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

Brad Adelberg† Hector Garcia-Molina‡ Ben Kao§

April 26, 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

Consider program trading, the use of computer programs to initiate trades in a financial market with little or no human intervention [Voe87]. A financial market (e.g., a stock market) is a complex process whose state is partially captured by variables such as current stock prices, changes in stock prices, volume of trading, trends, and composite indexes. These variables and others can be stored and organized in a database to model a financial market.

One type of process in this system is a sensor/input process which monitors the state of the physical system (i.e. the stock market) and updates the database with new information. If the database is to contain an accurate representation of the current market then this monitoring

*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

process must meet certain real-time constraints. A second type of process is an analysis/output process. In general terms this process reads and analyzes database information in order to respond to a user query or to initiate a trade in the stock market. An example of this is a query to discover the current bid and ask prices of a particular stock. This query may have a real-time response requirement of say 3 seconds. Another example is a program that searches the database for arbitrage opportunities. Arbitrage trading involves finding discrepancies in prices for objects, often on different markets. For example, an ounce of silver might sell for $10 in London and fetch $10.50 in Chicago. Price discrepancies are normally very short-lived and to exploit them one must trade large volumes on a moment's notice. Thus the detection and exploitation of these arbitrage opportunities is certainly a real-time task.

In addition to real-time tasks, the trading program will also have more traditional functions to perform. The program may have to generate reports, interact with humans through a graphical interface, etc. These functions do not require real-time support, and are actually better performed with general purpose operating systems, systems that have built a large suite of tools and applications over many years. Even if the different functions of a single application are split across two computers, it is still simpler to build the trading system for a single operating system rather than two: Buying, maintaining, and developing on two different operating systems will divert resources from application development. So then, how can the benefits of a general purpose operating system (GPOS) be combined with the need for real-time response? Answering this question is the main goal of this paper.

Our focus is on *soft real-time* (SRT) systems, such as the example above. In these systems, it is very difficult to guarantee that all deadlines will be met, and hence one tries to *minimize* the number of deadlines that are missed. Guaranteeing all deadlines is hard because it is impractical to place an upper bound on the load. In our trading example, for instance, we may want to estimate the maximum number of commodities which can change in value at once (or within some small time frame). Unfortunately, the number of changing commodities is only bounded by the total number of commodities (which can be orders of magnitude greater than the number of changing commodities). Another reason why soft real-time systems may miss deadlines is because the tasks they run are complex and it is hard to predict exactly how long they will run or what resources they may need. For instance, it is hard to know in advance how many rules the expert system in our example will trigger. In the database component, the running time of a search will depend on what other transactions are concurrently running and holding locks. Furthermore, disk access times will depend on where the previous request left the disk arm, again hard to determine in advance.

Soft real-time systems are different from hard real-time (HRT) systems. The discovery of new domains for soft real-time systems has led to research into new scheduling algorithms, different from the algorithms used in hard real-time systems (i.e. rate monotonic). Some differences are

- HRT systems typically perform scheduling off-line where as SRT systems perform scheduling on-line;
- HRT algorithms are usually optimized for periodic tasks where as SRT algorithms are designed for aperiodic tasks;
- HRT algorithms often require maximum task execution times to be known in advance where as SRT systems require at most an estimate of execution time, and often times require no knowledge at all.

Along with the renaissance in real-time scheduling algorithms, the notion of an appropriate operating system for soft real-time systems has begun to change. 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, number 0 to 4. 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 priorty 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 spectrum spans the following choices:

1. Set process priorities and then let the GPOS kernel schedule processes using its native algorithm;
2. Use a special scheduling process (daemon) which blocks all processes except the one it chooses to run, thereby forcing the operation of the GPOS scheduler;
3. Write a threads package to run on top of a GPOS with the desired scheduler in it.

To implement the first method, each process that starts a task must call a common routine to determine a priority for the new task. 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. (This problem is discussed for ED emulation in Section 5.1.2.) The obvious advantages to this scheme are that the amount of coding is minimal, and no coding changes must be made to any existing tasks. A disadvantage is that a GPOS scheduler, which often uses multi-level feedback (described in Section 5.2.3), might adjust the initial priority assignments, thereby sabotaging the intent of the priority generating algorithm.

Toward the other end of the spectrum is a method to circumvent the scheduler altogether (method 2 above). A scheduling daemon can be run as a separate process, effectively controlling all CPU access. All processes must send a message identifying themselves to the scheduler daemon when they start, and then block waiting on a reply. By only sending a reply to one process at a time, the daemon can insure that only one process is eligible to run at once, thereby forcing the GPOS scheduler to do its bidding. All synchronization must be done through the scheduler daemon since otherwise no processes would be eligible to run if the running process blocks. This scheme becomes even trickier when preemption is desired; regular preemption points need to be programmed into the running tasks. Preemption points are void calls to the scheduler daemon interspersed within the application code. Their single purpose is to allow the daemon to regain control during long execution periods when the application does not release the processor. Preemption points reduce the latency from when a high priority process arrives to when the current process can be preempted. The need for preemption points illustrates one of the chief drawbacks of the scheduler daemon approach: increased complexity. The scheduling daemon solution requires significant coding for both the daemon and the processes to be run. In addition, any error in a process could cripple the system since it requires that all participants be well behaved. There could also be a performance penalty for the frequent interaction between running processes and the daemon, both in communications overhead and in context switch overhead. The chief advantage of scheduling daemons is the excellent control that they provide in scheduling jobs.

The last option is to write a threads package on top of a GPOS. An entire application consisting of many threads of control will appear to the OS as one process. The author of the threads package will need to write machine dependent code normally provided by an OS, such as context switch code. The benefit is that the author gets to rewrite the scheduler as well, using any algorithm s/he wishes. Conversely, this approach involves the most coding and is the least portable. An advantage is that the threads (running tasks) do not have to be well behaved. In effect, an operating system must be written on top of another one, which, in addition to being a formidable task, nullifies many of the advantages of a GPOS that motivated us.

We chose the first option for further study for its simplicity and ease of implementation. Scheduling will be done by the native scheduler, with real-time algorithms emulated by setting priorities appropriately. 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.

4 Base Model

4.1 System Architecture

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. Real-time algorithms will be emulated by setting process priorities and letting the GPOS do the scheduling. We assume that the system is dedicated to the real-time 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. This model is more restictive than the situation we described in the introduction in which both types of tasks coexisted on the same machine. Some GPOSs, like Posix Unix (see Section 4.3), support a special class of processes which have higher priority than all batch processes. On a Posix system, batch and real-time processes can be run concurrently since the batch processes will run only if no real-time processes are waiting. We assume a more restrictive policy to make our approach applicable to more GPOSs.

4.2 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. Figure 1 shows two common arrangements. The numbering scheme used can be a source of confusion: the highest priority processes have the lowest priority level. To

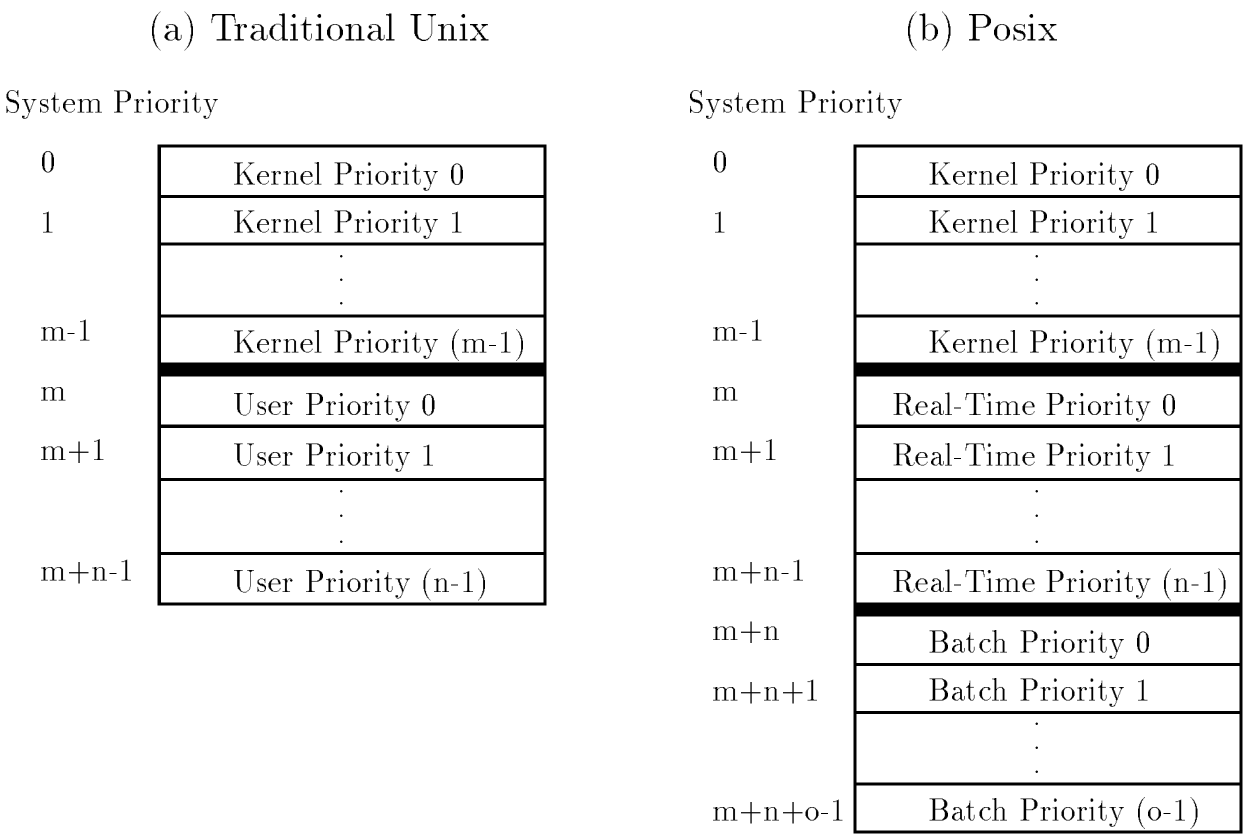

Figure 1: Operating System Priority Level Assignments

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. The relationship between p_{os} and actual system priority levels varies by operating system and is illustrated below. 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 $p_{os}(T)$.

4.3 Survey of Two Current Systems

Successfully emulating real-time scheduling will depend on the native scheduler in the traditional operating system. Even in the time share world, there is no single standard. This section outlines some of the algorithms in use in current popular GPOSs. For the systems described, n will always refer to the number of priority levels available from the perspective of the real-time emulator. The variable m is used to indicate the number of priority levels available for kernel use. Other variables will be explained below.

Standard Unix: Uses round robin multilevel feedback scheduling. This is described in general terms in Section 5.2.3. User processes will have priority levels between m and $m+n-1$ as shown in Figure 1a, which allows for n possible priority levels. Unfortunately, the user has very little control over the level of a process when it is submitted. The kernel assigns each process a base level, which is then adjusted over time as a function of recent CPU usage. The only control afforded to the user is through the *nice* command, which typically allows the base level to be adjusted by ± 20 levels. Since the *nice* command is the only method of altering a process's level, in standard Unix systems p_{os} will refer to the *nice* value range of $0 \ldots 40$ (normalized from ± 20 to be non-negative).

Posix Unix: Provides for a new process class with priority levels between kernel tasks and batch tasks called *rtprio* tasks, as shown in Figure 1b. Processes can be given priority levels typically between 0 and 127. Round robin is used within the same level, but with no multilevel feedback. Tasks from the lowest queue will be run to completion before tasks in higher queues or batch tasks are run at all. New tasks can preempt currently running tasks if they are placed in a lower queue. For Posix Unix, p_{os} will refer to the real-time priority value, usually in the range $0 \ldots 127$.

4.4 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 only and must be implemented on a real-time kernel. The three real-time algorithms used in this study are:

Earliest Arrival (EA) - The highest priority is assigned to the task with the earliest arrival time. This is the same as first come first serve. This algorithm uses no real-time information and is present only as a point of comparison.

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.

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 2. 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

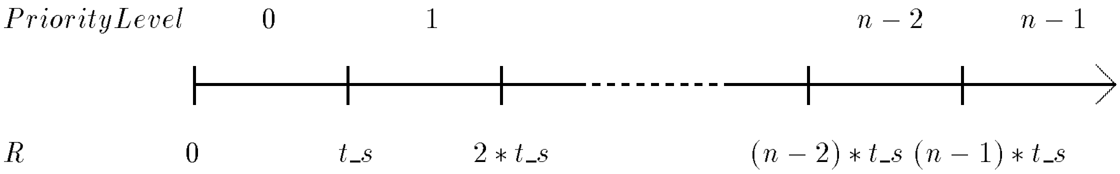

Figure 2: Conversion of a Real to a priority level

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:

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

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 it will probably discriminate against jobs with more distant deadlines, since such jobs will be given high levels and will never move; Even as their deadlines approach, new jobs with tighter 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
  $level = map_{lin}(d(T) - t_{pinned})$;
  IF $level = n - 1$ **THEN**
    $max\_assigns = max\_assigns + 1$;
  ELSE
    $max\_assigns = 0$;
  ENDIF
  IF $max\_assigns = N_r$ **THEN**
    $t_{pinned} = t$;
    $level = map_{lin}(d(T) - t_{pinned})$;
    $max\_assigns = 0$;
  ENDIF

Task Completion

  IF (Queue Is Empty) **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. 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 7.2.1 and a more general discussion of reshift rates is contained in Appendix A.

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

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

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 3, 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 3 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 3. 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,

	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

	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

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 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.

* Running Task
\- Completed Task
(n) Cumulative Execution

		0	1	2	3	4	5	6	7	8	9	10
LS		A*	A*(1),B	A-,C*,B	C-,D*,B	D*(1),B	D*(2),B,E	D*(3),B,E	D-,B*,E	B*(1),E	B-,E	E-
LSREL	0	A*	A*(1)	A-,C*	C-							
	1				D*	D*(1)	D*(2)	D*(3)	D-			
	2		B	B	B	B	B,E	B,E	B*,E	B*(1),E	B-,E	E-
	3											
ED		A*	A*(1),B	A-,C*,B	C-,B*,D	B*(1),D	B-,D*,E	D*(1),E	D*(2),E	D*(3),E	D-,E*	E-
EDABS	0											
	1	A*	A*(1)	A-								
	2			C*	C-		E*	E-				
	3		B	B	B*,D	B*(1),D	B-,D	D*	D*(1)	D*(2)	D*(3)	D-
EDREL	0											
	1	A*	A*(1)	A-,C*	C-							
	2						E*	E-				
	3		B	B	B*,D	B*(1),D	B-,D	D*	D*(1)	D*(2)	D*(3)	D-

TIME

Figure 3: Comparison of Scheduling Algorithms in Non-preemptive Case

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*. Three policies are described below. The first is used as a baseline, and the last two are both used in some version of 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 level are run at all. [1]

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

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

getting *schslice* seconds at a time. This is the policy used in the Posix standard for real-time jobs.

5.2.3 Round Robin with Multi-level Feedback (RRMLF)

RRMLF is like RR, but processes which have used a lot of CPU time recently are pushed into higher level queues to give other jobs a chance to run. This is a useful property in a conventional system, where it is better to let I/O bound jobs run at a higher priority than CPU bound jobs, but this behavior can frustrate our attempts at real-time scheduling.

The formula typically used is $level = base\_level + recent\_cpu\_usage$. The variable $base\_level$ is the level chosen by the priority assignment algorithm. The variable $recent\_cpu\_usage$ is a per process field that is incremented once for every tick that the process executes. In this study, the tick frequency parameter is f_{tick}. After every time slice, the value of $recent\_cpu\_usage$ for every process is halved. This decay allows tasks to move back into the high priority queues after being idle for a few slices.

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

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

Table 3: Task baseline settings

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 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 Sections 7.2.2 and 7.3.1, 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. For some later experiments it will be interesting to study the performance of the algorithms under conditions similar to those under an actual GPOS. Table 6 shows the parameter values which best approximate the conventional operating systems which were discussed in 4.3.

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
rate (s^{-1}) to increment cpu usage	f_{tick}	60.0
time slice (s) used by map_{lin}	t_s	-

Table 4: Scheduler baseline settings

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

Parameter	Unix	Posix
$OSscheduler$	$RRMLF$	RR
n	40	128
$preemption$	false	true
$schslice$	1.0	1.0
f_{tick}	60.0	60.0

Table 6: OS parameter settings

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).[2] 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 $B \in \{FIFO, RR, RRMLF\}$ 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

7.1.1 Base Comparisons Without Preemption

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 4 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 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 4 are important.

[2] 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.

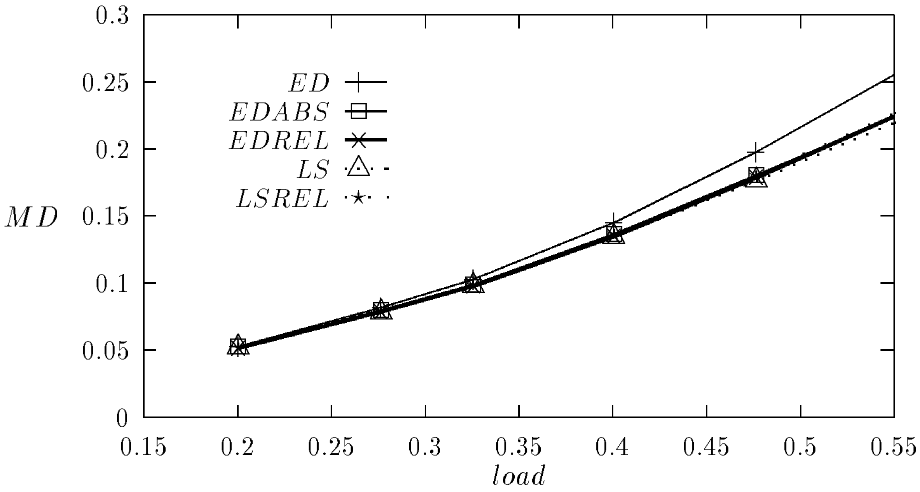

Figure 4: Performance of priority assignment strategies in the baseline experiment.

7.1.2 Preemption

Figure 5 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 5a, 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 5, 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.

7.1.3 Effect of OS Scheduling

Figure 6 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 FIFO, or RR with $schslice \gg 3$, one task will be run to completion. At $t = 3$, another task

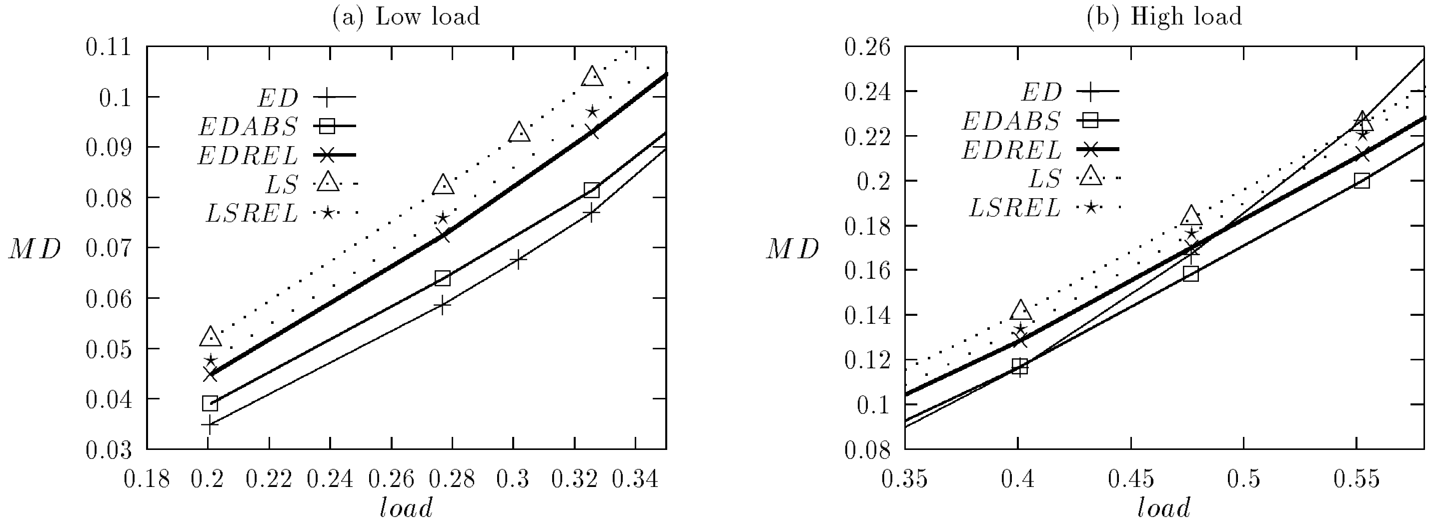

Figure 5: Performance of priority assignment strategies with preemption.

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.

The effects of round-robin scheduling with multi-level feedback are shown in Figure 7 ($n =$

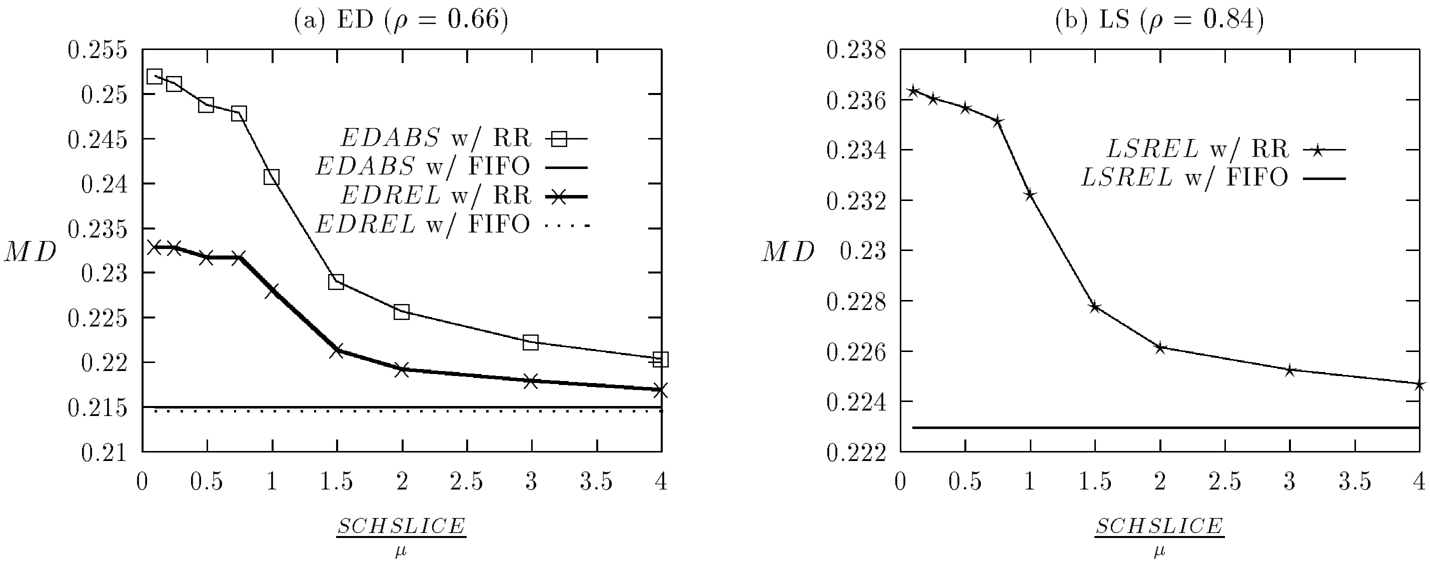

Figure 6: Effects of Round-Robin OS Scheduling on performance.

16). The performance of the algorithms under FIFO OS scheduling is also plotted as a basis for comparison. The shapes of the curves for all three algorithms are very similar. They perform well with a small ratio, degrade until $\frac{schslice}{\mu} = 1$, and then improve again as the ratio increases. A low $\frac{schslice}{\mu}$ ratio means that high priority jobs will probably be preempted frequently, but since *schslice* is small they will resume execution quickly. [3] Conversely, when the ratio is large, the cost of preemption will be high, since *schslice* is large, but preemption will be infrequent, so the unwanted preemption effect on MD will still be small. At $\frac{schslice}{\mu} = 1$, however, both effects combine to severely degrade the performance of all three algorithms. Having explained the shape of the curves, we can now evaluate the suitability of $RRMLF$ for our real-time emulation algorithms. The problem of unwanted preemption makes round-robin with multi-level feedback perform worse than round robin scheduling over the entire range of *schslice* for all three algorithms. Even as the ratio $\frac{schslice}{\mu}$ approaches 4, the algorithms are still performing 3-4% worse than under $FIFO$ scheduling. This experiment suggests that $RRMLF$ may not be a suitable scheduler for soft real-time emulation, at least not with the relatively small number of priority levels ($n = 16$) we have assumed so far.

[3]The performance of the algorithms when $\frac{schslice}{\mu} < 1$ shown in Figure 7 is nearly best case. Performance in this region is very sensitive to experimental parameters, and will often appear more like the curves in Figure 6 for round-robin scheduling.

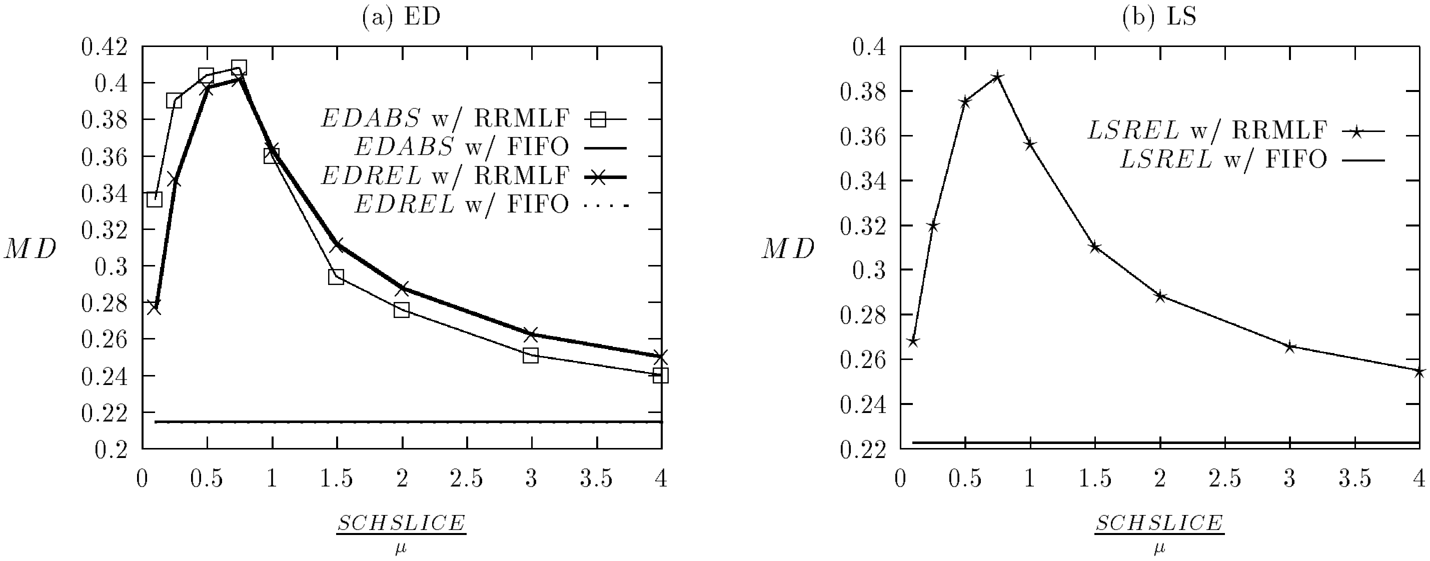

Figure 7: Effects of Round-Robin Multi-Level Feedback OS Scheduling on performance.

7.1.4 Effect of Number of Priority Levels (n)

Figure 8 shows MD for the priority assignment schemes as n is varied from 8 to 32. [4] In Figure 8a, 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 (not shown here), 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 8b. 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 Sections 7.2 and 7.3). Thus the number of priority levels provided by current operating systems is sufficient to allow excellent emulation.

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

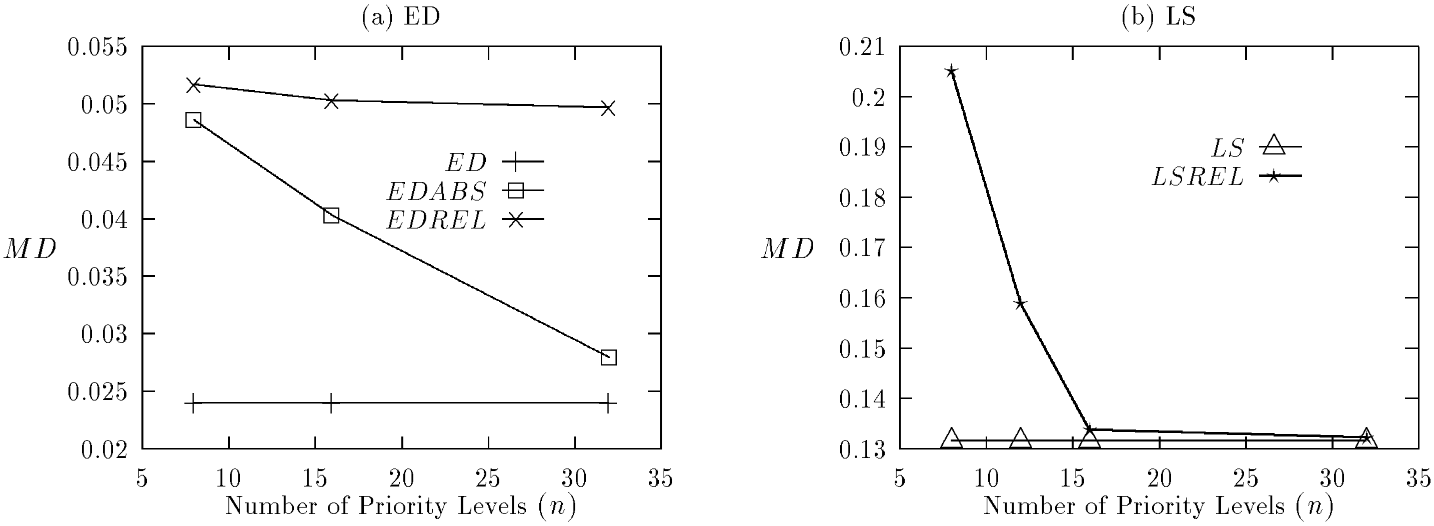

Figure 8: Effect of number of priority levels on performance.

7.1.5 Effect of Slack

Figure 9 shows MD for the priority assignment schemes as S_{max} varies from 0.1 to 1.3 with a load of $\rho = 0.4$. From Figure 9a we see that under small slack $MD_{EDREL} < MD_{EDABS} < MD_{ED}$, but that under large slack $MD_{ED} < MD_{EDABS} < MD_{EDREL}$. Similarly, under small slack $MD_{LS} < MD_{LSREL}$, but that under large slack $MD_{LSREL} < MD_{LS}$. Thus the algorithms behave analogously when the slack is reduced or the load is increased.

7.2 Emulating ED

Having studied the effects of the system parameters on the performance of our emulation algorithms, we will now investigate how to tune EDABS and EDREL for a particular system. As a sample system, we chose Posix Unix, whose parameters are listed in Table 6. Our goals for this section are to show how to tune t_s and N_r to maximize performance and to examine how closely EDABS and EDREL can approximate ED in a representative GPOS.

Figure 10 shows the 'error' in MD introduced by emulating ED. By 'error' we mean the difference in measured MD values between an emulation algorithm and the real-time algorithm it is trying to emulate. We will also refer to error as e_{MD}, where $e_{MD} = MD_A - MD_{ED}$ and A is in $\{EDABD, EDREL\}$. Note that although we call this difference an error, if $e_{MD} < 0$ the

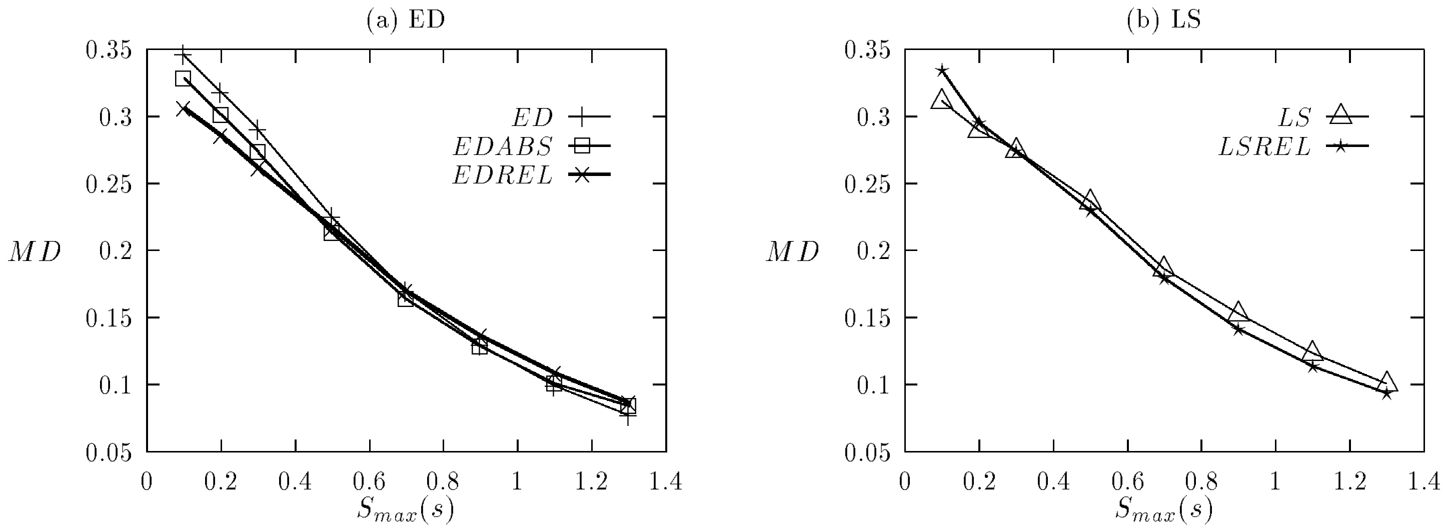

Figure 9: Effects of changing slack on performance.

emulation algorithm is outperforming the real-time algorithm, thus the error is in our favor. Finally, when we refer to algorithms differing by 2%, we mean that $|MD_A - MD_B| = 2\%$.

In Figure 10, $EDABS$ 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. To get good emulation it is necessary to tune the algorithms for the particulars of the system. Below we discuss how to adjust both the reshift count, N_r, and t_s.

7.2.1 Tuning N_r

EDABS uses N_r to determine when to reset t_{pinned}, which we call a reshift. Figure 11 shows the effect of changing N_r under two different system loads. (Note that the scales are different for graphs a and b.) [5] Figure 11a shows that at low loads, using a higher value for N_r lowers MD. This is because the busy periods are typically short for a lightly loaded system: In this case the average number of tasks served in a busy period, $\overline{N_{bp}}$, is 2.2. Short duration busy periods have the following two properties:

[5]Some parameters vary from the baselines setting: $n = 16$,$schslice = 0.3$.

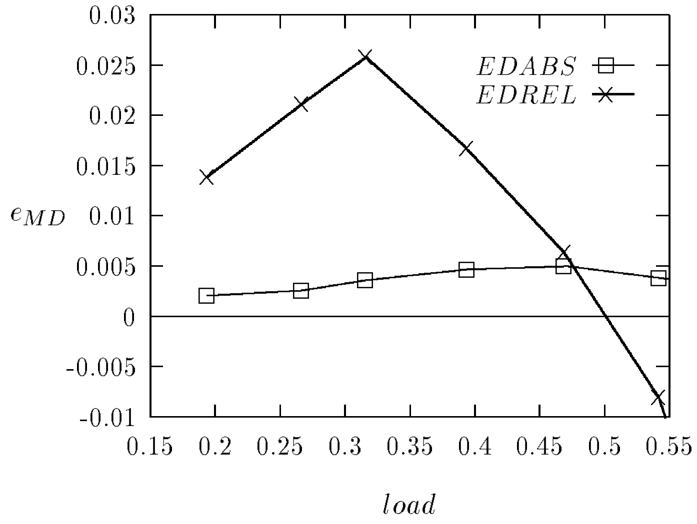

Figure 10: MD Error for Emulated ED

- A job placed in the last level is more likely to result from an unusually distant deadline than from all of the lower levels being full;
- Even if the busy period has been long enough to fill the priority levels, it will probably not last much longer, which will give the system a chance to catch up with the jobs that queued in the last slot. [6]

As Figure 11b shows, however, the situation is reversed under high load. For $\rho = 0.9$, we calculate that $\overline{N_{bp}} \approx 10$ (see Appendix A). In high load cases, it is better to reshift after the first task is assigned to the last level. These two examples show that it is impossible to pick a single value for N_r which will optimize EDABS's performance across all load ranges. 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.

[6] This is due to the memoryless property of the exponential distribution which we have assumed for arrival times.

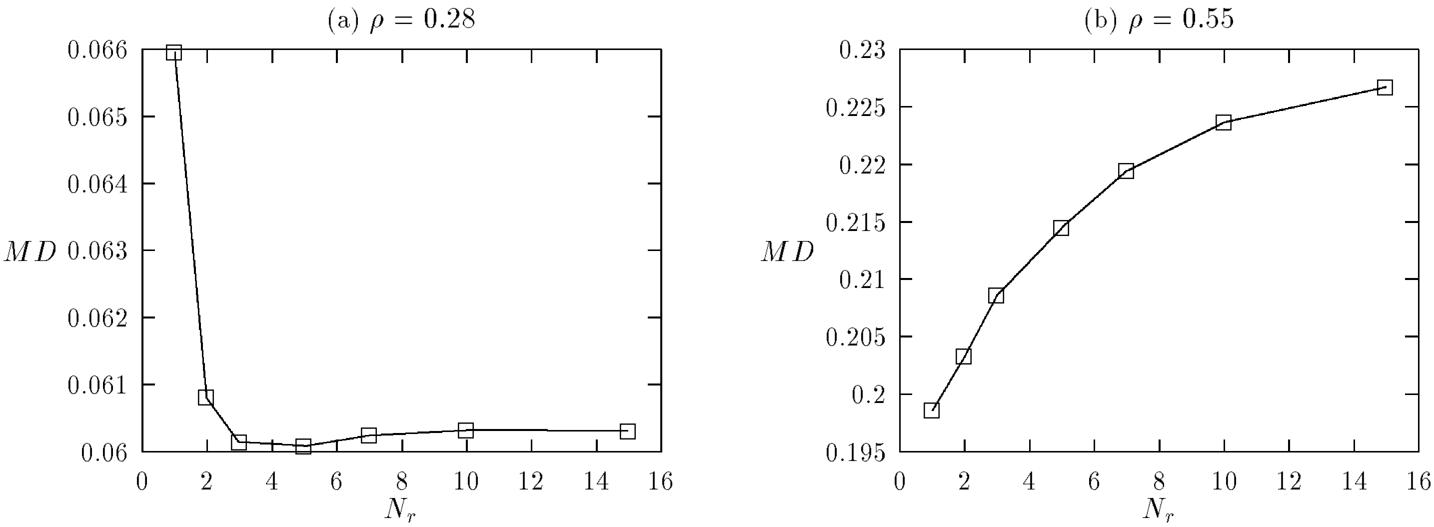

Figure 11: Tuning N_r for EDABS

7.2.2 Tuning t_s

Figure 12 plots MD as t_s is varied for two different load points. [7] As above, there is no single value of t_s which optimizes MD in both instances. 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 $max(d(T) - t)$ evenly into n regions. If we use the approximation

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

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 $max(d(T) - a(T))$. For the parameters used in this simulation run, and $k = 3$, $t_s \approx 2.3$. This shows excellent agreement with the high load graph, but poor agreement with the low load graph. A good compromise is $0.3 \leq t_s \leq 0.4$. For EDABS, the graphs suggest that $t_s = 0.3$ is a good choice, but there's no good intuition. Given that the range of MD under low load is

[7] The jitteriness of Figure 12a, and of some of the graphs that follow, occurs because the vertical scale of the graph is so greatly magnified that it shows fluctuations within the measured confidence intervals.

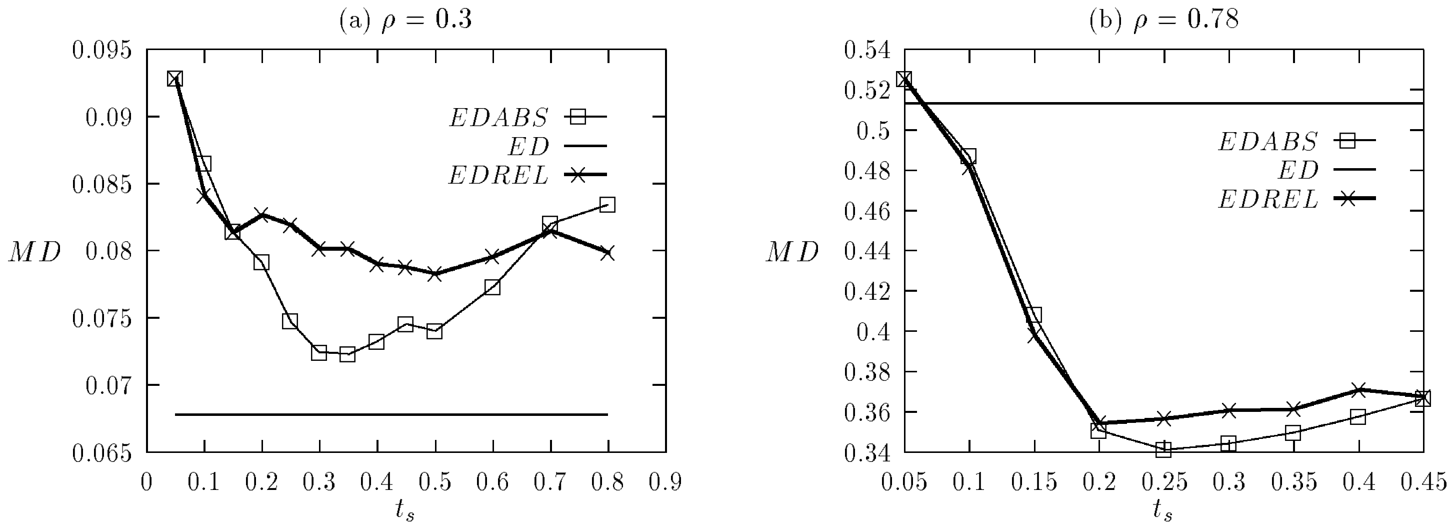

Figure 12: Tuning t_s for EDABS and EDREL

much smaller than the range under high load, if nothing is known about average system load it is safer to pick t_s for the high load case using the rule of thumb.

Clearly, the fact that one has to select a good t_s for all of the priority assignment algorithms is a drawback. If one has no idea what the range of expected slacks of task execution times will be, then it is hard to pick a value for t_s. However, Figure 12 shows that there is a wide range of t_s values that give similar results. Thus, one only needs to have rough estimates of the task characteristics to select a reasonable t_s value. 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.

7.3 Emulating LS

As in the previous section on emulating ED, we now turn our attention to studying LSREL under more representative system parameters and learn how to tune t_s to improve its performance. Again we use the Posix system parameters from Table 6.

Figure 13 shows the error in MD introduced by emulating LS. The quantity graphed is $e_{MD} = MD_{LSREL} - MD_{LS}$. $LSREL$ tracks LS to within the confidence interval of the

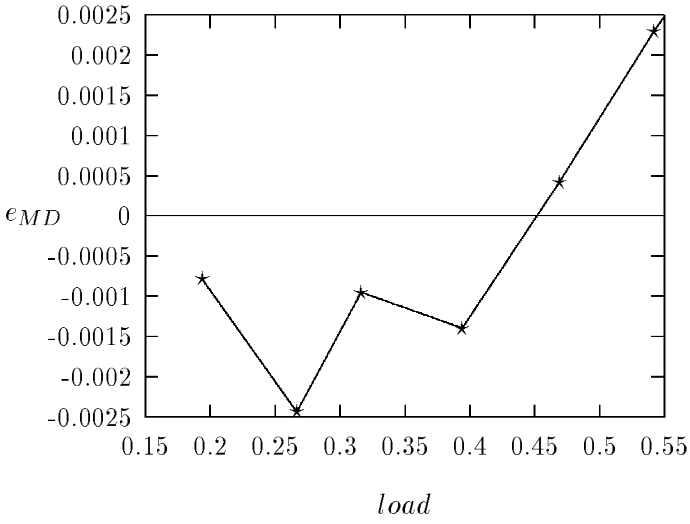

Figure 13: MD Error for Emulated LS

simulations: MD_{LS} and MD_{LSREL} never vary by more than 0.25%. From this we can see that even under a commercial GPOS, LSREL emulates LS very closely.

7.3.1 Tuning t_s

LSREL has only one parameter that can be adjusted, t_s. Figure 14 shows MD as t_s is varied for two different load points. In these two graphs we see problems similar to when we tuned t_s for EDREL. Figure 14b indicates that optimally, $t_s \approx 0.15$. This fits well with our intuition that $nt_s \approx S_{max}$. The low load case, however, is optimized with a much larger t_s. Note, however, that the differences in MD under low load are very small ($< 1\%$). So, without prior knowledge of the system load, it is safer to select t_s for the high load case using $t_s = \frac{S_{max}}{n}$.

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.

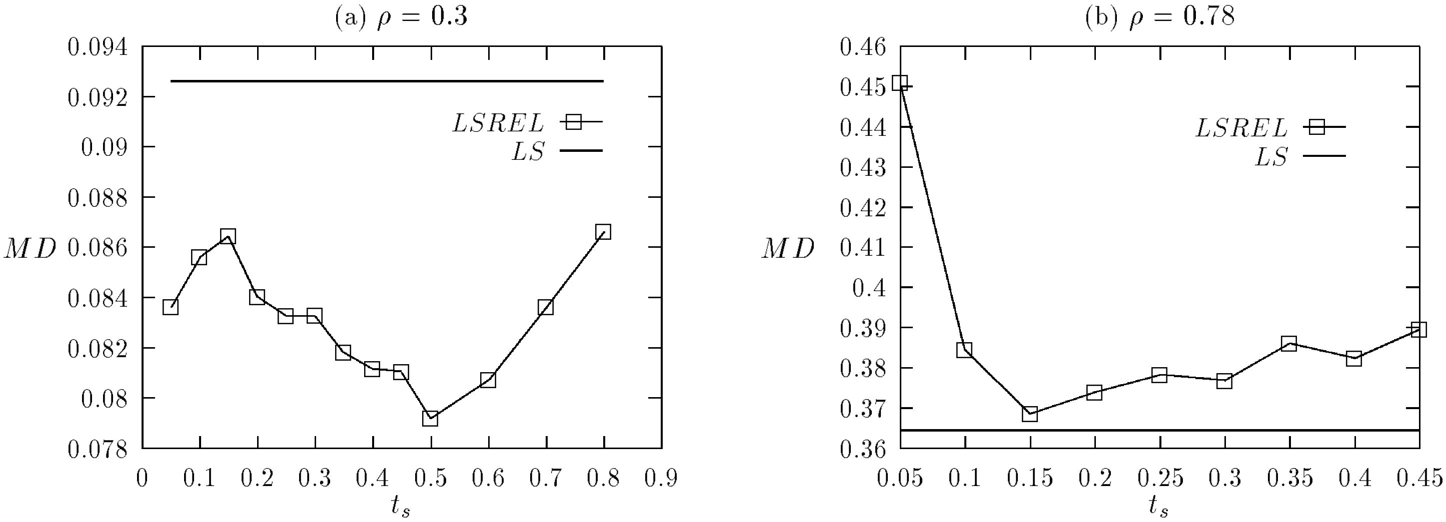

Figure 14: Tuning t_s for LSREL

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.

[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.

[Kle75] L. Kleinrock. *Queuing Systems*, volume 1. Wiley-Interscience, 1975.

[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.

[Voe87] J. Voelcker. How computers helped stampede the stock market. *IEEE Spectrum*, 24(12):30–33, 1987.

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

A Reshift Rates

When we perform a reshift, old tasks will have incorrect priorities relative to the tasks arriving after the reshift. These old tasks are not aborted (they will be executed as the queues empty) but are likely to miss their deadlines. When a reshift occurs, then, EDABS will perform very differently than ED for the remainder of the busy period; Therefore, Reshifts interfere with our goal of emulating ED as closely as possible. It then seems reasonable to ask the following question: Given a set of system parameters, how often will reshifting be necessary?

We define a new metric call reshift rate, $R_{reshift}$, which is the average number of reshifts per job. For any system, $R_{reshift}$ will always be in the range $[0 \ldots 1]$. The reshift rate will clearly be dependent on the average system load, ρ. As ρ is increased, the average length of the busy periods, and the average number of jobs served in a busy period ($\overline{N_{bp}}$), will increase as well. From [Kle75] we have that $\overline{N_{bp}}$ is

$$\overline{N_{bp}} = \frac{1}{1-\rho}.$$

We can use this result to rephrase our original question to be: On average, how many jobs can be serviced in a busy period before a reshift is necessary?

We will call this number $\overline{N_0}$. The variable $\overline{N_0}$ can be decomposed into the sum

$$\overline{N_0} = k + N_r$$

where k is the number of jobs that can arrive before one is assigned to the last priority level and N_r is the reshift count. Since N_r is a parameter to the scheduler, to find $\overline{N_0}$ we need only determine the value of k. With no loss of generality, assume that the first task, T_1, arrives at time $t = 0$. If the average execution time is $\overline{x}$ and the average slack is $\overline{s}$, then if T_1 is an 'average' task

$$d(T_1) = \overline{x} + \overline{s}.$$

Of course, the real deadline of task T_1 can only be described by a probability density function since x and s are random variables, but since we are only looking for rough numbers, we will accept this approximation. In a busy period, the average interarrival period is $\overline{x}$ ([Kle75]). So, again assuming that all tasks are average, we have that

$$d(T_k) = \overline{x} + \overline{s} + (k-1)\overline{x} = k\overline{x} + \overline{s}.$$

Since we want to know how many tasks can arrive and *not* be assigned into the last priority queue, we are interested in the greatest k such that

$$d(T_k) \leq (n-1)t_s.$$

Combining equations we have

$$k \leq \frac{(n-1)t_s - \overline{s}}{\overline{x}}.$$

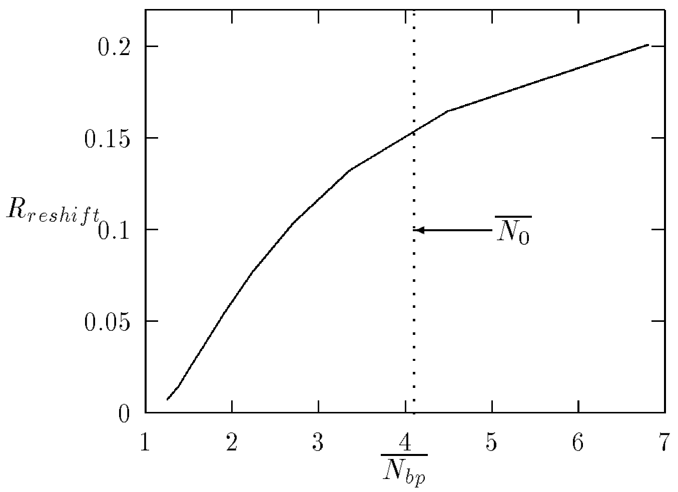

Figure 15: Effect of $\overline{N_{bp}}$ on Shift Rate.

Now we can solve for $\overline{N_0}$,

$$\overline{N_0} = k + N_r$$
$$\overline{N_0} \leq \frac{(n-1)t_s - \overline{s}}{\overline{x}} + N_r.$$

So, when $\overline{N_{bp}} \ll \overline{N_0}$, busy periods will not last long enough to cause reshifts, but when $\overline{N_{bp}}$ approaches $\overline{N_0}$, EDABS will be forced to reshift. Figure 15 shows the effect of $\overline{N_{bp}}$ on the reshift rate in an actual experiment [8]. For $\overline{N_{bp}} \leq 2$, less than 1 in 20 jobs cause a reshift, but as $\overline{N_{bp}}$ approaches $\overline{N_0}$, more than 1 job in 10 produce a reshift. By the time $\overline{N_{bp}} = \overline{N_0}$, more than 1 in 7 jobs causes a reshift, and this ratio only gets worse as $\overline{N_{bp}}$ is increased further. These experimental results show excellent agreement with the approximate guideline we derived above.

[8] This experiment was run with the baseline parameters given in Section 6