $I_2(t)$ versus the deviations from the ideal pressures. As either of the pumps produce pressure that deviates from the target, the importance of the task maintaining the deviating device increases. Figure 8 shows the importance of the task servicing the spill-off valve as a function of the pressures produced by Pumps 1 and 2. As

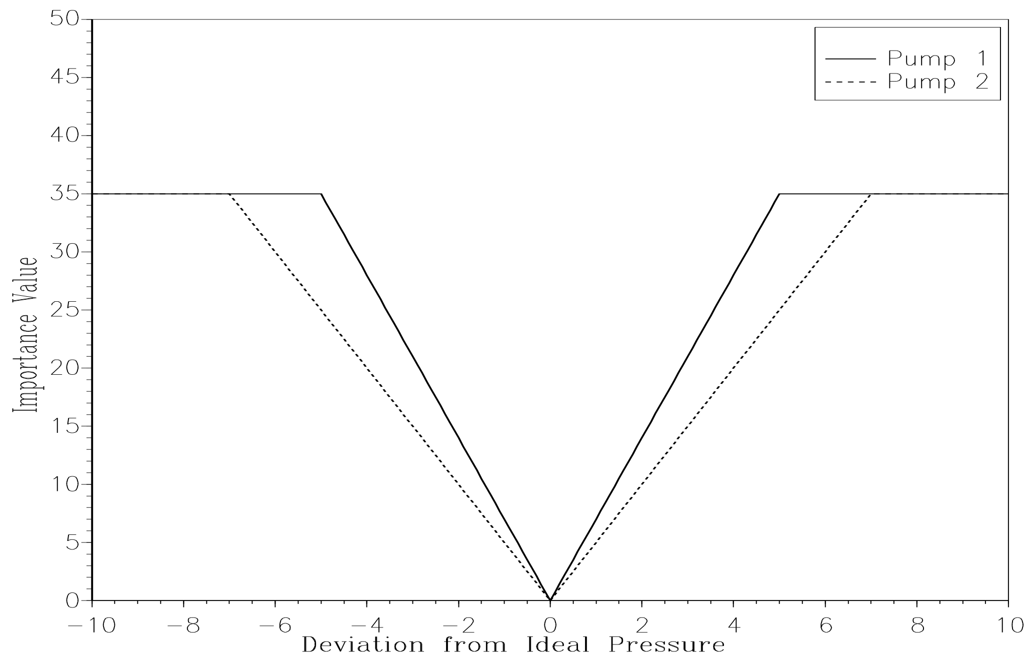

Figure 7 — Importance versus Deviations from Target Pressures

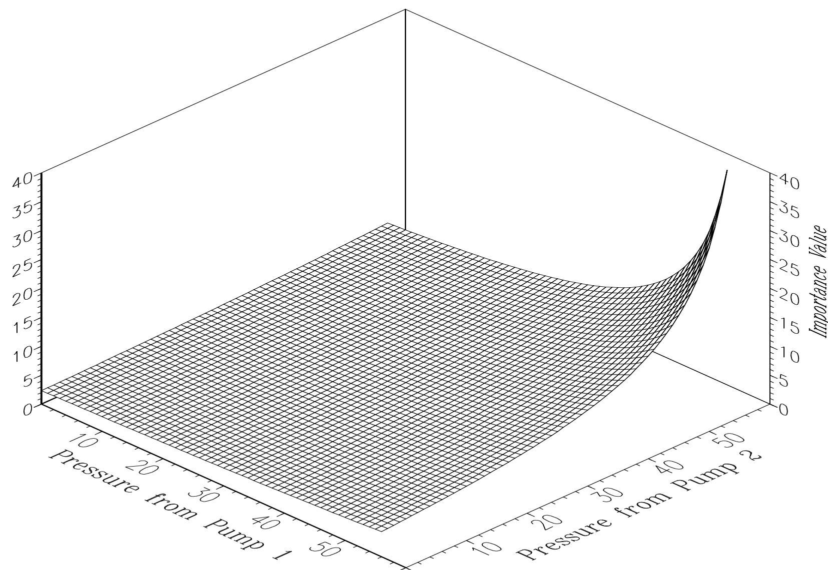

Figure 8 — Importance of Task Maintaining Valve 3

the combined pressures add to Z, which in this example we set to 120, the importance of the task for Valve 3 increases asymptotically.

Note that the design of the importance functions is based on the safety limits of the system so that the safety of the system is directly related to the behavior of the functions. Consequently, it is possible to know exactly which task will be most important under any set of circumstances. In this example, the pumps are serviced according to which pump is producing pressure proportionally closer to its limit of deviation. These tasks attempt to correct the pressure deviation. If the pressure at a pump exceeds its limit, something is wrong that can not be corrected by invoking the pump control tasks; the importance value becomes constant so that it will not compete with the task servicing the spill-off valve. The valve is guaranteed to be serviced if the combined pressures exceed $X + Y + x + y$ since, at this point, the importance of the task controlling the valve will exceed the importance of either task controlling the pumps. Also, The functions are designed to allow a task to become important as a limit is approached rather than after a fault has occurred.

Consider conventional methods for servicing tasks in process control systems. Polling loops are often used to "visit" each task in a round robin manner. At each visit, the task may find that corrections are required. A limit may be placed on how long a task is serviced so as not to starve other tasks. Polling creates a lag time between the occurrence of the problem and the servicing of the control task. The scheduler has no notion of the state of the devices controlled by the tasks; rather, it is up to the tasks to investigate the status of only the device for which it is the controller, and take action accordingly. Consequently, it is much more difficult to assure safety to such a system since the safety depends upon the worst case poll time. The poll time itself depends on the worst case time spent servicing each of the other tasks. This service time should be long enough to allow the proper

corrections to be made (or satisfactorily started) but short enough to allow service to be given to other tasks whose devices also have problems.

Some other methods are not appropriate. Static priority-based scheduling policies can not cope with the dynamics of the system. There are no deadlines in this system, so deadline-driven policies are inappropriate. Since these tasks are not naturally periodic, rate monotonic theory does not apply. Importance functions are a natural way to express the conditions under which the control tasks must take corrective action.

5. Analyzability

In the previous section we offered sets of importance functions which could emulate several traditional scheduling algorithms. In particular we have shown that priority-driven, nearest deadline first, and rate monotonic algorithms are easily expressed in terms of functions. It is interesting to note that all three of these policies have the property that tasks are ranked according to some criteria (priority, period, or deadline) and, once ranked, remain in this ranking relative to all other active tasks until some task leaves or some task arrives. We term this property a *static ranking*. In this section we examine these three scheduling policies to determine the *defining property* and, once found, prove that the importance function sets given as candidate sets for emulating these policies do in fact meet the defining property. Since these policies all present a static ranking, the proof that the candidate importance function set meets the defining property is similar for each policy. We continue by examining the nearest deadline first policy under certain relaxed or modified conditions.

Assume that a given system requires that, to meet the system goal, all tasks active in the system must be ranked according to some criteria known at task activation. Further assume that this ranking, once established within an active task set, does not change until

the composition of the task set changes. We term rankings that have this property *static rankings*. Several well-known traditional scheduling policies (e.g., priority-driven, nearest deadline first, and rate monotonic) have this property of static ranking.

The rank r for a task is based on the ranking criteria; for a priority-driven system this criteria is the priority, for rate monotonic it is the period of the task, and for nearest deadline first it is the deadline. Importance functions emulating these algorithms must use this ranking criteria as a parameter; moreover, at every point in time, the rank imposed must be maintained by the importance functions. Thus, for all time t,

$$r_i > r_j \Rightarrow I_i(t) > I_j(t) \qquad \forall (i, j \in T \text{ at time } t) \qquad \text{(Eq 20)}$$

This is the *defining property* for all scheduling policies based on static ranking.

The proof that a particular importance function set meets this defining property is trivial. For a priority-driven policy, the set of importance functions as given in (Eq 8) meet this property since, for all time t,

$$p_i > p_j \Rightarrow I_i(t) > I_j(t) \qquad \forall (i, j \in T \text{ at time } t) \qquad \text{(Eq 21)}$$

For rate monotonic, the priority is the inverse of the period, so the property holds in this case as well. For the nearest deadline first policy, the ranking is determined by the nearness of the deadline. Hence, for all time t,

$$\frac{1}{d_i - t} > \frac{1}{d_j - t} \Rightarrow I_i(t) > I_j(t) \qquad \forall (i, j \in T \text{ at time } t) \qquad \text{(Eq 22)}$$

$$d_i < d_j \Rightarrow I_i(t) > I_j(t) \qquad \forall (i, j \in T \text{ at time } t) \qquad \text{(Eq 23)}$$

Notice that this defining property does not state that all deadlines are met. The fact that, if all deadlines can be met, the nearest deadline first policy will meet them, is a consequence

of the ranking and not in itself the defining property. In fact, the nearest deadline first policy is only a single element in the set of policies that guarantee that all deadlines are met.

5.1. Completion Time

For policies which have this static ranking property, it is more interesting to ask questions about the schedules imposed by the sets of importance functions. For example, we can ask when a particular task will complete, or under what set of conditions will a task miss its deadline. In general, completion time is a difficult aspect to predict; for this discussion we assume knowledge of the complete task set and the attributes of the tasks within.

Assume that a task set T has cardinality n, and that the tasks within T have known arrival times. Let a_i be the arrival time for task i. Without loss of generality we can order the tasks in T such that the tasks are numbered from highest to lowest ranking. Hence, task i has ranking greater than or equal to task j if $i > j$. Tasks with equal rank are ordered by arrival; otherwise, arrival times have nothing to do with the ordering of the tasks in T. Assume the importance function set for task set T is given by

$$\forall (i \in T), \qquad I_i(t) = \begin{cases} 0, \text{ if } (t < a_i) \\ r_i, \text{ if } (t \geq a_i) \end{cases} \qquad \text{(Eq 24)}$$

for all $r_i \geq 0$.

For a given task j we seek the completion time c_j. There may be some tasks of higher importance that have arrived (become active) before a_j, and there may be tasks which arrive after a_j that are more important than task j. We can identify these more important tasks as those having indices less than j. Those active tasks that are more important than task j at

time a_j will complete before task j can begin. Let $p_{j,1}$ be the earliest time task j can start given that no tasks arrive after a_j:

$$p_{j,1} = \max_{1 \le i \le j-1}(c_i | (a_i \le a_j) \wedge (c_i < a_j + w_j)) \quad \text{(Eq 25)}$$

This states that task j can start no sooner than the greatest completion time of the more important active tasks.

But some tasks may arrive between a_j and $p_{j,1}$ plus the amount of work left to complete task j. During this period task j is subject to preemption by some higher ranking tasks, thus possibly delaying the completion time of task j. We call this a "vulnerable period." Since we must consider this, let $p_{j,2}$ be defined as follows:

$$p_{j,2} = \max_{1 \le i \le j-1}(c_i | (a_j \le a_i \le p_{j,1} + w_j|_{p_{j,1}}^{\infty})) \quad \text{(Eq 26)}$$

This states that task j can finish no sooner than the greatest completion time of any tasks arriving within the vulnerable period. We use the term $w_j|_{p_{j,1}}^{\infty}$ since this value reflects the amount of work left to do on task j after the time $p_{j,1}$. By considering the vulnerable period above, a new vulnerable period is created. To consider this new vulnerable period as well, define $p_{j,3}$ as follows:

$$p_{j,3} = \max_{1 \le i \le j-1}(c_i | (a_j \le a_i \le p_{j,2} + w_j|_{p_{j,2}}^{\infty})) \quad \text{(Eq 27)}$$

Continuing this chain of logic through iteration k:

$$p_{j,k} = \max_{1 \le i \le j-1}(c_i | (a_j \le a_i \le p_{j,k-1} + w_j|_{p_{j,k-1}}^{\infty})) \quad \text{(Eq 28)}$$

Since there are n tasks in the set T, there can be at most n–1 periods of vulnerability. Hence:

$$p_{j,n} = \max_{1 \le i \le j-1}(c_i | (a_j \le a_i \le p_{j,n-1} + w_j|_{p_{j,n-1}}^{\infty})) \quad \text{(Eq 29)}$$

Note that once $p_{j,k} = p_{j,k+1}$ for some k we need not to calculate any more periods of vulnerability, thus we can make the assignment $p_{j,n} = p_{j,k}$. Task j will complete at time c_j, where c_j is given by

$$c_j = p_{j,n} + w_j\Big|_{p_{j,n}}^{\infty} \qquad \text{(Eq 30)}$$

5.2. Meeting Deadlines

Since nearest deadline first is a static ranking, the result given in (Eq 30) also applies for schedules imposed using importance functions emulating the nearest deadline first algorithm. Such a set of importance functions is given in (Eq 7). We know from [LIU73] that, if deadlines can be met for a given task set, they will be met using the nearest deadline first policy. However, Liu and Layland show this for a set of *periodic* tasks by proving that nearest deadline first will schedule tasks to meet deadlines if the *utilization factor* (the sum over all tasks of the ratios of work required to length of period) for the task set is 1 or less. Unfortunately, the utilization factor proof in [LIU73] only holds for periodic task (a counterexample: task 1 has arrival time $a_1 = 5$, work required $w_1 = 10$, deadline $d_1 = 15$, and task 2 has $a_2 = 15$, $w_2 = 10$, $d_2 = 25$; the utilization factor is 2, yet the task set is feasible).

To show that the nearest deadline first algorithm will meet all deadlines for an aperiodic task set if there exists any schedule which can meet all deadlines, we must show that the completion time from (Eq 30) for each task is always less than or equal to the task's deadline; that is, for each task i with deadline d_i, $c_i \le d_i$. We prove this within the importance abstraction by using the property, given by (Eq 23), for the importance function set used to emulate the nearest deadline first algorithm. We also need the condition under which any schedule will meet all deadlines. For a schedule to meet every deadline in the task set the schedule must ensure that the following is true for all points in time:

$$\underset{1 \le j \le n}{\text{AND}} \left(\sum_{i=1}^{j} w_i \Big|_t^{\infty} \le \max\,(d_j - t, 0) \right) \qquad \text{(Eq 31)}$$

This is actually a set of conditions, all of which must be true. Consider $t = 0$. For $j = 1$, the amount of work done on task 1 over all time must not exceed its deadline. For $j = 2$, the amount of work done on task 2 in addition to the work done on task 1 must not exceed the deadline d_2. For $j = n$, where $n = |T|$, the amount of work done on all n tasks must not exceed the deadline of task n.

Theorem 1

Given a task set T for which there exists some schedule that meets all deadlines, then a schedule imposed by the nearest deadline first algorithm will also meet all deadlines.

Proof

Assume the tasks of task set T are scheduled by a set of importance functions for which (Eq 23) is a property. Let T be ordered such that $d_1 \le d_2 \le \ldots \le d_n$. Let task k be the lowest indexed task for which $c_k > d_k$, where c_k is the completion time and d_k is the deadline for task k.

There are two cases. First, if there is no idle time between time 0 and time c_k, then the sum of all of the work done on all tasks over that interval is the length of the interval and equals c_k. Therefore:

$$\sum_{i=1}^{n} w_i \Big|_0^{c_k} = c_k \qquad \text{(Eq 32)}$$

Since the property given in (Eq 23) holds for this task set, only the tasks whose deadlines are d_k or prior are serviced over the interval 0 to c_k; we may rewrite (Eq 32) as:

$$\sum_{i=1}^{k} w_i \Big|_0^{c_k} = c_k \qquad \text{(Eq 33)}$$

Also, these tasks are run to completion before task k is completed, so we can replace $w_i \Big|_0^{c_k}$ with w_i:

$$\sum_{i=1}^{k} w_i = c_k \qquad \text{(Eq 34)}$$

But the kth condition of (Eq 31), for $t = 0$, states:

$$\sum_{i=1}^{k} w_i\Big|_0^{\infty} \le \max\,(d_k - 0, 0) \;=\; d_k \qquad \text{(Eq 35)}$$

Since $w_i\big|_0^{\infty}$ equals all of the work required by the task, this expression in (Eq 35) can be replace by the value w_i:

$$\sum_{i=1}^{k} w_i \le d_k \qquad \text{(Eq 36)}$$

By substitution of (Eq 34) into (Eq 36), we arrive at $c_k \le d_k$, a contradiction of our initial assumption that $c_k > d_k$.

For the second case, if there is at least one gap of idle time between time 0 and time c_k, let t_g be the time when the last gap ends so that on the interval t_g to c_k there is no idle time. The work done over that interval must sum to the length of the interval:

$$\sum_{i=1}^{n} w_i\Big|_{t_g}^{c_k} \;=\; c_k - t_g \qquad \text{(Eq 37)}$$

Since (Eq 23) holds, no tasks of index greater than k will be serviced during this interval, so we can change the upper limit of the summation. Also, since each task with index k or less will finish before time c_k, we can replace the expression $w_i\big|_{t_g}^{c_k}$ with $w_i\big|_{t_g}^{\infty}$:

$$\sum_{i=1}^{k} w_i\Big|_{t_g}^{\infty} \;=\; c_k - t_g \qquad \text{(Eq 38)}$$

The kth condition of (Eq 31), for $t = t_g$, yields:

$$\sum_{i=1}^{k} w_i\Big|_{t_g}^{\infty} \le \max\,(d_k - t_g, 0) \qquad \text{(Eq 39)}$$

By substitution of (Eq 38) into (Eq 39), we arrive at:

$$\begin{aligned} c_k - t_g &\le d_k - t_g \\ c_k &\le d_k \end{aligned} \qquad \text{(Eq 40)}$$

Again, we find the contradiction.

Therefore, if there exists a schedule which can meet all deadlines in a task set, then the schedule imposed by the importance functions which emulate the nearest deadline first algorithm will also meet all deadlines. Since the importance functions and the algorithm impose the same schedule, then the result holds for the nearest deadline first algorithm as well.

∎

For rate monotonic, each task is instantiated exactly once during the task's period. This instantiation must complete before its period expires and the new instantiation is created. We can therefore think of each instantiation of a task as a separate task, and the end of the period as that task's deadline. In this sense rate monotonic is similar to the nearest deadline first algorithm where the deadline d_i is given by $d_i = a_i + T_i$, for T_i the period for task i.

5.3. Meeting Deadlines with Arbitrary Arrivals

Theorem 1 assumes that the task set T has a constant cardinality n and is known *a priori*. In a system where the task set T can not be known *a priori*, and where the cardinality is not known to be a constant (i.e., there may be arbitrary future arrivals), we can not prove that all deadlines will be met. We can, however, create a test which will identify as early as possible when a task will miss its deadline.

Let tasks be requested at arbitrary times such that the request time for task i is less than or equal to the arrival time for task i; that is, $req_i \le a_i$. Index the tasks such that, for all tasks $i, j \in T$

$$i > j \Rightarrow (req_i < req_j) \vee ((req_i = req_j) \wedge (a_i < a_j)) \quad \text{(Eq 41)}$$

Thus the tasks are indexed by when they are requested.

We need to define a few functions for convenience. Let $D{:}T \rightarrow N$ be a function that takes a task and returns a natural number representing the task's current order with respect

to deadline nearness. If task i has the jth nearest deadline, then $D(i) = j$. Let $D':N \rightarrow T$ be the inverse function which, given a natural number j, returns the task index whose deadline is currently the jth nearest. Let $n(t)$ be a function which returns the cardinality of the set T at time t. The condition for meeting all deadlines for the task set T at time t is:

$$\underset{1 \leq j \leq n(t)}{\text{AND}} \left(\sum_{i=1}^{D(j)} w_{D'(i)} \Big|_t^{\infty} \leq \max(d_j - t, 0) \right) \quad \text{(Eq 42)}$$

This condition is similar to the condition given in (Eq 31). This condition states that, at some time t and for all tasks j from 1 to the current cardinality of the task set T, the sum of the work required by all tasks whose deadlines are priori to task j must be less than or equal to the amount of time between the current time and task j's deadline. We define the term *overload* to be the state of the task set at time t such that (Eq 42) is not true.

Theorem 2

Let T be an arbitrarily large task set containing tasks with arbitrary request times. The nearest deadline first algorithm will meet all deadlines if any algorithm can meet all deadlines.

Proof

Assume that a system requests work on tasks at arbitrary time such that the size of the task set is not known *a priori*. Assume that task k is requested at time req_k, and at that time an overload occurs such that some task m can not meet its deadline using the nearest deadline first algorithm. At time req_k we can construct a task set T_k which includes all tasks requested from time 0 to time req_k. Let these tasks be indexed according to the nearness of their deadlines such that $i < j \Rightarrow d_i < d_j$. By application of Theorem 1 we know that no algorithm can meet all deadlines if the nearest deadline first algorithm can not meet all deadlines.

∎

5.4. Meeting Critical Deadlines, with Arbitrary Arrivals

One of the problems with a pure nearest deadline first algorithm is that the tasks are not otherwise ranked in the presence of missed deadlines such that the most critical tasks are given preference at the expense of the least critical. The importance abstraction can

easily express this bilevel ranking, where the nearest deadline first policy is augmented by considering criticalness measures associated with each task. Let us call this new for of nearest deadline first the nearest critical deadline first (NCDF). From the representation of the NCDF policy within the importance abstraction we seek the conditions under which a given task *k* will meet its deadline, and from that prove that NCDF maximizes a quantity based on the criticalness of the tasks serviced.

Let each task *i* have two attributes: the deadline d_i and a criticalness p_i. Assume that the criticalness p_i is an element of L, where L is the set of natural numbers in the range MINCRIT to MAXCRIT. To construct a set of importance functions which will implement the NCDF policy we first define a few auxiliary functions for convenience. Define the function $Over{:}\{T\} \times time \rightarrow Boolean$ as:

$$Over(T, t) = \underset{1 \le j \le |T|}{\text{AND}} \left(\sum_{i=1}^{D(j)} w_{D'(i)} \Big|_t^{\infty} > \max(d_j - t, 0) \right) \qquad \text{(Eq 43)}$$

The function *Over* returns one if the task set *T* will not meet all deadlines at time *t*, zero otherwise. Note that this is a functional representation of the conditions from (Eq 42). Define $Crit{:}L \rightarrow \mathcal{P}(T)$ as a function that takes the criticalness level from the set *L* and returns the subset of *T* that share this criticalness level. Finally, we define a function $InMostCrit{:}T \times time \rightarrow Boolean$ that takes a task and a time and returns one if the task is in the set of tasks whose deadlines will be met because they are among the most critical at that time, and returns zero otherwise. The function body is:

```
InMostCrit(i, t) {
    T′ = Ø
    for k = MAXCRIT downto MINCRIT do
        for each j ∈ Crit(k) do
            if not Over(T′ ∪ {j}, t)
            then
                T′ = T′ ∪ {j}
            endif
        endfor
    endfor
    return (i ∈ T′)
}
```

Now for the importance functions:

$$\forall (i \in T), \qquad I_i(t) = \begin{cases} 0, \text{ if } (t < a_i) \\ (d_i - t)^{-1}, \text{ if } ((d_i > t \geq a_i) \wedge InMostCrit(i, t)) \\ 0, \text{ otherwise} \end{cases} \qquad \text{(Eq 44)}$$

Given a task k with an importance function defined as above, we seek the conditions under which this task k will meet its deadline. Since we are now considering a task set with arbitrary future arrivals, we can not predict *a priori* that task k will meet its deadline; rather, we can show the conditions necessary at certain points in time for task k to meet its deadline. At time req_k, task k is schedulable if the following is true:

$$\left(\sum_{i=1}^{D(k)} (w_{D'(i)}\big|_{req_k}^{\infty})(p_{D'(i)} \geq p_k) \right) \leq d_k - req \qquad \text{(Eq 45)}$$

We must check this condition not at time req_k but every time a request for service is made, hence:

$$t = req_j\big|_{(req_k \leq req_j \leq d_k)}^{\text{AND}} \quad \left(\left(\sum_{i=1}^{D(k)} (w_{D'(i)}\big|_t^{\infty})(p_{D'(i)} \geq p_k) \right) \leq (d_k - t) \right) \qquad \text{(Eq 46)}$$

This expression states that, for each time t that a new task arrives between the request of task *k* and task *k*'s deadline, the following must be true: the sum of the work remaining for tasks whose deadlines are nearer than task *k*'s and whose criticality is at least as great as task *k*'s must be less than or equal to the difference between task *k*'s deadline and the time we are considering.

Biyabani *et al* explore this kind of bilevel ranking in [BIYA88]. Most notably they offer a new sematic for the term *guarantee* that reflects the uncertainty of the future task set composition. They state that at request time a task is guaranteed to meet its deadline if (1) it is among the most critical tasks in the current task set, and (2) the arrivals of subsequent tasks do not cause this task to leave the set of the most critical tasks. The system guarantees that the most critical tasks will meet their deadlines; however, we can not predict which tasks will be in the set of most critical tasks.

We constructed the importance functions so that only the most critical tasks are serviced to completion. When the system presents more tasks than can be serviced without missing a deadline, some tasks must be pruned. The condition *InMostCrit* is used within the importance functions of (Eq 44) to do this pruning. We can quantify how well the goal of meeting most critical deadlines is being met by summing the criticalness values for all tasks whose deadlines have been met by time *t*. Define the quantity *CritCount* as:

$$CritCount = \sum_{i=1}^{n(t)} \left(w_i \Big|_{a_i}^{d_i} \geq w_i \right) p_i \qquad \text{(Eq 47)}$$

When the work done on a task *i* is greater than or equal to the work required, the criticalness value of task *i* is added to the criticalness count *CritCount*. Because the NCDF policy is greedy, we expect the *CritCount* for the schedule produced to be optimal among all

policies. The following lemma supports a theorem that proves that NCDF is optimal with respect to maximizing this quantity.

Lemma 1

Any task set that is schedulable by the nearest deadline first (NDF) policy is also schedulable by the NCDF policy

Proof

Let T be a task set that is schedulable by NDF. Thus, by (Eq 42) we know that, for all time t, the following is true:

$$\underset{1 \le j \le n(t)}{\text{AND}} \left(\sum_{i=1}^{D(j)} w_{D'(i)} \Big|_t^{\infty} \le \max(d_j - t, 0) \right) \qquad \text{(Eq 48)}$$

Since the only difference in NDF and NCDF is the presence of the condition *InMostCrit*, as long as *InMostCrit* is true for some task i over all time t, then task i will be scheduled by both algorithms at exactly the same time, for exactly the same duration, and having exactly the same completion time. Let task k be a task schedulable by NDF but not by NCDF. Thus, *InMostCrit*(k,t) must not be true for some time t. This implies by (Eq 43) that

$$\sum_{i=1}^{D(k)} w_{D'(i)} \Big|_t^{\infty} > d_k - t \qquad \text{(Eq 49)}$$

But from (Eq 48) we know that

$$\sum_{i=1}^{D(k)} w_{D'(i)} \Big|_t^{\infty} \le d_k - t \qquad \text{(Eq 50)}$$

This is a contradiction.

∎

Theorem 3

The NCDF policy maximizes the criticalness count *CritCount* among all scheduling policies.

Proof

Assume that there exists some scheduling policy A that, at some time t, produces a schedule that has a higher value for *CritCount* than NCDF. Let T_A be the set of tasks scheduled by policy A by time t, and T_{NCDF} be the set of tasks scheduled by NCDF.

If these tasks are equal then their *CritCount*s must also be equal and thus we have a contradiction.

If the task sets are not equal, then the set of tasks chosen by policy *A* must contain some tasks not chosen by NCDF. For the quantity *CritCount* of T_A to be higher than that for T_{NCDF}, policy *A* either scheduled more tasks or instead scheduled tasks of greater criticalness. By Theorem 1 we know that the task set T_A can be scheduled by NDF. By Lemma 1 we know that any task set schedulable by NDF is also schedulable by NCDF. Therefore, policy *A* could not have scheduled more tasks than NCDF; instead, to have a higher value for *CritCount*, policy *A* must have scheduled different, more critical tasks.

Since, at every point in time, NCDF chooses the most critical task with the nearest deadline, any more critical tasks chosen by policy *A*, and therefore schedulable by both NDF and NCDF, would have also been chosen by NCDF. Thus policy *A* could not have scheduled more critical tasks than NCDF, and a contradiction results.

∎

5.5. Heterogeneous Task Sets

Consider a task set that contains some tasks that are only deadline-driven and some tasks that are only priority-driven. Because the priority-driven tasks do not have a time constraint, most policies schedule the deadline-driven tasks first and use any remaining processor cycles to service the priority-driven tasks. Policies of this type are easily constructed within the importance abstraction by using the following importance functions: Let T_d be the subset of T that are deadline-driven tasks and T_p be the subset of T that are priority-driven tasks. Let p be equal to MAXCRIT. The importance functions for both types of tasks are given by:

$$\forall (i \in T) \qquad I_i(t) = \begin{cases} (d_i - t)^{-1} + p, & \text{if } (i \in T_d \wedge a_i \le t < d_i) \\ p_i, & \text{if } (i \in T_p \wedge a_i \le t) \\ 0, & \text{otherwise} \end{cases} \qquad \text{(Eq 51)}$$

Since the importance of a deadline-driven task is always higher than the importance of any priority-driven task, Theorems 1 and 2 from the previous sections still hold. A

characteristic of schedules produced using this set of importance function is that priority-driven tasks are always deferred until there are no deadline-driven tasks in the set to be serviced. Thus, as a consequence of trying to meet the deadlines of the tasks of subset T_d the priority-driven tasks must wait until there are no active deadline-driven tasks.

Consider a system that must meet all deadlines as well as attempt to minimize the average response time to the priority-driven tasks. If there is no stated benefit from servicing the deadline-driven tasks sooner rather than later, as long as the deadline is met if it can be met, then we want a schedule that defers deadline-driven tasks to the last possible moment. Unfortunately, deferring deadline-driven tasks without *a priori* knowledge of future task arrivals may indeed cause some deadlines to be missed where not deferring the tasks (as with NDF and NCDF) would have met the deadlines. Consequently there must be restrictions on the task set in order to explore a policy that uses procrastination of deadline-driven tasks to reduce the response time for priority-driven tasks.

Clearly, the most conservative restriction is to require a fixed size task set that is known *a priori*. Let each of the n tasks in T be indexed thus: tasks 1 through m are elements of T_p and are ordered by increasing arrival times, and tasks m+1 through n are the elements of T_d and are ordered by increasing deadlines. To keep the procrastination of deadline-driven tasks from causing some task's deadline to be missed, the latest possible starting time for a given task i such that it can still meet its deadline must be determined. Define s_i to be this latest possible starting time:

$$s_i = \min_{i \le j \le n} \left(d_j - \sum_{k=i}^{j} w_k\right) \quad \text{(Eq 52)}$$

The restriction of a fixed size task set known *a priori* can be relaxed to allow arbitrary arrivals with conditions placed on when the request for service for each task is made. Assume that the tasks are now indexed by their request times such that $i < j \Rightarrow req_i < req_j$. The restriction must ensure that, if any two tasks' deadlines are sufficiently close together, then the tasks must be requested appropriately. Recall that $D'(i)$ returns the index of the task whose deadline is the ith nearest. If the difference between the deadlines of tasks $D'(i+1)$ and $D'(i)$ is less than the quantity $w_{D'(i+1)}$, then it is possible for task $D'(i)$ to be deferred in such a way as to interfere with the meeting of task $D'(i+1)$'s deadline. Both tasks can be taken into account if the task whose deadline is later is known about at the same time as or before the task whose deadline is nearer. Specifically, the request times for tasks $D'(i)$ and $D'(i+1)$ must be ordered such that:

$$d_{D'(i+1)} - w_{D'(i+1)} < d_{D'(i)} \Rightarrow req_{D'(i+1)} \leq req_{D'(i)} \qquad \text{(Eq 53)}$$

Rewriting (Eq 52) to reflect indexing tasks by request time order, the latest starting time for some task i is given by:

$$s_i = \min_{D(i) \leq j \leq |T_d|} \left(d_{D'(j)} - \sum_{k=i}^{j} w_{D'(k)} \right) \qquad \text{(Eq 54)}$$

In either case, a set of importance functions for a procrastination policy is:

$$\forall (i \in T) \qquad I_i(t) = \begin{cases} (d_i - t)^{-1} + p, & \text{if } (i \in T_d \wedge s_i \leq t < d_i) \\ p_i, & \text{if } (i \in T_p \wedge a_i \leq t) \\ 0, & \text{otherwise} \end{cases} \qquad \text{(Eq 55)}$$

5.6. Meeting Critical Deadlines for Heterogeneous Task Sets, with Arbitrary Arrivals

We can combine the conditions from the importance functions of (Eq 44) and (Eq 55) to form a set of importance functions that provide guaranteed service to the most critical deadline-driven tasks while minimizing the average response time for tasks that are

priority-driven. Assuming the restrictions on the request times for the task set as given in (Eq 53), the importance functions are:

$$\forall (i \in T) \qquad I_i(t) = \begin{cases} (d_i - t)^{-1}, \text{ if } (i \in T_d) \\ \qquad \wedge\ (s_i \le t < d_i) \\ \qquad \wedge\ InMostCrit(i, t) \\ p_i, \text{ if } (i \in T_p \wedge a_i \le t) \\ 0, \text{ otherwise} \end{cases} \qquad \text{(Eq 56)}$$

In the nearest deadline first policy processor idle time occurs only after the deadlines of all of the active tasks are met. With a heterogeneous task set, the idle time is used to service the priority-driven tasks. When the deadline-driven tasks are deferred until the last possible moment, the priority-driven tasks are serviced sooner, thus moving the idle time in between the servicing of tasks from T_p and tasks from T_d. The final variation on the nearest deadline first policy presented here observes that, although there may be no benefit from servicing deadline-driven tasks earlier than later, there is no benefit from waiting to serve them while idle time exists. We construct a set of importance functions that implement a policy that (1) meets the deadlines for tasks in T_d, (2) prunes the least critical deadline-driven tasks when necessary, (3) reduces the response time for tasks in T_p, and (4) eliminates processor idle time if any task is active.

Define the function $Active: \{T\} \to Boolean$ to take a task set and return the value one if the set has any tasks which have arrived but for whom service is not completed, and return value zero otherwise. The set of importance functions is:

$$\forall(i \in T) \qquad I_i(t) = \begin{cases} (d_i - t)^{-1}, \text{ if } (i \in T_d) \\ \qquad \wedge\ (s_i \le t < d_i) \\ \qquad \wedge\ InMostCrit(i, t) \\ p_i, \text{ if } (i \in T_d \wedge a_i \le t < s_i \wedge \neg Active(T_p)) \\ p_i, \text{ if } (i \in T_p \wedge a_i \le t) \\ 0, \text{ otherwise} \end{cases} \qquad \text{(Eq 57)}$$

Schedules produced using these importance functions will service deadline-driven tasks in criticalness order during what would have been idle time until either some priority-driven task becomes active or the current time equals the latest possible start time for this task.

Since servicing tasks from T_d during the idle time will affect the latest possible start time, the term s_i can be made into a continuous function $s_i(t)$:

$$s_i(t) = \min_{D(i) \le j \le |T_d|} (d_{D'(j)} - \sum_{k=i}^{j} w_{D'(k)} \Big|_t^{\infty}) \qquad \text{(Eq 58)}$$

Replacing s_i with $s_i(t)$ in (Eq 57) will constantly update the latest possible start time. As this time is made later, the priority-driven tasks are given longer service times before being preempted for the deadline-driven tasks. This further reduces the average response time for tasks in T_p.

6. Efficiently Implementable

The scheduler within the importance abstraction logically consists of a function M which returns the most important task at the moment the function M is evaluated. In the general case, importance functions can be arbitrarily complex. To ensure the service of the most important task at every point in time, the function M must be evaluated at every point in time. This assumption serves a purpose for the theoretical analysis of sets of importance

functions, but for real systems, an implementation of such a scheduler would be impractical.

Although the importance abstraction places no restrictions on the complexity and composition of the set of importance functions profiling the task set for a particular system, it is clear that certain sets of importance functions may have properties which lend to an efficient implementation of the scheduler. These properties form three main classes: (1) those sets of importance functions for which discrete evaluation points can be determined, (2) those sets of importance functions which maintain the same relative ranking over time, and (3) those sets of importance functions for which discrete evaluation points must be assigned.

6.1. Discrete Evaluation Points

There are two ways to determine discrete points in time for importance function evaluation: (1) find the points of intersection of the importance functions, and (2) determine when the parameters to the importance function may change. Using the intersection method may be more efficient in terms of reducing the number of evaluation points; keying on the parameters may be easier to implement if the changes in the state or value of the parameters can be signaled, as with an interrupt.

With the intersection method we begin with the highest-valued function at this point in time. By pairwise evaluation we can determine when the next intersection point will occur, and as a consequence which function will be the next highest-valued function, since it is exactly at this point of intersection that one task becomes more important than the currently most important task. The function of the new most important task is used in the pairwise evaluation to determine the next point of intersection, and hence the next scheduling point. Unfortunately, in the general model importance functions can be

arbitrarily complex, and finding intersection points can be quite difficult. In fact, for functions of degree four or greater, it is impossible, in general, to find the intersection points of any two given functions. Even degree three is difficult, though there exists a closed form expression for finding the roots.

The other method for determining evaluation points is to use changes in the parameters to the functions to signal reevaluation of the functions. Many of the system characteristics which may be used as parameters to the importance functions change at discrete times as the result of an "event." If these discrete times can be known *a priori*, then these times can be built into the scheduler. If the events are signaled by the system, then the scheduler can use this signal. An important example is the set of parameters whose values are determined after a system interrupt. If it can be shown that the importance functions will maintain a stable relative ranking until one of a set of identifiable events occurs, then the scheduler can use that set of signals of the events to determine the scheduling point.

6.2. Static Rankings

Consider the set of importance functions which emulate static ranking scheduling policy. Recall that a static ranking r implies that, for all active tasks, $r_i > r_j \Rightarrow I_i(t) > I_j(t)$, for the range of time during which both tasks i and j are active. Since we know that the only events which will change the task set composition, and hence which task is most important, are task arrivals and departures, it is only for these events that the scheduler need be invoked. The fact that, given a task set, the tasks can be ordered by a ranking r means that the scheduler could maintain an ordered list, adding to it upon arrival and removing from it upon departure. Therefore, only at task arrival do the importance functions need to be evaluated. Note that this is a special case of the events method above.

6.3. Approximations

For some situations it may not be possible to discover simplifying aspects of the set of importance functions. Consequently, in this case the scheduler must evaluate every function in the set of importance functions as often as is possible. Unfortunately, it may be impossible to assure that the most important task is being scheduled at every point in time. In this case, the implementation approximates the importance abstraction.

The value of the function at its time of evaluation approximates the value of the function until the next time the function can be evaluated. Furthermore, if there are no parallel evaluations, the values will be the result of evaluations of each function at a different time. Fortunately, this is a common problem in real computing systems, so the solutions are not esoteric.

The degree of imprecision tolerable by a specific system is system-dependent. By system examination the worst-case time between evaluations can be determined. If the system can tolerate the possibility of the wrong task being serviced for at most the worst-case amount of time, then the approximations are adequate for the system. The analysis of the system suggested in Chapter 6 must take these approximations into account.

If the system can not tolerate the degree of imprecision inherent, the importance abstraction can be implemented in dedicated hardware. If a digital processor is used, the approximations still exist but the worst-case time between evaluations may be reduced to a tolerable level. Otherwise, an analog device may be used, eliminating the need for approximations.

7. Conclusions

The *importance abstraction* is a universal framework for expressing scheduling polices. The framework is universal in that the scheduling algorithm does not vary with the

policy—choose the most important task at every point in time. Each task has associated with it a function that profiles that task's importance to the system over time. The function, called an *importance function*, reflects the task's importance by taking as parameters all of the task attributes and system characteristics that may cause the task to become more or less important to the system. As the system's characteristics change over time, the task which is most important to the system may change as well.

Traditional scheduling algorithms are easily emulated within this abstraction by creating importance functions that cause particular tasks to become most important at precisely the same instant that the scheduling algorithm would have chosen that task for service. Novel scheduling policies may also be modeled due to the flexibility of using functions to express the policy.

Since the scheduling policies are expressed in terms of sets of functions, these sets may be manipulated and analyzed using mathematical techniques. The scheduling problem is therefore moved from the traditional algorithmic domain to the functional domain, where mature analytical tools may be employed.

Even though the importance abstraction relies on a central algorithm that requires that every function be evaluated at every point in time, there are certain classes of importance function sets that allow us to relax this requirement. Scheduling policies whose importance function representation belongs to this class can be implemented in an efficient manner using functions to driven the scheduling.

The importance abstraction is a new way to express scheduling problems. It places the emphasis on individual tasks and what makes them important to the system, rather than fitting a task set onto a well-known algorithm in order to use its analytical results. Since the importance abstraction expresses the scheduling problem in terms of what tasks are most

important, a wide range of problems can be presented under a unified abstraction and analyzed using similar tools.

8. Bibliography

[BERN71] Bernstein, A. J. and Sharp, J. C., "A Policy-Driven Scheduler for a Time-Sharing System", Communications of the ACM, Vol. 14, No. 2, pp. 74-78, (February 1971).

[BIYA88] Biyabani, S. R., Stankovic, J. A. and Ramamritham, K., "The Integration of Deadline and Criticalness in Hard Real-Time Scheduling", *Proceedings of the 1988 IEEE Real-Time Systems Symposium*, Huntsville, Alabama, pp. 152-160, (December 6-8, 1988).

[DHAL78] Dhall, S. K. and Liu, C. L., "On a Real-Time Scheduling Problem", *Operations Research*, Vol. 26, No. 1, pp. 127-140, (January-February 1978).

[GOOD88] Goodenough, J. B. and Sha, L., "The Priority Ceiling Protocol: A Method for Minimizing the Blocki/ng of High-Priority Ada Tasks", Technical Report CMU/SEI-88-SR-4, Carnegie-Mellon University Software Engineering Institute, March 1988.

[JENS85] Jensen, E. D., Locke, C. D. and Tokuda, H., "A Time- Driven Scheduling Model for Real-Time Operating Systems", *Proceedings of the Real-Time Systems Symposium*, pp. 112-122, (December 3-6, 1985).

[LEHO87] Lehoczky, J. P., Sha, L. and Strosnider, J. K., "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments", *Proceedings of the 1987 IEEE Real-Time Systems Symposium*, San Jose, California, pp. 261-270, (December 1-3, 1987).

[LIU73] Liu, C. L. and Layland, J. W., "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *Journal of the ACM*, Vol. 20, No. 1, pp. 46-61, (January 1973).

[LOCK86] Locke, C. D., "Best-Effort Decision Making for Real- Time Scheduling", Dissertation (Computer Science Report No. CMU-Computer Science-86-134), Carnegie- Mellon University Department of Computer Science, May 1986.

[LOCK88] Locke, C. D. and Goodenough, J. B., "A Practical Application of the Ceiling Protocol in a Real-Time System", Technical Report CMU/SEI-88-SR-3, Carnegie- Mellon University Software Engineering Institute, March 1988.

[RUSC77] Ruschitzka, M. and Fabry, R. S., "A Unifying Approach to Scheduling", *Communications of the ACM*, Vol. 20, No. 7, pp. 469-477, (July 1977).

[SHA86] Sha, L., Lehoczky, J. and Rajkumar, R., “Solutions for Some Practical Problems in Prioritized Preemptive Scheduling”, *Proceedings of the 1986 IEEE Real-Time Systems Symposium*, New Orleans, Louisiana, pp. 181-191, (December 2-4, 1986).

[SHA87] Sha, L., Rajkumar, R. and Lehoczky, J. P., “Priority Inheritance Protocols: An Approach to Real-Time Synchronization”, Technical Report CMU-Computer Science-87-181, Carnegie-Mellon University, Computer Science Department, 1987.

[SHA90] Sha, L. and Goodenough, J. B., “Real-Time Scheduling Theory and Ada”, *IEEE Computer*, Vol. 23, No. 4, pp. 53-62, (April 1990).

[SPRU88] Sprunt, B., Lehoczky, J. and Sha, L., “Exploiting Unused Periodic Time For Aperiodic Service Using The Extended Priority Exchange Algorithm”, *Proceedings of the 1988 IEEE Real-Time Systems Symposium*, Huntsville, Alabama, (December 6-8, 1988).

[TOKU87] Tokuda, H., Wendorf, J. W. and Wang, H., “Implementation of a Time-Driven Scheduler for Real- Time Operating Systems”, *Proceedings of the 1987 IEEE Real-Time Systems Symposium*, San Jose, California, pp. 271-280, (December 1-3, 1987).

[WEND88] Wendorf, J. W., “Implementation and Evaluation of a Time-Driven Scheduling Processor”, *Proceedings of the 1988 IEEE Real-Time Systems Symposium*, Huntsville, Alabama, pp. 172-180, (December 6-8, 1988).

Appendix A

In Chapter 2 we discussed an inherent attribute of a task, called the task *length*. The task length is the amount of time required for a processor to complete the task, including securing any additional resources, scheduling this and all other active tasks, and other associated latencies, such as context switching. Here we expand upon the discussion of the task length by providing a notation and an algebra for task lengths.

Define w_i as the amount of processing time required by task *i*. For some tasks it may be possible to calculate or estimate the amount of processing time required; for others it may not be possible to know the processing time requirement until the task finally completes. The quantity w_i has a definite value; however, that value may not be known *a priori.*

Define the amount of work done on task *i* over the interval (*a, b*) as $w_i\big|_a^b$. Since the work on a task cannot exceed the time allotted for that work,

$$w_i\big|_a^b \leq b - a \qquad \text{for } (b \geq a)\,. \tag{Eq A.1}$$

For the degenerate case[3] of $b < a$, $w_i\big|_a^b = 0$.

Assume that task 1 is processed until completion. Then,

$$w_i = w_i\big|_0^\infty \tag{Eq A.2}$$

Furthermore, the sum of the work done to a task before some point in time *t* and the work done after that point *t* is the total work done to the task:

$$w_i = w_i\big|_0^\infty = w_i\big|_0^t + w_i\big|_t^\infty \tag{Eq A.3}$$

When a task *i* has a deadline d_i, then

[3]. The normal case is assumed for the rest of this discussion.

$$w_i\big|_a^b \le \min(d_i, b) - a \qquad \text{(Eq A.4)}$$

Consider two tasks, *i* and *j*, receiving work over some interval *a* to *b*,

$$w_i\big|_a^b + w_j\big|_a^b \le b - a \qquad \text{(Eq A.5)}$$

For some set of tasks *T* receiving work over the interval *a* to *b*,

$$\sum_{i:\tau_i \in T} w_i\big|_a^b \le b - a \qquad \text{(Eq A.6)}$$

For tasks *i* and *j* considered over the intervals *a* to *b* and *a* to *d* respectively,

$$w_i\big|_a^b + w_i\big|_a^d \le \max(b, d) - a \qquad \text{(Eq A.7)}$$

Considering these two tasks over the intervals *a* to *b* and *c* to *b*,

$$w_i\big|_a^b + w_i\big|_c^b \le b - \min(a, c) \qquad \text{(Eq A.8)}$$

For task *i* over the interval *a* to *b* and task *j* over the interval *c* to *d*, the following is true:

$$w_i\big|_a^b + w_i\big|_c^d \le \max(b, d) - \min(a, c) \qquad \text{(Eq A.9)}$$

The quantity $w_i\big|_a^b$ is either (1) known, (2) derived, (3) assigned, or (4) unknown. At time *t*, $w_i\big|_0^t$ is *known* since the amount of work done for task *i* at time *t* is known at time *t*. Also at time *t*, $w_i\big|_t^\infty$ is *derived* as $w_i - w_i\big|_0^t$. If it is determined that from time *a* to time *b* task *i* will get 3 time units of work (for $b - a \ge 3$) then $w_i\big|_a^b = 3$ is assigned. Otherwise, $w_i\big|_a^b$ is unknown. If $w_i\big|_a^c$ is known, derived, or assigned, then

$$w_i\big|_a^b - w_i\big|_a^c = w_i\big|_c^b \qquad \text{for } (b \ge c)\,. \qquad \text{(Eq A.10)}$$