Emulating Soft Real-Time Scheduling Using Traditional Operating System Schedulers *

Brad Adelberg† Hector Garcia-Molina‡ Ben Kao§

April 25, 1994

Abstract

Real-time scheduling algorithms are usually only available in the kernels of real-time operating systems, and not in more general purpose operating systems, like Unix. For some soft real-time problems, a traditional operating system may be the development platform of choice. This paper addresses methods of emulating real-time scheduling algorithms on top of standard time-share schedulers. We examine (through simulations) three strategies for priority assignment within a traditional multi-tasking environment. The results show that the emulation algorithms are comparable in performance to the real-time algorithms and in some instances outperform them.

Keywords: soft real-time, priority assignment, scheduling.

1 Introduction

In this paper we focus on "soft real-time" applications, which have the following characteristics:

- tasks have real-time deadlines;
- missing some task deadlines is acceptable;
- the goal is to minimize the number of missed deadlines;
- task arrival and system load are unpredictable.

A typical soft real-time application is telecommunications. Here a missed deadline might correspond to a dropped call. Another example is program trading for financial markets. Missing a deadline will correspond to missing a trading opportunity. In both cases missed deadlines are

*This work was supported by the Telecommunications Center at Stanford University and by Hewlett Packard Company.

†Stanford University Department of Computer Science. e-mail: adelberg@cs.stanford.edu

‡Stanford University Department of Computer Science. e-mail: hector@cs.stanford.edu

§Princeton University Department of Computer Science. Current Address: Stanford University Department of Computer Science. e-mail: kao@cs.stanford.edu

to be avoided but will not cause calamitous results. Real-time systems typically run on top of an operating system that provides basic services such as scheduling and memory management. The choice of operating system for a soft real-time operating system is an important issue. Previously, most researchers assumed a dedicated machine running a real-time operating system (RTOS). Now, however, there is interest in techniques for developing real-time applications on general purpose operating systems (GPOSs), like Unix. At least one implementation study [MT89] has demonstrated that using a GPOS eases an application's implementation and results in extremely high quality code. This is especially promising since in practice, general purpose operating systems (GPOSs) are much more common than RTOSs.

While there will probably always be a need for RTOSs, there are at least three reasons to believe that GPOSs will become more popular for soft real-time applications:

- As real-time system design moves out of its closed community and into the general computing population, programmers will want to develop on the platforms which they're familiar with. Traditional operating systems, like Unix, have accumulated a large suite of development tools. In addition, programs written in a GPOS are more portable than programs written for proprietary operating systems.
- Many applications will be split across the real-time/batch processing boundary. For example, investment bankers may interface with a real-time system to monitor trading opportunities, while at the same time using spreadsheets, news readers, and other non-critical applications. Also, the inclusion of multimedia will introduce time constraints (20 frames/sec) into otherwise non-real-time applications. Running all components of an application under a single OS will ease development as compared to a heterogeneous approach.
- Using only a GPOS will reduce total system cost. Economy of scale has driven the price of a GPOS much lower than that of a real-time kernel. Increased speed of development and code quality will also reduce product expense.

This is why we believe that soft real-time processing must be intergrated into traditional OSs. In fact, the research we report on here was motivated by our implementation of a real-time database at Stanford [AGMK94]. We do not have the resources to purchase a real-time OS, nor the staff to maintain it. Hence, we are implementing our database system on a conventional Unix system, HP-UX from Hewlett Packard. We suspect many other users of soft real-time systems will be in the same situation.

In this paper we study the *real-time emulation* (RTE) problem, which we define as how to build soft real-time scheduling on top of a traditional OS. In Section 3, we look at three approaches to RTE and settle on one: design an algorithm to assign a priority to a new process according to its real-time constraints in such a way that the priority scheduling done by the GPOS mimics that of a real-time scheduler (such as earliest deadline first, least slack first). We call this type of algorithm a *priority assignment algorithm* because it must use the real-time information about a task (i.e. deadline, slack, ...) to determine which OS priority level to assign to it.

To illustrate the difficulties we will face in emulating a real-time scheduler, suppose we have an OS with 5 priority levels, numbered 0 to 4, with level 0 having the highest priority. Initially, the system is idle, and a task A arrives. What priority do we run it at? Well, suppose we run it at a middle priority, in this case 2. While A is running, task B arrives with an earlier deadline than A. Since the OS should schedule B first, we will assign it a priority of 1. Of course, if a new task arrives with a deadline in between those of A and B, we are stuck since there is no priority to assign to it. The algorithms we will present will have to cope with situations like this. We will also evaluate the performance of our new algorithms, and compare them against conventional real-time ones. As we will see, not only do the emulation algorithms perform well, in some cases they outperform the real-time algorithms.

The priority assignment algorithms which we develop will have applicability in other scenarios. For instance, designers trying to interface real-time systems to token-ring networks need to assign priorities. Tasks in the real-time system have deadlines associated with them, but the token-ring only supports message priorities, usually 8 levels. Somehow a message's priority must be assigned based on the deadline of the task that sends it. This is similar to the RTE problem applied to earliest deadline first scheduling studied in this paper, although the token-ring problem requires extensions for distributed scheduling.

To study the RTE problem, we assume that we have no apriori knowledge of real-time task arrival patterns or execution requirements. Given the application areas outlined above, we expect that little will be known about the real-time requests that will be made. Unlike hard real-time systems used in control applications, where tasks are periodic and of known execution time, our soft real-time system will probably be used in less structured situations, with tasks being event driven and unpredictable. We assume a task receives its deadline just before it is submitted for execution.

The rest of this paper is organized as follows. In Section 2, we mention some related work. Next, in Section 3 we explore the possible approaches to the priority assignment problem and focus on one. Section 4 describes the logical base model for our study. Different priority assignment strategies are introduced in Section 5. A brief description of our simulation experiments is contained in Section 6. In Section 7 we display and analyze the results of our experiments. Finally, in Section 8 we present conclusions.

2 Related Work

A lot of research has been done on real-time scheduling in various environments, be it I/O scheduling, processor scheduling, or transaction scheduling [AGM90, AGM88a, AGM88b, LL73, HTT89, CW90]. Through this work, the behavior and properties of earliest arrival (EA) first, earliest deadline (ED) first, and least slack (LS) first have been delineated. These studies all assume an infinite range of priorities and a custom scheduler. The problem of scheduling with a limited number of priority levels was studied in [SLR86], but centered on rate monotonic scheduling for periodic tasks in hard real-time systems.

Some researchers have studied how to develop real-time systems on traditional operating systems. [FPG+89], [Wel93], [Cra88], and [MT89] have all studied the suitability of Unix for real-time applications. [FPG+89] identifies two properties as essential for an operating system which is to support real-time applications: *Performance* and *Determinism*. The paper then shows that REAL/IX, a fully preemptive Unix, compares favorably to a real-time OS based on the standards above. [MT89] studies a different real-time Unix, RX-UX 832, and comes to similar conclusions. Finally, [Wel93] examines two other GPOSs, SCO XENIX System V and OS/2, and concludes that both may be viable for real-time applications, with OS/2 being particularly well suited due to its high predictability. While these studies demonstrate that a GPOS can be used in many real-time applications, none address the problem of priority assignment. It is implicitly assumed that process priorities can be determined during the design of the application, probably by a variant of the general rate monotonic algorithm.

3 General Approaches

Our goal is to emulate a real-time scheduler on top of a GPOS scheduler. Solutions to this problem vary depending on the accuracy of the emulator that is desired, the amount of total coding complexity that is tolerable, and the distribution of new code between applications and the system that is appropriate. The easiest solution, in terms of coding complexity, is to set process priorities and then let the GPOS kernel schedule processes using its native algorithm; If the priorities are assigned based on the tasks' real-time constraints, the GPOS scheduler may emulate the real-time algorithms very closely. The other extreme is to write a threads package to run on top of a GPOS with a custom scheduler in it: The entire application and its many threads of control would appear to the OS as one process [1]. The custom scheduler could be designed to use any real-time algorithm desired. The price one pays for this degree of control is huge development costs and lack of portability. The author of the threads package will need to write machine dependent code normally provided by an OS, such as context switch code.

We chose to study the first option for its simplicity and ease of implementation. Each process that starts a task must call a common routine to determine a priority for the new task. In this paper, we will not investigate changing the priorities of processes after their arrival (this idea is discussed further in Section 5.1.2). The routine can be part of a shared library if the GPOS supports them, and can use shared memory for any global data-structures related to scheduling. The challenge with this approach is to determine which priority level to choose for each newly arrived process in order to emulate a particular real-time scheduling algorithm (i.e. ED, LS). In general, the tighter a task's timing constraint is, the higher a priority it will receive. Priority assignment is difficult both because there are only a limited number of priority levels and because the emulator does not know what future tasks will be arriving and what their requirements will be. Another disadvantage is that some GPOS schedulers use algorithms that adjust process priorities as they run based on CPU usage, which would sabotage the intent of the priority generating algorithm [2]. Still, the simplicity and portability of this approach makes it worthy of study.

4 Base Model

The system consists of a single processor which runs all of the real-time tasks. We assume that the operating system being run is a general purpose OS, like Unix, and not a real-time OS. We assume that the system is dedicated to the RT application. This is necessary because if batch processes are run on the same machine, they will compete for processor time and interfere with the real-time processes. Some GPOSs, like Posix Unix, support a special class of processes which have higher priority than all batch processes. On a Posix system, batch

[1] No other processes would be run on the system to avoid interference.

[2] Scheduling algorithms that adjust priorities, like round robin multi-level feedback, are studied in an extended version of this paper [AGMK].

and real-time processes can be run concurrently since the batch processes will run only if no real-time processes are waiting.

4.1 Priority in Traditional OS's

Traditional operating systems work with a fixed number of discrete priority levels, not the continuous range of priorities associated with ED and LS. Each priority level has a queue associated with it. The numbering scheme used can be a source of confusion: the highest priority processes have the lowest priority level. To keep these two concepts clear, $p_{real}(T)$, where $p_{real}(T) \in \Re$, will denote the real-time priority of task T. If $p_{real}(T_1) > p_{real}(T_2)$, T_1 should have precedence over T_2. Similarly, $p_{os}(T)$, where $p_{os}(T) \in \mathcal{I}$ and $0 \leq p_{os}(T) < n$, will refer to the OS notion of priority levels. If $p_{os}(T_1) > p_{os}(T_2)$, T_2 will have precedence over T_1. In this paper, whenever the priority of a task T is referred to without a clear context, we will mean $p_{real}(T)$. Similarly, the *level* of T will refer to $p_{os}(T)$.

4.2 Task Model

In our model, when a task T arrives at the system, the following four values will be known:

$a(T)$ arrival time of T;
$x(T)$ execution time of T;
$s(T)$ slack of T;
$d(T)$ deadline of T.

Only three of these attributes must be provided to the system since they are related by the equation $d(T) = a(T)+x(T)+s(T)$. Other attributes of tasks that are generated and maintained by the emulator will be discussed later. I/O requests, resource contention, and other application specific effects are not studied. $a(T)$ and $d(T)$ are always known to the system on task T's arrival, but $x(T)$ is required only by the least slack first algorithm and its variants. Although a real system would only have an estimate of $x(T)$, for our study we believe it is reasonable to use actual value of $x(T)$ because our goal is to compare our algorithms to least slack, not to study the impact of erroneous estimates for $x(T)$. (Incidentally, the impact of errors in the estimate of $x(T)$ has been studied (e.g., [AGM90]); the scheduling algorithms are not very sensitive to errors.)

The goal of the scheduler is to minimize the number of tasks that miss their deadlines. For the tasks that do miss their deadlines, however, we need an overload management policy.

Suppose that task X has already missed its deadline but has not completed execution. One option is to abort X as soon as it misses its deadline, under the assumption that whatever it was doing is now useless. (Example: after analyzing the current state of the stock market, a decision is made to sell certain stock. A task X is issued to sell by a given time. If the time is exceeded, it is best to abort X, as the market conditions may have changed.) A second option is to continue to process X, under the assumption "better late than never." (Example: at a bank, customers are "guaranteed" a two second response time. However, if the guarantee is not met, it is still desirable to complete the task.) In this paper, we focus on the no abortion case (no specific action is taken when a deadline expires). This will fit in best with our goal of setting process priorities and then letting the operating system handle everything. The performance of standard scheduling algorithms under different overload management policies is studied in [AGM92].

5 Scheduling Algorithms

In this paper we discuss two classes of scheduling algorithms: real-time and emulated. The real-time algorithms are included for comparison purposes and must be implemented on a real-time kernel. The two real-time algorithms used in this study are:

Earliest deadline (ED) - The highest priority is assigned to the task with the earliest deadline. This has been shown to be optimal [Der74] in systems with no overload. In other words, if a group of tasks can be scheduled by any algorithm (without preemption) so that all of their deadlines are met, they will be schedulable with earliest deadline as well. ED with preemption is optimal within the class of all preemptive algorithms.

Least Slack (LS) - The highest priority is assigned to the task with the least slack. Slack is determined statically at task arrival, and is not adjusted thereafter. Least slack was shown to be optimal in [MD78].

In the remainder of this section we describe real-time scheduling in a GPOS. There are two components for this: the priority assignment algorithms (to assign a priority level based on deadline or slack) and the OS policy for scheduling jobs within a priority queue. These two components are orthogonal, and subsequent sections will examine how the underlying OS scheduling policy affects the choice of priority assignment algorithms.

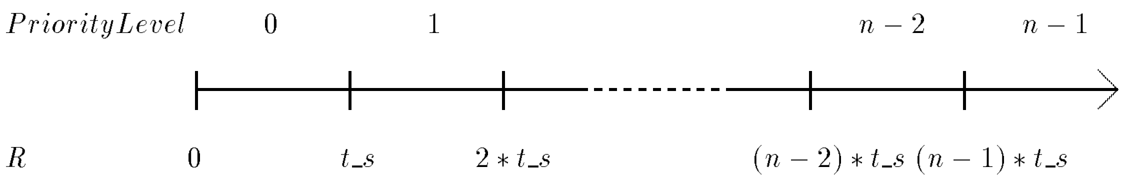

Figure 1: Conversion of a Real to a priority level

5.1 Assigning Priorities

The algorithms for assigning priorities are described below. Section 5.1.6 will provide an example of five tasks scheduled with each of the three emulation algorithms as well as with ED and LS.

5.1.1 Converting Reals to Priorities

All of the priority assignment algorithms below convert a real number, which we call R, into an integral priority level to be assigned to a task. (What R means depends on the algorithm. See below.) The mapping function, called $map_{lin}(R)$, is defined below, and referenced in all of the emulation algorithms to follow.

$map_{lin}(R)$
    $p = \lfloor \frac{R}{t_s} \rfloor$;
    IF $p \geq n$ **THEN**
        $p = n - 1$;
    ENDIF
    return(p);

This algorithm employs a linear mapping, which is denoted by the subscript lin, dividing the range of R from 0 to nt_s evenly across the n priority levels. This is shown in Figure 1. The value t_s, which is the scaling factor for the linear mapping, should be picked so that $max(R) = nt_s$. Since the maximum value of R will be vary by priority assignment algorithm and possibly by the task parameter distributions, t_s must be tuned for each situation. Tuning is discussed for each algorithm in Section 7.

5.1.2 Emulating Earliest Deadline

The next two algorithms attempt to emulate earliest deadline scheduling with a GPOS scheduler. The goal is to assign tasks with earlier deadlines to the lower levels and tasks with later deadlines to higher levels. In practice, this is a difficult problem. What should the level of the first task to arrive to an empty queue be set at? If it is set too low, and many tasks arrive with earlier deadlines, we will run out of levels to use. Setting it too high results in an analogous problem. One approach would be to assign the middle level ($\frac{n}{2}$) to the first task. The next task could be assigned to either level $\frac{n}{4}$ or $\frac{3n}{4}$, depending on its deadlines relative to the first task. As each new task arrives, it could be placed in the middle of the two other tasks which bracket it. The problem with this approach is that the number of priority levels required to guarantee placement for all tasks grows exponentially: 64 levels only allows for 6 tasks. In addition, tasks with later arrivals will also tend to have later deadlines, leading to an imbalanced tree weighted toward the higher priorities levels.

One solution to these problems would be to change the priority of tasks *after* their arrival. If a new task arrives and no priority level is available for it, it may be possible to shift existing tasks up or down to create an open level. While such an approach offers promise in solving the problems we encountered in our first solution, it may present new problems of its own. On many systems, changing the priority of a running task is difficult for two reasons:

- The priority of a running task can only be changed by the process that created it. In such a system, an outside process, like a scheduling daemon, would be unable to change task priorities.
- The system call to change priorities can only change one task at a time. If many tasks have to be shifted down in order to create an empty level, many calls will have to be made to the system; This could seriously degrade the performance of the scheduler.

Thus it is not always practical or possible to change priorities once a task is running. For this study, we chose to focus only on setting the priority of tasks when they are created, and do not attempt to change their priority at any future point. The next two algorithms are attempts to intelligently assign priorities given this constraint.

5.1.3 Earliest Deadline Relative (EDREL)

Instead of assigning priorities based on the priorities that have already been assigned, this algorithm treats each task independently from the others in the system. It calculates a priority level based on a task's deadline relative to its arrival time.

New Task Arrival:

$p_{os} = map_{lin}(d(T) - a(T));$

The independence of tasks frees the scheduler from creating an absolute scale for priorities which allows decentralized scheduling. The algorithm only uses t_s (the normalization constant in map_{lin}) and does not use information on other active tasks. A disadvantage is that the priority computation is static, thus it will probably discriminate against jobs with more distant relative deadlines, since such jobs will be given high levels and will never move; Even as their absolute deadlines approach, new jobs with tighter relative deadlines will receive higher priority.

5.1.4 Earliest Deadline Absolute (EDABS)

This algorithm uses the arrival time of the first task to set all future priority levels within the current busy period. A *busy period* is a contiguous segment of time in which a system is busy performing work. The time between busy periods is defined as an *idle period.* Clearly, a lightly loaded system will experience a series of long idle periods punctuated by an occasional short busy period, whereas a highly loaded system will experience the opposite work pattern.

In EDABS, at the start of a busy period t_{pinned} will be set to t, the current time. For the remainder of the busy period, priorities will be assigned based on a task's deadline with relation to t_{pinned}. If the busy period is long enough, later tasks will tend to be placed in higher and higher levels until all new tasks are being placed in the highest level. When $reshift\_count$ (N_r) tasks in a row get placed in the highest level, t_{pinned} is reset to the current t so that tasks will start being added from the lowest levels again. We only consider *consecutive* assignments so that a task with an unusually distant deadline does not force a reshift when many priority levels are still available. An example of this algorithm in use is given in Section 5.1.6.

Initialization

$pinned = false;$

New Task Arrival

IF $(not\ pinned)$ **THEN**
$pinned = true;$
$t_{pinned} = t;$
$max\_assigns = 0;$
ENDIF
$p_{os} = map_{lin}(d(T) - t_{pinned});$
IF $p_{os} = n - 1$ **THEN**
$max\_assigns = max\_assigns + 1;$
ELSE
$max\_assigns = 0;$
ENDIF
IF $max\_assigns = N_r$ **THEN**
$t_{pinned} = t;$
$p_{os} = map_{lin}(d(T) - t_{pinned});$
$max\_assigns = 0;$
ENDIF

Task Completion

IF $(queueisempty)$ **THEN**
$pinned = false;$
ENDIF

When we perform a reshift, old tasks will have incorrect priorities relative to the tasks arriving after the reshift. New tasks will have their priorities calculated with a more recent t_{pinned} so that even if their deadlines are later than those of the remaining old tasks, they will probably be assigned to lower levels. An example of this is given in Section 5.1.6. These old tasks are not aborted (they will be executed as the queues empty) but are likely to miss their deadlines. Intuitively, when a reshift occurs, we have detected a system overload; the idea is to start from scratch, giving up hope for old transactions. Hopefully, reshifts will be rare. The rate of reshifts is very dependent on two values: the average system load, ρ, and the reshift count, N_r. The system load is not under direct control so reshift rates must be controlled by selecting a proper value of N_r. Tuning N_r is discussed in Section 7.5.

	a	x	l	d
A	0	2	1	3
B	1	2	4	7
C	2	1	1	4
D	3	4	2	9
E	5	1	4	10

Table 1: Example Scheduler Tasks

5.1.5 Least Slack Relative (LSREL)

This algorithm attempts to emulate least slack scheduling with the Unix scheduler. Priority levels are determined based on a task's slack at arrival.

New Task Arrival

$p_{os} = map_{lin}(d(T) - a(T) - x(T));$

5.1.6 Scheduling Example

Table 1 lists five tasks and associated parameters. Variables a, x, l, and d are the arrival time, execution requirement, laxity, and deadline, respectively. The behavior of the different priority assignment algorithms is illustrated in Figure 2, which shows the queue(s) for each algorithm at time intervals of 1 unit. The example assumes that the underlying GPOS is using a FIFO scheduling policy for each priority level. (See Section 5.2.) The state of the queues shown at each time point are the states after scheduling has been completed. The number of priority levels available in the system is 4. The $reshift\_count$, N_r, is 2. The tuning factor t_s is also 2. For comparison, Figure 2 also shows two real-time schedulers, ED and LS, and how they perform. The completion times of the tasks under all five schemes are summarized in Table 2.

An example should aid in interpreting Figure 2. Picking LSREL as the algorithm to follow, look at time 0. Task A has been assigned $p_{os} = \lfloor \frac{l(A)}{t_s} \rfloor = \lfloor \frac{1}{2} \rfloor = 0$. The asterisk after A signifies that it is the running task. At time 1, task B has entered the system. Based on its slack of 4, it has been assigned p_{os} of 2 ($\lfloor \frac{4}{2} \rfloor$). A is still the running task; the $A(1)$ indicates that it has been running for 1 time unit already. At time 2, task A is finally complete (indicated by '-'), having run for 2 units. Task C has entered the system, and has been assigned $p_{os} = 0$. Because $p_{os}(C) < p_{os}(B)$, the scheduler has chosen C for execution. The remainder of the figure can be

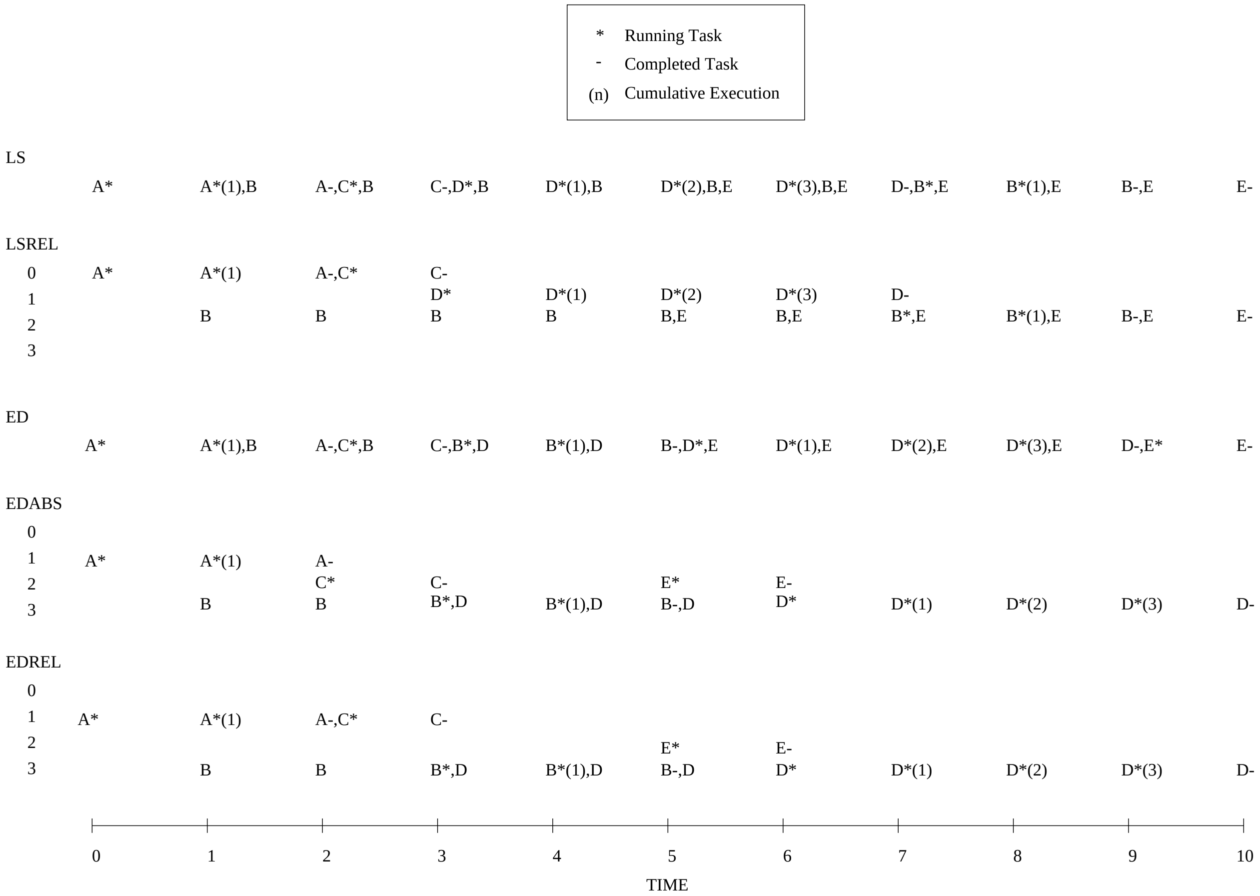

Figure 2: Comparison of Scheduling Algorithms in Non-preemptive Case

interpreted the same way.

One noteworthy incident occurs at time 5 for the EDABS algorithm. When task E arrives, it should be assigned to the last queue. This would mean that two tasks in a row had been assigned to the last queue. Since for this example $N_r = 2$, task E causes a reshift to occur. Before time 5, $t_{pinned} = 0$, which was the beginning of the busy period. At time 5, after the reshift, $t_{pinned} = 5$. This causes $p_{os}(E) = \lfloor \frac{10-5}{2} \rfloor = 2$. The level of D, which was computed at time 3, is not recomputed. Now we have $p_{os}(E) < p_{os}(D)$ but $d(E) > d(D)$, so the priorities of D and E have become inverted.

The only algorithm which successfully scheduled all of the tasks was earliest deadline. Under both EDABS and EDREL, task D missed its deadline; and under LS and LSREL, task B missed its deadline. Although some of the algorithms in this example have identical behavior, this is an artifact of the particular numbers chosen, and is not generally true.

	completion time					
	ED	EDABS	EDREL	LS	LSREL	**Deadline**
A	2	2	2	2	2	**3**
B	5	5	5	*9*	*9*	**7**
C	3	3	3	3	3	**4**
D	9	*10*	*10*	7	7	**9**
E	10	6	6	10	10	**10**

Table 2: Example Scheduler Results

5.2 OS Scheduling Policy

The previous section discussed how to assign a numerical priority to a real-time task. In this section, we examine different options for task scheduling based on priority. The decision of how to use priority levels to assign CPU time will determine how effectively the proposed algorithms are able to emulate their real-time counterparts. There are two orthogonal issues involved in OS scheduling: *preemption* and *intralevel scheduling*. *Preemption* can best be described using the following scenario. Suppose that task A is currently running on the system when task B arrives, and that $p_{os}(B) < p_{os}(A)$. On a non-preemptive system, task B will have to wait for task A to finish before it can start running. If the system is preemptive, however, task A will be suspended and task B will be run immediately. By using preemption, a higher priority task is not held up waiting for a lower priority task. Depending on the OS and on what the running code is doing, sometimes there is a small time lag from the arrival of a higher priority job until preemption. A system that has no time lag is said to be *fully preemptive*. For this study, OSs will either be non-preemptive or fully preemptive.

We have previously stated that when the scheduler determines the next process to run, it will always choose a process from the lowest occupied level. We will refer to the question of which process to select from the chosen queue as *Intralevel scheduling*. Two policies are described below. The first is used as a baseline and the second is used in Posix Unix.

5.2.1 First In First Out (FIFO)

In FIFO, separate queues are maintained for each priority level. Tasks in the lower level queues will be run before tasks in higher level queues. Within a single queue, execution will not be shared. The first task to have arrived with a particular level will be run to completion before any other tasks of the same priority are run at all. [3]

5.2.2 Round Robin (RR)

In RR, separate queues are maintained for each priority level. Only tasks from the lowest occupied level queue will be run, but they will be run round robin within the queue, each getting *schslice* seconds at a time. This is the policy used in the Posix standard for real-time jobs.

6 Simulation Model

In order to study and contrast system behavior for the three priority assignment algorithms, we developed a simulation model and performed extensive experiments. In this section we describe the model; our results are presented in Section 7.

The behavior of a traditional operating system can be very complex. Even if no other user tasks are being run besides those used in the real-time application, kernel tasks can have unpredictable interaction with the system. If we model all this diversity and complexity, fundamental insights on RTE would be obscured by many secondary effects. Instead, we chose a very simple model that captures the essential features that impact RTE. To pick a particular example, in some experiments we assume that the priority determination algorithm requires no time to run. Clearly this is not realistic. If we wanted to predict the exact absolute performance of a particular algorithm, this assumption would not be acceptable. But if we are trying to understand how the algorithm perform relative to each other, this assumption is reasonable. Our goal is to select a small number of simple, key parameters which are rich enough to illuminate the fundamental differences of the algorithms without clouding the results with uninteresting detail.

Our simulator is written in the simulation language **DeNet**[Liv90]. Each simulation experiment (generating one data point) consists of a simulation run lasting either 100,000 time units or until the 95% confidence interval for MD is within 1% of its value. For all of the MD values stated in later sections, the largest absolute error is 0.35%.

The structure of our simulation model follows the conceptual model described in Section 3 with the following characteristics. Task arrival is modeled as a Poisson process with arrival rate

[3] Although if the scheduler uses preemption, it may be temporarily preempted by a lower level task.

Parameter	Value
μ	0.5
σ	0.1
$[S_{min}, S_{max}]$	[0.1,1.0]

Table 3: Task baseline settings

Description	Parameter	Value
Intralevel scheduling policy	$OSscheduler$	$FIFO$
# of priority levels	n	8
turns preemption on	*preemption*	true
reshift count	N_r	1
time (s) between calls to scheduler	*schslice*	1.0
time slice (s) used by map_{lin}	t_s	-

Table 4: Scheduler baseline settings

λ. The execution requirements for tasks are normally distributed, with mean μ and standard deviation σ. Although exponentially distributed execution times would make analytical work more tractable, we feel that the normal distribution more accurately reflects the job mix in many soft real-time system: An exponential distribution would occasionally produce long tasks, which are shunned by real-time application programmers because of the increased resource conflicts they create. The final G refers to the slack distribution, which is uniformly distributed in $[S_{min}, S_{max}]$. In the graphs that follow, changes in performance versus λ are never shown. Instead, we use ρ, the average system utilization, which is defined as $\rho = \mu\lambda$ $(0 \leq \rho \leq 1)$.

Tables 3, 4 and 5 show the parameter setting of our baseline experiment. To study the effect of these parameters on system performance, we will vary the parameters from their base settings. This is discussed in the following section. The values for parameter t_s are given in Table 5 since they depend on the priority assignment algorithms and on the value of n, the number of priority levels. These values were tuned for t_s in each situation from many simulation runs. We delay discussing tuning in detail until Section 7.5, but still use the tuned values now to allow the performance of the emulation algorithms to be compared more accurately to the performance of the real-time algorithms.

Algorithm	n			
	8	16	32	128
$EDABS$	0.3	0.3	0.3	0.3
$EDREL$	0.2	0.1	0.05	0.0125
$LSREL$	0.15	0.075	0.0375	0.0094

Table 5: t_s baseline values

7 Results

In this section, we summarize the results of our simulation experiments. As performance measure we use the percentage of missed deadlines (or miss ratio).[4] For the real-time algorithms, we represent the percentage of missed deadlines by MD_A, $A \in \{EA, ED, LS\}$. For the real-time emulation algorithms, we use MD_A^B where $A \in \{EDABS, EDREL, LSREL\}$ is the priority assignment scheme and $B \in \{FIFO, RR\}$ is the OS scheduling policy in use. For example, MD_{EDREL}^{RR} denotes the probability that a task misses its deadline with priority assigned under the $EDREL$ strategy on an operating system using round robin scheduling.

7.1 Baseline Experiment

As a starting point, let us look at how the various strategies do relative to each other in our baseline experiment but with no preemption. The parameters for this experiment can be found in Table 4. Figure 3 shows MD for the priority assignment schemes and the real-time algorithms as $load$ varies from 0.15 to 0.55. By $load$ we are referring to the average system utilization, defined as $\rho = \mu\lambda$. For loads less than 0.32, the performances of the algorithms are virtually indistinguishable. For higher loads, ED performs worse than the other four algorithms. Two interesting conclusions can be drawn from these observations. First, LSREL tracks LS so closely under the conditions of this experiment that the two curves are indistinguishable. Second, EDABS and EDREL both either equal or outperform ED across the entire load range. It is encouraging to see that the emulation algorithms are performing as well as or better than the real-time algorithms themselves.

We should point out that traditionally soft real-time systems are studied under high load situations. Hopefully, the system will mostly operate under low load in practice; no deadlines will be missed regardless of what scheduling policy is used. Unfortunately, occasionally the

[4] A secondary measure could be tardiness, i.e., by how much time do tasks miss their deadlines. However, due to space limitations, we do not consider this measure.

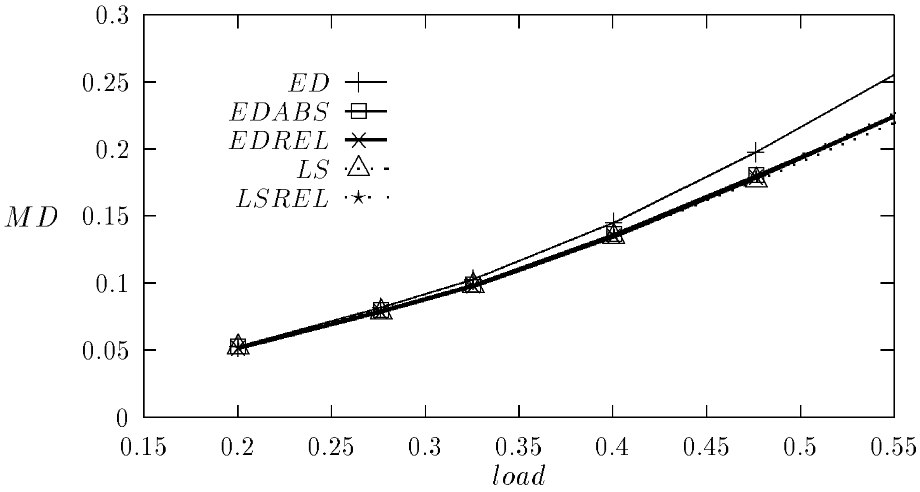

Figure 3: Performance of priority assignment strategies in the baseline experiment.

system will be overloaded, and it is precisely at those times when we need a scheduling policy that can miss the fewest deadlines. For this reason, the big differences in missed deadlines under high load in Figure 3 are important.

7.1.1 Preemption

Figure 4 shows MD for the priority assignment schemes as *load* varies from 0.15 to 0.55, with preemption on. The performance of all of the algorithms improve compared to the baseline experiment. When we compare MD rates between different algorithms or between different parameter sets, we will always refer to absolute differences in missed deadline rate. A difference of 3% between two MD values means that $|MD_1 - MD_2| = 3\%$. For this experiment, at the highest loading the performance of LS and of the three emulation algorithms show the most improvement, about 5%. In the lower loading range, ED and EDABS show the most improvement, outperforming the other algorithms by 2%. The best algorithm across the entire range is EDABS, which is a close second to ED under low load and the best performer under high load.

As in the case with no preemption, this result is encouraging because although we knew that the emulation of the real-time algorithms would not be perfect, we did not expect the emulation algorithms to outperform the real-time algorithms. In the case of EDABS, under high load the algorithm must occasionally reshift. When a reshift occurs, the tasks currently in the higher levels will be delayed by new tasks which arrive and are placed in the lower queues. Although the older tasks will probably miss their deadlines due to the delay, sacrificing them allows the system to start afresh with the newly arriving tasks so that the system as a whole misses fewer deadlines.

As shown by Figure 4a, however, the low load situation is different. Here ED slightly outperforms EDABS. Under low load, very few reshifts occur, so EDABS loses the advantage described above. If there were no other differences between ED and EDABS, we would expect the two curves to be identical in the low load range. The reason ED outperforms EDABS is that ED is able to distinguish between tasks with similar deadlines whereas EDABS sometimes has to group tasks with similar deadlines into the same level. Consider two tasks, task A and task B, where task A arrives before task B. If $|d(B) - d(A)| < t_s$, tasks A and B could be assigned to the same level. If $d(B) < d(A)$, however, this will result in priority inversion under FIFO scheduling since task A will be executed first but task B has an earlier deadline. This effect can be minimized by increasing n or by decreasing t_s (although this has negative repercussions under high load).

LSREL also demonstrates surprising behavior, outperforming LS across the entire load range in Figure 4, though only slightly. We hypothesize that this effect is the result of a limited number of priority levels which draws the performance of LSREL away from LS and towards ED. The limited number of levels (n) forces LSREL to group tasks with similar slack into the same queue. For the simulation whose results are shown, FIFO was used as the intralevel scheduling policy. This led to approximately ED scheduling within a level, since tasks with earlier arrivals tend to have earlier deadlines. The effect of combining both scheduling algorithms gives LSREL an MD curve between those of ED and LS.

7.2 Effect of OS Scheduling

Figure 5 shows MD for the priority assignment schemes for round-robin scheduling as the scheduling slice, $schslice$, is varied (and $n = 16$). The performance of the algorithms under FIFO scheduling is also plotted as a basis for comparison. The curves for all three emulation algorithms have nearly the same shape. When $schslice \leq \mu$, the MD rate under RR is much higher than the rate under FIFO scheduling. This is because even average length tasks are being preempted at least once before they can complete, increasing their service time. To understand this effect, consider three tasks A,B, and C which arrive in the system at time $t = 0$. Further assume that each task requires 3 seconds to execute, and each has a deadline of $t = 8$. Under

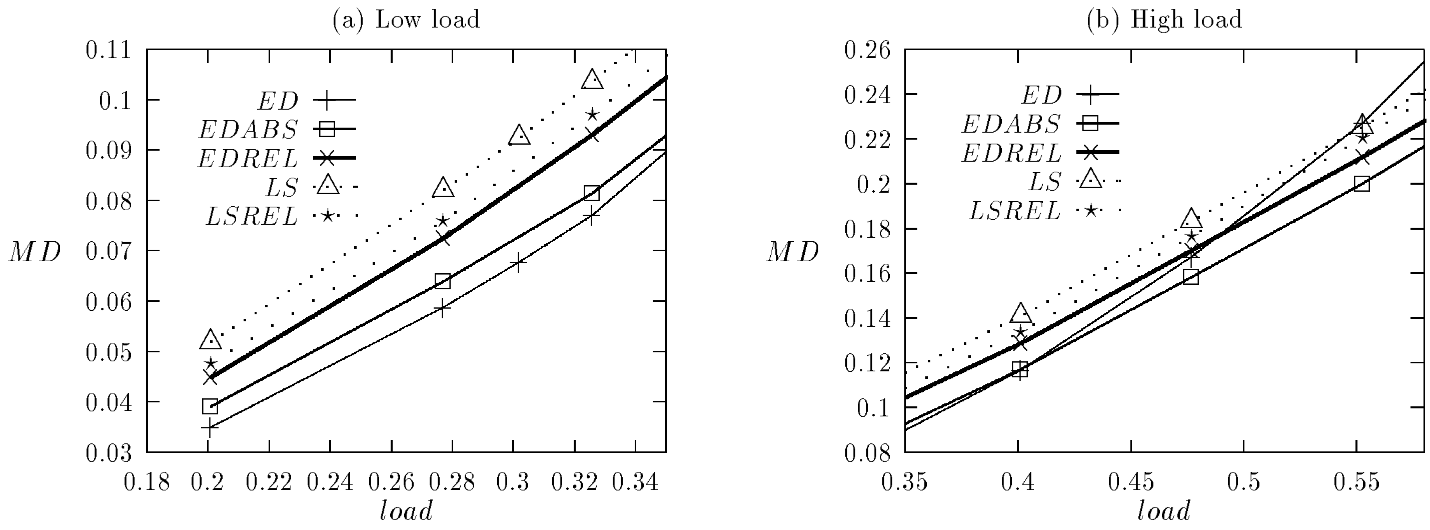

Figure 4: Performance of priority assignment strategies with preemption.

FIFO, or RR with *schslice* $\gg 3$, one task will be run to completion. At $t = 3$, another task will be started and run to completion. Finally, at $t = 6$, the last task will be started and run to completion. In this schedule, the first two tasks both meet their deadlines, but the last task finishes at $t = 9$, missing its deadline. Under RR with *schslice* $\ll 1$, the system will constantly rotate between running jobs so that at $t = 8$, each will have run for $2\frac{2}{3}$ seconds, and all will miss their deadlines.

As the ratio $\frac{schslice}{\mu}$ increases above 1, MD falls towards the FIFO value since fewer tasks are being preempted. For the simulation runs that generated these graphs, no jobs had execution times of over 2μ. From this we might have assumed that for $\frac{schslice}{\mu} > 2$, no jobs would be interrupted by the scheduler and so MD^{RR} should equal MD^{FIFO}. In some systems this might be the case. If the scheduling timer is started when a new task is started, MD rates would be equal for $\frac{schslice}{\mu} > 2$. In our simulation, however, we modeled what many operating systems do for simplicity: they program the timer to interrupt every *schslice* seconds, completely discoupled from the start times of any jobs. In a design like this, if one job finishes half-way through a time slice, the next job will be interrupted if it requires more than the $\frac{1}{2}$ of the slice which remains. Because of this effect, round robin will never quite approach the performance of single queue. In conclusion, round robin scheduling is detrimental to real-time emulation. If it cannot be disabled, the *schslice* should be set as large as possible: If *schslice* can be set at 3 to 4 times μ, the performance penalty for round robin scheduling will be tolerable for most applications.

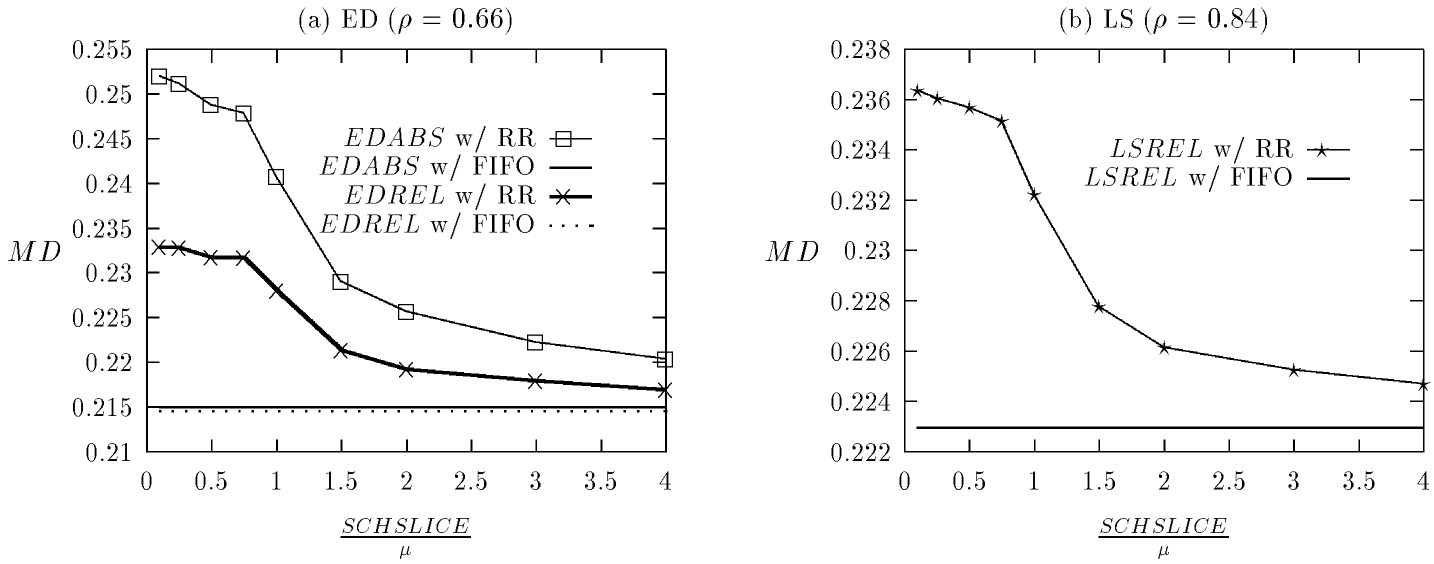

Figure 5: Effects of Round-Robin OS Scheduling on performance.

7.3 Effect of Number of Priority Levels (n)

Figure 6 shows MD for the priority assignment schemes as n is varied from 8 to 32. [5] In Figure 6a, we can see that as n is increased, EDABS emulates ED more closely, whereas EDREL does not change significantly. It should be pointed out, however, that a close emulation of a real-time algorithm is not always desirable. As an example, consider EDABS. Under low system load, as in this example, ED outperforms EDABS, so increasing n results in improved EDABS performance. Conversely, under high load (Figure 4b), EDABS outperforms ED, so increasing n degrades the performance of EDABS as it nudges it closer to that of ED. The choice of n should depend on the relative performance of the emulation algorithm to the real-time algorithm in the load range of interest.

The performance of LSREL is shown in Figure 6b. Even by 16 priority levels, LSREL is already emulating LS to within 0.2%. From these simulation runs and others, it is clear that 32 priority levels will provide adequate emulation in many cases, and that by 128 priority levels, the emulation is extremely good (see Section 7.4). Thus the number of priority levels provided by current operating systems is sufficient to allow excellent emulation.

[5] Some parameters vary from the baselines setting: $\mu = 0.12, \sigma = 0.01$.

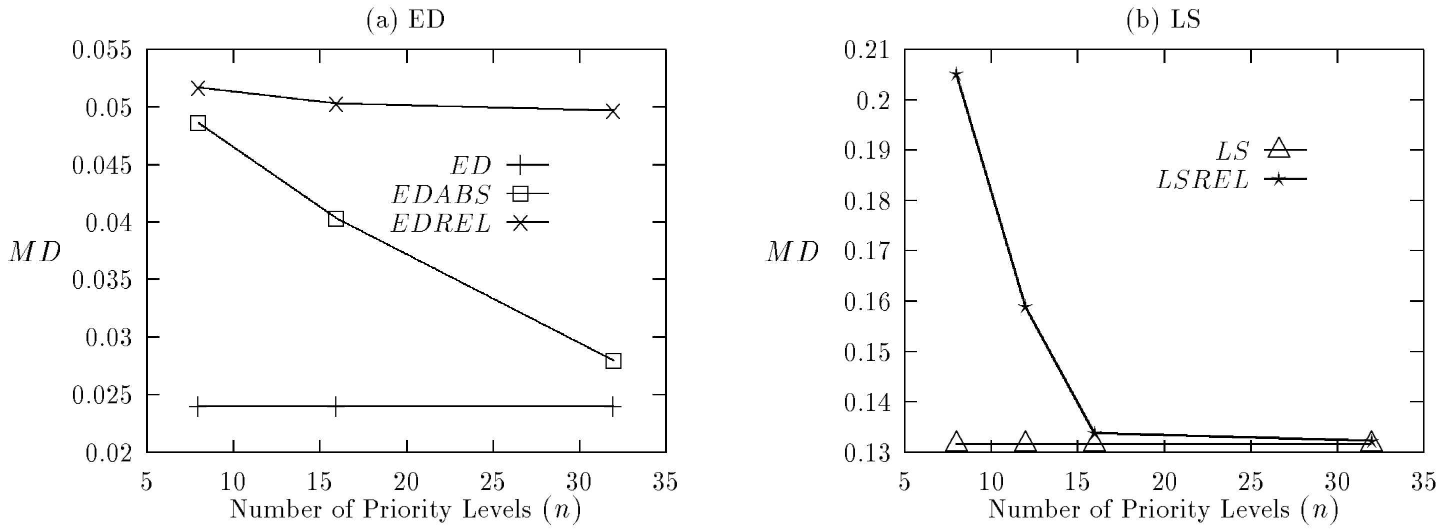

Figure 6: Effect of number of priority levels on performance.

7.4 Overall Performance

Figure 7 shows the 'error' in MD introduced by emulating ED and LS using parameters representing a typical GPOS [6]. By 'error' we mean the difference in measured MD values between an emulation algorithm and the real-time algorithm it is trying to emulate. Note that although we call this difference an error, if $e_{MD} < 0$ the emulation algorithm is outperforming the real-time algorithm, thus the error is in our favor. In Figure 7, $EDABS$ shows the closest tracking to ED: MD_{ED} and MD_{EDABS} never differ by more than 0.5%. $EDREL$ does not emulate ED as well, the two are off by as much as 2.5% in MD, but under higher load $EDREL$ slightly outperforms ED. $LSREL$ tracks LS to within the confidence interval of the simulations: MD_{LS} and MD_{LSREL} never vary by more than 0.25%. These experiments demonstrate that even under a commercial GPOS, both $EDABS$ and $LSREL$ emulate their respective real-time algorithms very closely.

7.5 Tuning

One negative of the priority assignment algorithms is that to get good emulation it is necessary to tune their parameters for the particulars of the system. Below we discuss how to adjust both the reshift count, N_r, and the scaling factor, t_s. The recommendations that follow are based

[6] Some parameters vary from the baseline setting: $OSscheduler = RR$, $n = 128$.

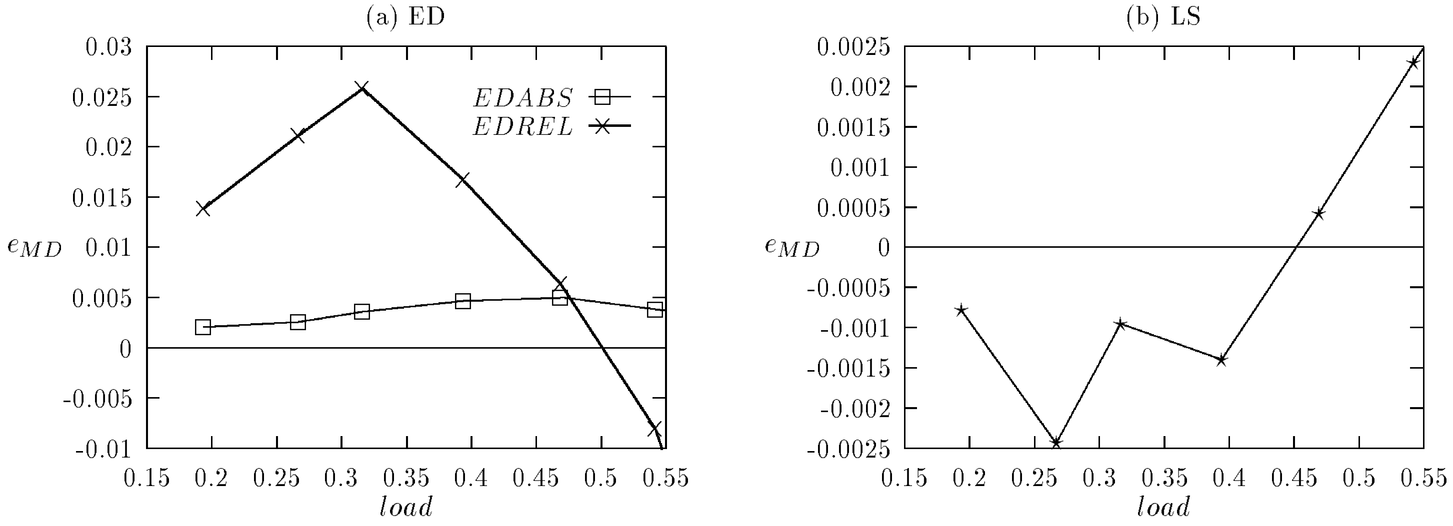

Figure 7: MD Error for Emulation Algorithms

on additional experiments [AGMK] not described in this paper due to space constraints.

The parameter N_r is only used by $EDABS$. EDABS uses N_r to determine when to reset t_{pinned}, which we call a reshift. No single value of N_r optimizes the algorithm's performance under all load conditions. At low loads, using a higher value for N_r lowers MD, but the situation is reversed under high load. In high load cases, it is better to reshift after the first task is assigned to the last level ($N_r = 1$) so that overload conditions are detected early. As a compromise in the general case, we suggest $N_r = 1$ or 2, since any higher value will only negligibly improve low load performance but will greatly damage high load performance.

Tuning t_s is a necessity for all three priority assignment algorithms. As with N_r, we again have the problems of picking a single value of t_s that optimizes MD across all load conditions. The best value of t_s will have to be determined by experimentation, but we provide rules of thumb for two of the three emulation algorithms. For EDREL, the value passed to map_{lin} is $d(T) - a(T)$. We might assume that we want to pick t_s so that it divides $max(d(T) - a(T))$ evenly into n regions. If we use the approximation

$$max(d(T) - a(T)) = max(x(T) + l(T)) \approx \mu + k\sigma + S_{max}$$

then we have

$$nt_s = max(d(T) - a(T)) \approx \mu + k\sigma + S_{max}$$

or

$$t_s \approx \frac{\mu + k\sigma + S_{max}}{n}$$

where k is a constant signifying how many standard deviations of μ we want to include in $max(d(T) - a(T))$. This approximation (using $k = 3$) showed good agreement with experimental results [AGMK].

An intuitive approach for tuning $LSREL$ is to make t_s divide the maximum slack into n equal intervals, such that $nt_s \approx S_{max}$. Experimental results indicate that the resulting t_s will be much better for high load conditions than for low load conditions. Still, without further information about the load on a particular system, $t_s = \frac{S_{max}}{n}$ is a good first approximation.

The necessity of tuning algorithm parameters through experimentation or simulation is undeniably a weakness of our approach. Still, even with parameters set using our rules of thumb, performance can be very good over wide ranges of parameter values [AGMK]. A possible further improvement would be to design an adaptive algorithm that adjusts t_s as the systems workload changes. We do not investigated such an approach in this study, but we do believe that it is an interesting area of future study.

8 Conclusions

This paper considered the problem of emulating soft real-time scheduling with a general purpose operating system. Through simulation, we examined the performance of algorithms to emulate both earliest deadline first scheduling and least slack first scheduling. Surprisingly, not only did the emulation algorithms we presented here track the real-time algorithms very well, they outperformed them in many cases.

EDABS had the best overall performance. It was close to ED for low load conditions, and was usually the best algorithm for high load conditions as well. Both EDABS and LSREL emulated their respective real-time counterparts well, and improved as the number of priorities was increased.

Finally, we would like to remark that the RTE problem is an important one in the design of the real-time applications of the future. These applications will increasingly be developed on standard hardware running GPOSs and connected by networks that have no notions of real-time. The ability to convert parameters like deadlines and slack into priority levels will be essential if soft real-time is to develop into a more general model for application design.

References

[AGM88a] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions. In *ACM SIGMOD Record*, pages 1–12, 1988.

[AGM88b] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: a performance evaluation. In *Procceedings of the 14th VLDB Conference*, 1988.

[AGM90] R. Abbott and H. Garcia-Molina. Scheduling I/O requests with deadlines: a performance evaluation. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 113–124, 1990.

[AGM92] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *IEEE Transactions on Database Systems*, 17(3):513–560, 1992.

[AGMK] B. Adelberg, H. Garcia-Molina, and B. Kao. Emulating soft real-time scheduling using traditional operating system schedulers. Stanford University Technical Report. Available by anonymous ftp from db.stanford.edu in /pub/adelberg/1994.

[AGMK94] B. Adelberg, H. Garcia-Molina, and B. Kao. A real-time database system for telecommunication applications. In *Proceedings of the 2nd International Conference on Telecommunication Systems Modelling and Analysis*, 1994.

[Cra88] B. Cramer. Writing real-time programs under unix. *Dr. Dobb's Journal*, 13(6):18–33, 1988.

[CW90] D. Craig and C. Woodside. The rejection rate for tasks with random arrivals, deadlines, and preemptive scheduling. *IEEE Transactions on Software Engineering*, 16(10):1198–1208, 1990.

[Der74] M. L. Dertouzos. Control robotics: The procedural control of physical process. In *Proceedings of the IFIP Congress*, 1974.

[FPG+89] B. Furht, J. Parker, D. Grostick, H. Ohel, T. Kapish, T. Zuccarelli, and O. Perdomo. Performance of Real/IX - fully preemptive real time unix. *Operating Systems Review*, 23(4):45–52, 1989.

[HTT89] J. Hong, X. Tan, and D. Towsley. A performance analysis of minimum laxity and earliest deadline scheduling in a real-time system. *IEEE Transactions on Computers*, 38(12):1736–1744, 1989.

[Liv90] M. Livny. **DeNet** user's guide. Technical report, University of Wisconsin-Madison, 1990.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[MD78] A. K. Mok and M. L. Dertouzos. Multiprocessor scheduling in a hard real-time environment. In *Proceedings of the Seventh Texas Conference on Computing Systems*, 1978.

[MT89] Y. Mizuhashi and M. Teramoto. Real-time operating system: RX-UX 832. *Microprocessing and Microprogramming*, 27:533–38, 1989.

[SLR86] L. Sha, J. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 181–191, 1986.

[Wel93] G. Wells. A comparison of four microcomputer operating systems. *Real-Time Systems*, 5:345–368, 1993.