Bringing Real-time Scheduling Theory and Practice Closer for Multimedia Computing

R. Gopalakrishnan
gopal@dworkin.wustl.edu

Gurudatta M. Parulkar
guru@flora.wustl.edu

Department of Computer Science
Washington University in St.Louis

Abstract

This paper seeks to bridge the gap between theory and practice of real-time scheduling in the domain of high speed multimedia networking. We show that the strict preemptive nature of real-time scheduling leads to more context switching, and requires system calls for concurrency control. We present our scheduling scheme called rate-monotonic with delayed preemption (RMDP) and show how it reduces both these overheads. We then develop the analytical framework to analyze RMDP and other scheduling schemes that lie in the region between strict (immediate) preemption and no preemption. Our *idealized scheduler simulation* methodology accounts for the blocking introduced by these schemes under the usual assumption that the time for context switching and preemption is zero. We derive simpler schedulability tests for non-preemptive scheduling, and prove a variant of rate-monotonic scheduling that has fewer preemptions. Our measurements on Sparc and Pentium platforms, show that for the workloads we considered, RMDP increases useful utilization by as much as 8%. Thus our scheduling policies have the potential to improve performance over existing methods.

[0]Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMETRICS 96-5/96 Philadelphia, PA, USA

©1996 ACM

1 Introduction

Multimedia applications represent an important class of network applications. These applications are enabled by the availability of high speed networks, and affordable high performance personal computers and workstations. We identify two important requirements of multimedia applications:

- Continuous media communication and processing must occur in real-time. In addition, both the network and the endsystems must provide quality-of-service (QoS) guarantees to shield such applications from unpredictable network and system loads.
- Multimedia applications are compute and data intensive. So they require high performance in terms of both transmission speeds and processing power. Efficient use of network and compute resources is therefore important.

This paper is concerned with operating system (OS) support for multimedia applications. An essential feature required of an OS to support multimedia is real-time scheduling. However, providing guaranteed real-time performance to applications without compromising their efficient operation in a practical implementation is an interesting and elusive problem. This problem we believe points to a gap between theory and practice of real-time scheduling in the following sense–scheduling algorithms that have a good theoretical basis are not the most suitable choice from a practical standpoint, and scheduling schemes that are dictated by practical considerations do not have theoretical results to guarantee their real-time operation.

The purpose of this paper is twofold. We first illustrate the above situation, that is we describe a real-time scheduling scheme that is highly efficient but for which none of the existing theoretical results can be used as they are. Second we present new theoretical results that can be used for our scheduling scheme as well as for a variety of other efficient scheduling algorithms. More specifically, we show that the strict preemptive nature of traditional scheduling algorithms such as rate-monotonic (RM) and earliest-deadline-first (EDF) leads to inefficiencies specially when used to schedule protocol processing at high speeds. Strict preemption leads to excessive involuntary context switching. It also requires the use of expensive real-time concurrency control operations during protocol processing.

We present our modified RM scheme called RMDP (RM with delayed preemption) as an example that solves both the problems mentioned above. The scheme is specially suitable for protocol processing as well as media processing for multimedia applications. However a survey of existing theoretical results show that they either deal with the case when preemption is immediate *i.e.*, the instant a higher priority task arrives (fully preemptive), or when there is no preemption (non-preemptive). The main contribution of this paper is a general technique to analyze real-time scheduling schemes in the region that lies between the fully preemptive and the non-preemptive cases. A new feature of our theoretical framework is the notion of a *preemption policy*. Our proof method, that we call *idealized scheduler simulation*, allows us to evaluate the impact of a given preemption policy on well known scheduling algorithms such as RM and EDF.

Our study of preemption policies have led to other interesting results. One result consists of schedulability tests for non-preemptive scheduling by treating it as a special case of our general preemption policies. Another interesting result is a variant of RM scheduling (that we call *mixed* scheduling) that uses an EDF preemption policy and is able to achieve almost as few preemptions as EDF scheduling. All the results above are derived assuming that context switching costs are zero. Since our main concern is efficiency issues in protocol processing, it is essential to quantify the cost of context switching that arises due to real-time scheduling. We therefore instrumented our implementation to measure and quantify the effect of context switching on performance. Our results show that in most cases, useful CPU utilization can be increased by using our efficient scheduling schemes.

The outline of the paper is as follows. Section 2 describes relevant aspects of our target environment that includes the real-time upcall (RTU) facility, and the RMDP scheduling policy. Section 3 presents existing results and shows why we cannot use them as they are to analyze RMDP and similar schemes. Section 3.1 gives an overview of our experimental results that show that RMDP can increase effective utilization. The different preemption policies and the ISS method is presented in Section 4. The next three sections apply the ISS methodology. Section 5 accounts for the effect of preemption policies on the schedulability tests for RM and EDF. Sections 6 and 7 discuss extensions of these results to non-preemptive and mixed scheduling respectively. Section 8 describes our experiments with the RTU facility on the Pentium and Sparc platforms to quantify context switching costs. We finally present our conclusions and future work.

2 Target Environment

Our work is targeted towards protocol processing support for networked multimedia applications. Media processing and protocol processing for each connection are modeled as periodic activity with certain compute and communication resources allotted each period. In our context, periodic activity in each process is implemented as a real-time upcall (RTU). Each RTU has a period and a computation time. It is also associated with a handler routine in the user program that gets upcalled by the kernel when the RTU is run. The OS performs a schedulability test to determine if all RTUs can complete before their deadlines *i.e*, before their next period. The kernel makes an RTU runnable at the start of its period by placing it in the run queue. From the run queue the RTU scheduler picks an RTU to run based on its priority. The priority is determined by the scheduling policy used. The RTU scheduler uses RM priorities which means that smaller the period of an RTU, higher is its priority.

The key feature of the scheduling scheme is that it takes advantage of the iterative nature of protocol processing to preempt a running RTU only at the end of an iteration. An iteration in this case corresponds to processing one (or more) protocol data units (PDUs). To implement this preemption scheme, each RTU and the kernel communicate via a data structure in shared memory. The kernel sets a flag in this shared structure to inform a running RTU that a higher priority RTU has become ready to run. The handler checks this flag after each iteration and if it is set, it yields the CPU by returning from the handler. If the yielding RTU has not completed its work for the current period, the kernel puts it back in the run queue. The RTU is invoked later when it regains the highest priority status in the run queue. Because a running RTU can delay its preemption by one iteration, we call this scheme RM with delayed preemption (RMDP). The RMDP scheme has two efficiency benefits.

- Since RMDP does not preempt a handler asynchronously, RTU handlers do not need to lock shared data that is accessed only inside an iteration. In a real-time system, lock operations must be mediated by the OS to prevent unbounded priority inversion [13]. Thus each lock operation involves a system call. For our Pentium machine running NetBSD, the system call overhead is about 770 processor cycles (or 280 instructions) [1]. Typically shared state is updated after each PDU, and so locking system calls must be made for each PDU. Since PDU processing itself takes only a few hundred instructions, getting and releasing a lock per PDU increases PDU processing time manyfold. In addition, the number of PDUs processed per unit time increases linearly with transmission rate. This results in a linear increase in the cost of concurrency control.
- Since at least one iteration is completed every time a handler runs, the number of preemptions is reduced. Indeed our experiments show that RMDP reduces the number of involuntary context switches thus increasing useful CPU utilization.

Full details of the RTU implementation including security issues, system call interface, other efficiency benefits, and protocol performance measurements can be found in [2, 4].

Thus RMDP increases efficiency in at least two ways: it eliminates per packet system calls for locking, and reduces involuntary context switching. To provide guarantees we also need to have a schedulability test that takes into account the preemption delay introduced by RMDP. In the next section we survey some of the scheduling algorithms that have been studied in previous work, and show why they cannot be used as they are to analyze RMDP.

	Utilization Bound	
	RM	EDF
Fully Preemptive	$n(2^{1/n}-1)$	1.0
No Preemption	N/A	pseudopolynomial test
With Blocking	$n(2^{1/n}-1)$ $-\max(B_i/T_i)$	$1-\max(B_i/T_i)$

Figure 1: Related Results

3 Related Work and their Limitations

From now on we refer to periodic activity (an RTU or real-time thread) as a *task*. A task has a period T and a computation requirement C. The utilization of a task is C/T, and the total utilization U of a set of tasks is the sum of their individual utilizations. We start with the well known RM and EDF scheduling algorithms introduced in [11]. RM gives higher priorities to tasks with smaller periods, and EDF gives higher priorities to tasks with earlier deadlines. The analysis in [11] assumed a uniprocessor system where tasks are independent, and deadlines are equal to the periods. It also assumes that preemption is immediate *i.e*, a running task is preempted as soon as a higher priority task becomes runnable We refer to such a scheduler as an *idealized scheduler*. We assume that the context switch time is zero.

Table 1 summarizes the relevant schedulability results for RM and EDF scheduling. The first row corresponds to the fully preemptive case and is the most widely used. Each table entry gives the maximum task utilization that can be supported by the scheduling scheme to guarantee schedulability (*i.e*, all tasks meet their deadlines). As can be seen, EDF can support 100% CPU utilization. RM on the other hand limits utilization to $n(2^{1/n}-1)$, which is about 69% for reasonable values of n (the number of tasks). However RM is widely used because it is simple and is easy to implement. The second row corresponds to the case when there is no preemption. There are no simple schedulability tests for this case as there are in the fully preemptive case. We discuss non-preemptive EDF scheduling in Section 6. The third row presents schedulability tests for RM and EDF that take into account the effect of *blocking*. Blocking occurs when a task is prevented from running by a lower priority task. Blocking that occurs when a lower priority task is in a critical section of code is considered in [13]. It can be seen that blocking reduces the utilization bound compared to the idealized case. The reduction is equal to the maximum value of B_i/T_i over all tasks, where B_i is the time for which task i can be blocked, and T_i is its period. Blocking due to other system events such as periodic timer interrupts is considered in [7].

None of the results in Table 1 can be directly used to model delayed preemption. The fully preemptive test does not hold because it assumes tasks cannot be blocked. Using the non-preemptive test is a conservative approach because it may reject task sets that are schedulable with delayed preemption. The blocking case at first glance seems to capture the effect of delayed preemption. However its derivation assumes that during the duration that a task is blocked, no task makes any progress. In other words, the blocking time is regarded as a system overhead (somewhat like a high priority system activity). Thus it underestimates the utilization that can actually be supported. Theoretically the amount by which it can underestimate the utilization can approach 50%. This shows that it is important to distinguish between blocking caused due to overhead and blocking due to other tasks. Borrowing terminology in [8] we characterize existing results as analyses of *mechanism blocking i.e.*, concerned with overheads caused due to system activities. Our use of preemption policies on the other hand is an example of *policy blocking*, and this is an important distinction between our results and previous work.

3.1 Experimental Results for RMDP

In Section 4.1 we show analytically that policy blocking caused by our preemption policies reduces the utilization bound. These results are derived assuming that there is no mechanism blocking. However in a real system, the savings in context switching due to the use of our preemption policies *increases* the utilization that can be supported compared to the case when immediate preemption is used. The net effect is that there can often be an increase in the useful utilization. We demonstrate this with an experiment in which we created 16 processes, each with 6 RTUs with periods ranging from 40 milliseconds (msec) to 90 msec in increments of 10 msec. We compared two cases, one in which preemption was immediate, and the other in which preemption is delayed until the running handler completes. The running time of each handler was 0.54 msec giving a total utilization of 0.86. The experiment was run long enough (least common multiple of RTU periods) on a 100 Mhz Pentium running the NetBSD OS with a software clock interrupt frequency (`hz`) of 1000. Figure 2 plots the smallest difference measured between the deadline and completion time for each RTU. It can be seen that for RM with immediate preemption, all the 90 msec have negative ordinates, which means that each of them finished after their deadline in at least one invocation. In the case of RMDP, no RTU was found to miss its deadline. This result can be interpreted as saying that the increase in utilization due to RMDP is equal to the utilization of the 90 msec RTUs. This gain comes to around 9%. The rest of the paper essentially explains this result by analysis and measurement. Section 4.1 develops the tools to quantify the effect of policy blocking. Section 8 quantifies the effect of reducing the number of involuntary context switches by measurements.

4 Preemption Policies

Before we present preemption policies, we introduce some notation and describe how the scheduler operates when a particular preemption policy is in effect. Tasks are represented as $J_1, \ldots, J_n$. A task J_i has a period T_i and a CPU requirement (or usage requirement) of C_i. We assume that tasks with smaller subscripts have smaller periods. The utilization u_i of task J_i is given by C_i/T_i, and the total utilization U is the sum of the utilization for all tasks. The scheduler operates as follows. When a task arrives, it is assigned a priority according to the particular scheduling policy (RM or EDF) used. The task is then placed in the run queue which we assume is ordered on the basis of priority. If the new task lands at the head of the queue (i.e it has the highest priority among all runnable tasks in the system), the

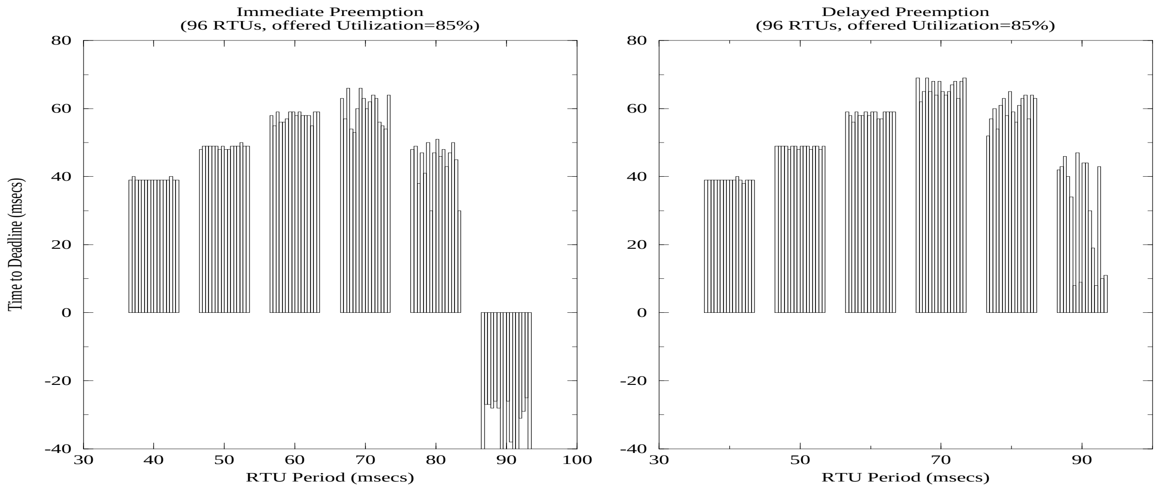

Figure 2: Increase in utilization by reducing context switching

preemption policy is used to decide whether the running task should be preempted or not. If a preemption decision is made the appropriate action is taken to move the running task back to the run queue. In the case of RTUs, this would involve setting a flag in the shared data structure. The CPU becomes available for the next eligible task some time after the preemption action was initiated. This time delay is called the *preemption interval.* We assume that the preemption mechanism itself does not introduce any delays and only the preemption policy can contribute to the preemption interval. Once the running task yields the processor, the highest priority task in the run queue is assigned the processor. It must be noted that several tasks can arrive during the preemption interval. These tasks are also placed in the run queue and are considered when the choice for the next task to run is made.

We denote the running task by J_r and an arriving task by J_a. The threshold and delayed preemption policies are formalized as follows.

- **Delayed Preemption:** In this scheme, the computation time C_i for a task is divided into equal sized quanta c_i. We allow J_r to be preempted only at the end of a quantum. If the priority of J_a is higher than that of J_r, action is initiated to preempt J_r at the end of its current quantum. The fully preemptive case occurs when the quantum size for all tasks is set to 0. The non preemptive case occurs when c_i is set to C_i for all tasks.
- **Threshold Preemption:** In this scheme, J_a preempts J_r if it has the highest priority among the runnable tasks (including J_r), and the periods T_a and T_r are related by the inequality $T_a/T_r < K_r$ where the threshold K_r is given by $0 \leq K_r \leq 1$. It is easy to see that the fully preemptive case occurs when the value of the threshold K_r is set to 1. Similarly the non preemptive case occurs when K_r is set to T_1/T_r. The idea behind threshold preemption is that tasks that have almost equal periods are treated as though they have the same priority.

4.1 Idealized Scheduler Simulation

We have developed idealized scheduler simulation (ISS) as a general method to analyze the effect of policy blocking on the schedulability test of a scheduling algorithm. The basic idea behind ISS is to simulate the essential operation of a blocking scheduler (the actual case) with an ideal scheduler (idealized case) that never blocks. By essential operation, we mean that task invocations in the idealized case complete at the same time as in the actual case. Thus if the tasks in the idealized case meet their deadlines, then they do so in the actual case as well. The schedulability test for the idealized case can therefore be adopted for the actual case. To go from the actual case to the idealized case in ISS, the computation requirements of the tasks are modified while keeping their periods the same. For each blocking event that occurs when a task delays its preemption (the blocking task), the ISS method does the following steps.

1. It determines which tasks need to be modified in the idealized case so as to simulate the essential operation of the actual scheduler. Lemma 1 shows that we only need to modify tasks that have priority higher than that of the blocking task.

2. For the higher priority tasks it determines by how much the original computation requirements must be changed and this is shown in Lemma 2.

3. Due to these changes, the requested utilization of the modified task set becomes artificially higher than that of the original task set. Since the upper bound on utilization is fixed by the scheduling policy (RM or EDF), some utilization becomes unavailable. Lemma 3 shows how to calculate this upper bound on the unavailable utilization.

4. Finally the schedulability test for the blocking case is obtained by taking the maximum of the unavailable utilization due to all blocking tasks, and subtracting this from the utilization bound of the idealized case.

In this section we state Lemmas 1, 2, 3, and discuss them informally. The proofs are only of technical importance and are not provided in this paper. They may be found in [3].

We begin with some notation. A task J_i in the actual case has a corresponding task J_i' in the idealized case. The tasks J_i' have modified computation requirements as per the ISS method. We can imagine two schedulers, one that runs the tasks J_i, and the other that runs the tasks J_i'. We parameterize the time duration for which a running task J_r can block other tasks (the preemption interval) as Δ_r. For example, in the case of threshold preemption we have $0 \le \Delta_r \le C_r$, and in the case of delayed preemption we have $0 \le \Delta_r \le c_r$. We now compare a running task J_r, and its counterpart J_r' in the following lemma.

Lemma 1 *Suppose that corresponding invocations of J_r and J_r' are running (in their respective schedulers) at a given time, and a task with higher priority arrives. Suppose J_r continues to run for a duration Δ_r thereby gaining a usage of Δ_r compared to J_r'. Then at the instant that J_r resumes execution, J_r' would have resumed and regained exactly the amount Δ_r in usage.*

An important consequence of this lemma is that in both the actual and the idealized cases, the time at which the CPU becomes available to tasks with priority lower than J_r is the same. Hence only tasks with priority higher than J_r are affected by delayed preemption.

We next identify the set of higher priority tasks that actually get affected by the blocking due to J_r. Identifying this set is necessary to isolate and study the effect of the blocking caused by J_r. This set W_r, consists of tasks that arrive during the preemption interval of J_r and are blocked by J_r, but which run in the idealized case that is simulating the essential behavior of the actual case. While analyzing a particular preemption policy we will have to determine the set W_r. Examples are shown in Sections 5.2 and 5.1 where we apply ISS to analyze RM and EDF scheduling.

We define the set $W_r = \{J_{r_1}, \ldots J_{r_i}, \ldots J_{r_m}\}$, and the set W_r' as the set that contains for each invocation J_{r_i} in W_r, its counterpart J_{r_i}'. The tasks in W_r are listed in order of decreasing priority. For each invocation J_{r_i}', let the amount of usage that it obtains within the delay interval be d_{r_i}. It can be seen that the sum of these usages $\sum d_{r_i} = \Delta_r$.

Lemma 2 *Consider corresponding invocations in W_r and W_r'. At the end of the preemption interval, each invocation J_{r_i}' has got an extra d_{r_i} amount of usage compared to J_{r_i}. If the usage requirement of each J_{r_i}' is increased by an amount d_{r_i}, then both J_{r_i}' and J_{r_i} complete execution at the same time. In addition, if a task J_{r_i}' meets its deadline with this increased requirement, then so does the task J_{r_i}.*

The above lemma determines the amount by which the usage requirements of tasks should be increased so that they complete (in the idealized case) at the same time as the corresponding tasks blocked by J_r in the actual case. We also note that if the usage requirement of the task J_r' is reduced by the amount Δ_r then it will complete at the same time as J_r. With these changes, the essential operation of the simulated and actual cases would be identical. The unavailable utilization due to these modifications is given next.

Lemma 3 *The amount of unavailable utilization N_r due to blocking by J_r is bounded by $\Delta_r(1/T_{r_1} - 1/T_r)$.*

The above result can be explained simply as follows. The ISS method increases the computation requirement of the highest priority task that can be blocked by J_r by an amount Δ_r and reduces the computation requirement of J_r by Δ_r. The unavailable utilization is the (artificial) increase in utilization of the blocked task minus the (artificial) decrease in utilization of J_r. Knowing the minimum period T_{r_1} (from W_r), and the maximum value of Δ_r for the task J_r, the value of N_r can be computed using the above formula. To determine the sufficient condition for the task set to be schedulable, the worst case value of N_r must be determined for all tasks in the task set and subtracted from the utilization bound of the scheduling algorithm used.

5 Analysis of Preemption Schemes

We now apply the ISS method to analyze various preemption policies introduced earlier. The general technique is to determine the set W_r for the particular scheduling policy (RM or EDF) and the preemption policy being used. These can then be used to compute N_r in Lemma 3.

5.1 Delayed Preemption

In the case of delayed preemption, a task J_r can block any higher priority task. Thus W_r can potentially include all tasks with priority higher than J_r. In the rate-monotonic case this implies all tasks with periods smaller than that of J_r are part of W_r. The situation is the same in the earliest-deadline-first case as well. This is because tasks with periods bigger than that of J_r will have deadlines later than that of J_r. Thus the value of the smallest period in W_r is T_1 in both the cases. Since the maximum time for which J_r can block others is c_r, N_r is bounded by

$$N_r \leq c_r(1/T_1 - 1/T_r)$$

The schedulability test for RM is given by 1 and the test for EDF is given by 2.

$$U \leq n(2^{1/n} - 1) - \max_{1 \leq r \leq n} c_r(1/T_1 - 1/T_r) \tag{1}$$

$$U \leq 1 - \max_{1 \leq r \leq n} c_r(1/T_1 - 1/T_r) \tag{2}$$

The above results give an increased utilization bound compared to [13] where the reduction in utilization due to blocking would be c_{max}/T_1.

5.2 Threshold Preemption

In this scheme a task J_r can block another task for a maximum time C_r. Thus $\Delta_r \leq C_r$. To determine W_r we have to consider the RM and EDF policies as shown next.

Rate Monotonic Policy In RM an arriving task J_a has higher priority if $T_a < T_r$. However J_a does not preempt J_r if $T_a \geq K_r T_r$. Thus in this case the set W_r includes all tasks T_a with periods less than T_r such that $T_a \geq K_r T_r$. The period of the task with the smallest period in W_r is given by $T_{r_1} \geq K_r T_r$. Thus N_r is bounded by

$$N_r \leq C_r(1/K_r T_r - 1/T_r) = u_r(1/K_r - 1)$$

It must be noted here that if W_r is empty (as it is when $r = 1$), then $N_r = 0$ regardless of the value of U_r. The schedulability test can therefore be written as

$$U = \sum_{i=1}^{n} u_i \leq n(2^{1/n} - 1) - \max_{1 \leq r \leq n} u_r(1/K_r - 1) \tag{3}$$

It can be easily seen that for $K_r = 1$ the test reduces to the standard result in [11]. It must be noted that the above result holds for any combination of task periods. For a particular task set, the value of T_{r_1} may be larger than $K_r T_r$ and so we may get a smaller value of N_r, and consequently a better utilization bound.

Earliest Deadline First Policy We determine the set W_r as follows. To decide if a task J_a that arrives during $J_r' s$ preemption interval should be included in W_r or not, we consider two cases—

$T_a \geq K_r T_r$: We first consider the case when J_a has the highest priority among all runnable tasks (including J_r). When it arrives, J_a is blocked by J_r because of the threshold condition. However in the idealized case, J_a preempts J_r and therefore it is included in W_r. On the other hand if J_a at the time of its arrival is not the highest priority task among the runnable ones, then it does not cause a preemption in the actual case. In the idealized case, the counterpart of the highest priority task in W_r should be running when J_a arrives. This is because the ISS method would have increased its usage requirement in such a way that it can complete only after the delay interval. Thus there is no preemption in the idealized case either, and so J_a should not be a member of W_r.

$T_a < K_r T_r$: J_a cannot belong to W_r in this case because of the following reasons. If J_a, when it arrives, has the highest priority among all runnable tasks (including J_r) then it preempts the running task both in the idealized, and in the actual case. In fact the delay interval Δ_r ends at the instant this preemption occurs. So J_a does not belong to W_r in this case. On the other hand if J_a at the time of its arrival is not the highest priority task among the runnable ones, then it does not preempt J_r in the actual case by definition of the preemption policy. As in the previous case the counterpart of the highest priority task in W_r should be running in the idealized case and so J_a does not cause a preemption in the idealized case either. Thus J_a cannot be a member of W_r.

Hence the value of the smallest period in W_r is such that $T_{r_1} \geq K_r T_r$. Thus N_r, is bounded by

$$N_r \leq C_r(1/K_r T_r - 1/T_r) = u_r(1/K_r - 1)$$

As before if W_r is empty (as it is for $r = 1$), then $N_r = 0$ regardless of the value of U_r. The schedulability test can therefore be written as

$$U = \sum_{i=1}^{n} u_i \leq 1 - \max_{1 \leq r \leq n} u_r(1/K_r - 1) \qquad (4)$$

Improvement over Existing Results. For both RM and EDF scheduling, the improvement in utilization compared to the result given in [13] is equal to u_r. As K_r approaches 1.0, the unavailable utilization given by [13] approaches u_r whereas in our case it approaches 0. Intuitively our result shows that in the limit when K_r approaches 1.0, there is no blocking and consequently there should be no reduction in utilization.

6 Analysis of Non-preemptive EDF Scheduling

In this section we show that non-preemptive scheduling can be treated as a special case of both the threshold preemption and delayed preemption schemes. Non-preemptive scheduling schemes have many practical advantages from the implementation standpoint. As stated in [5], they are easier to implement, and their overheads are easier to characterize. A necessary and sufficient condition is also derived in [5] to determine if a task set is schedulable under a non-preemptive EDF scheduling scheme. However the time complexity of evaluating this condition is polynomial in the maximum task period [5]. This is because the test has to consider all task arrivals for a duration equal to the maximum task period. We have obtained a much simpler test for non-preemptive EDF by considering it as a special case of the threshold and delayed preemption schemes. We illustrate this below:

- *Threshold preemption:* For each task J_i, we set its threshold K_i such that it cannot be preempted even by the task with the smallest period. Since T_1 is the smallest period, if we set K_i to the value T_1/T_i, then no task can preempt J_i.

- *Delayed preemption:* For each task J_i, we set the quantum size c_i equal to C_i. Therefore J_i can never be preempted by any higher priority task.

Substituting these values into Equations 4, and 2 we obtain a simple sufficiency condition for non-preemptive EDF scheduling given by Equation 5 below.

$$U \leq 1 - \max_{1 \leq r \leq n} C_r(1/T_1 - 1/T_r) \qquad (5)$$

Equation 5 has a time complexity of $O(n)$ which is a vast improvement over the test in [5]. However our test only gives a sufficient condition for schedulability whereas the test in [5] provides a necessary and sufficient condition. In the next section we simulate task sets of different characteristics to see how close our sufficient condition is to the optimal condition given in [5].

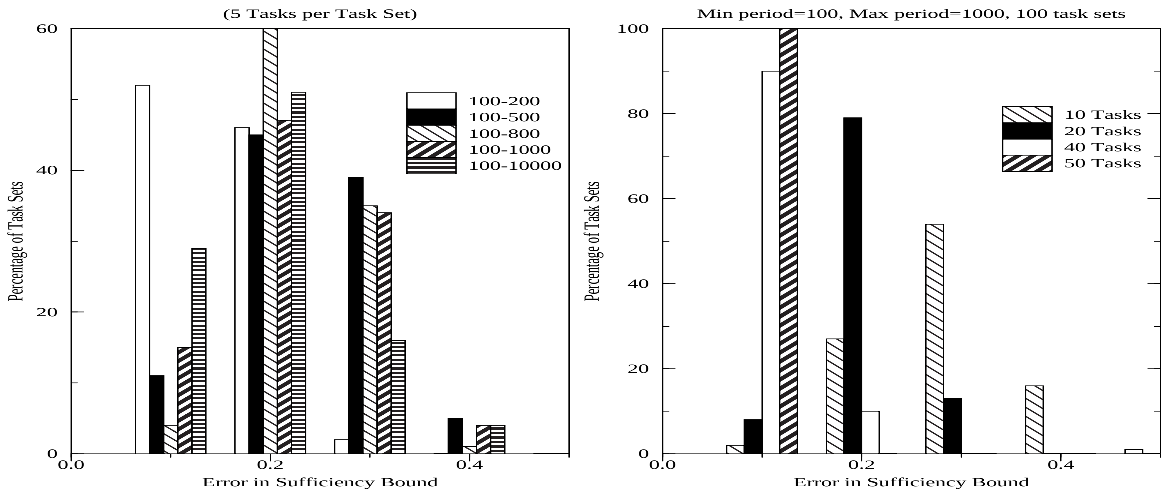

Figure 3: Accuracy of Sufficiency Condition

6.1 Comparing the Sufficient and Optimal Conditions

We compare the simple utilization based sufficiency condition (Equation 5) to the optimal, but more complex necessary and sufficient conditions derived in [5]. We make the comparison by generating task sets based on some criteria, and determining the breakdown utilization [7] predicted by the two tests. Breakdown utilization is defined as follows. The execution time for each task is multiplied by a constant scaling factor α while the periods remain fixed. The utilization point at which any further increase in α causes the schedulability test to be violated is the breakdown utilization. The difference between the breakdown utilizations for the two tests is a measure of the amount by which the simple sufficiency condition underestimates the scheduling capacity. For evaluating the conditions in [5] we used a simplified method suggested in [6].

We vary two parameters while generating a task set– the number of tasks per task set, and the range of periods of tasks in a given task set. In the first simulation we fix the number of tasks per task set to be 5, and try out different ranges for task periods in each set. In the second simulation, the task periods range from 100 to 1000, but the number of tasks per set is varied.

- *Simulation 1:* The ratio of the maximum task period to the minimum task period was varied over different runs in the range 2 to 10. For each run, 100 task sets were generated with task periods chosen according to a uniform distribution between the minimum and maximum periods. The computation requirement for each task was uniformly distributed between 1 and the minimum period. The breakdown utilization for the two schedulability tests was computed, and the difference between the two was obtained. This is shown in the first part of Figure 3. The x-axis has different values of this difference (or error). For a given error ϵ, the y-axis shows for what percentage of task sets belonging to a particular period range, the error in our simple test was less than ϵ. As can be seen, when the range of periods in the task set is small, the majority of the task sets have errors in the lower ranges. As the range of periods increase, higher error ranges are more prevalent. The absolute error however can be as high as 45%, which indicates that the sufficiency bound alone cannot give good utilization in all cases. A reasonable strategy would be to apply the simple test first while admitting a new task. This is a constant time operation. If the sufficiency condition fails, the more sophisticated optimal condition can be checked.
- *Simulation 2:* Here we investigate the effect of increasing the number of tasks in a task set on the accuracy of the sufficiency bound. The tasks were generated as before and the difference of breakdown utilizations was obtained. It can be seen (second part of Figure 3) that as the number of tasks are increased, the accuracy of the test increases steadily. When the number of tasks per set is 50 and above,

the error is within 10% for all task sets. We therefore conclude that the sufficiency bound is very useful for larger task sets. It must be noted that the complexity of evaluating the necessary condition in [5] increases with the number of tasks, which makes our simpler test even more preferable.

7 Mixed Scheduling

In this section we use our earlier results to analyze a simple variation of the rate monotonic scheme. In this variation, tasks are assigned priorities according to the RM scheme. However, a task arrival that has higher priority than the running task preempts it only if its deadline is earlier than that of the running task. In other words, tasks are chosen to run according to the rate monotonic policy, but the preemption decision is based on an earliest deadline first policy. We refer to this as the *mixed* policy. We show that any set of tasks that are schedulable by the RM policy, are schedulable by the mixed policy.

7.1 Analysis of mixed scheduling

We show that any task set that is schedulable by RM is also schedulable by mixed scheduling. We use Lemma 1 which proves that if a running task J_r delays yielding the processor, then its usage at the time instant it resumes[1], is the same as that of its counterpart in the idealized simulation. We consider two cases–

1. *All higher priority tasks that arrive have deadlines later than that of J_r:* In this case J_r runs to completion before yielding the processor. When J_r resumes by definition all the blocked tasks should have completed. By Lemma 1 J_r' also completes at this time. Since J_r' meets its deadline, so do the blocked tasks since they have deadlines later than J_r'.

2. *Some tasks have deadlines earlier than that of J_r:* An arrival that has an earlier deadline would also have a smaller period than that of J_r (since deadlines are equal to periods for periodic tasks). When J_r resumes, by definition all higher priority blocked tasks would have completed, and by Lemma 1 J_r

[1] If it completes execution when it yields the processor, we assume that it resumes but runs for zero time

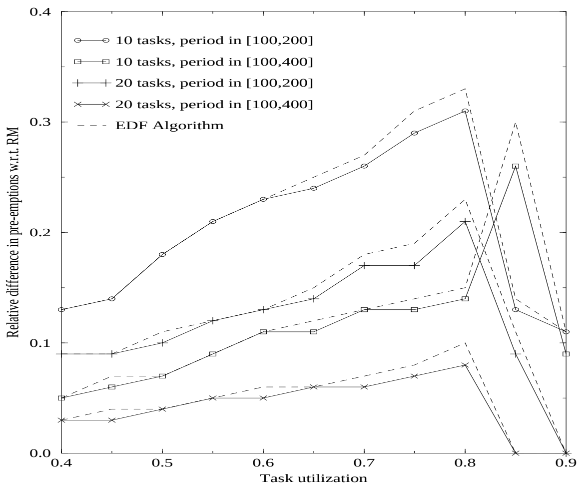

Figure 4: Reduction in Context Switching

will have the gotten the same usage as J_r'. After this instant either both J_r and J_r' complete execution without further higher priority task arrivals, or there is a higher priority task arrival and the above two cases repeat again.

Performance of Mixed Scheduling. We simulated randomly generated task sets to compare the number of context switches in the RM, EDF, and mixed scheduling policies. The task periods and computation times were uniformly distributed as before. The utilization of the task sets was increased in each run until the breakdown utilization was reached. A count of the number of context switches for each policy was recorded in each run. The count for each utilization point was averaged over several task sets. The averaged number of context switches saved in the EDF and mixed policies compared to RM, normalized with respect to the number of context switches in the RM policy is plotted at different utilization points in Figure 4. Each task set was simulated long enough so that these values stabilized.

It can be seen that the number of context switches saved in EDF is only slightly more than in the mixed policy. These two curves track each other closely in the utilization range shown. We also see that as the number of tasks in the task set increases, the relative savings in context switching diminish for both the EDF and mixed policies. As expected the savings in context switches increase at higher utilizations. The difference between the EDF and the mixed policies is more at higher utilizations but maximum difference did not exceed 2%.

From this we conclude that the mixed policy is almost always preferable to the RM policy if reducing context switching is of concern. The only extra operation in the mixed scheme compared to RM is the deadline comparison which is a constant time operation. In EDF on the other hand the arriving task must be inserted in a queue sorted by deadline and this takes $O(\log n)$ time. The mixed policy therefore combines the simplicity of RM with the context switching performance of EDF.

8 Accounting for Context Switching Costs

Our analysis of preemption policies shows that the blocking caused by them reduces utilization. However our experimental results with RMDP indicate that we can support task sets with higher utilization than the case when we use immediate preemption. It follows that the reduction in the number of involuntary context switches offsets the loss of utilization due to blocking. We have verified this fact experimentally on the Sparc and Pentium platforms. We first describe our experiment to measure the context switch time using the RTU mechanism. We then measure the rate at which involuntary context switches occur when the task set is run. This rate depends on the clock interrupt frequency. Finally we calculate the loss of utilization due to context switching and how much we gain by using RMDP.

Measuring Context Switch Time The RTU facility helps us measure context switch time using user level programs. We ran two experiments, one to measure context switch time between RTUs in two different processes (inter-process), and the other to measure the time to switch between RTUs in the same process (intra-process). The context switch times as measured on a 100 Mhz Pentium and on a 25 Mhz Sparc-1 are shown in Table 5. We can see that with a 1 msec clock rate, preemptions on the Sparc-1 will impose too much overhead. We expect that newer Sparc machines will not have this problem. For now, all experiments are reported for the Pentium machine.

Measuring Number of Context Switches. We now go back to our experiment in Figure 2 to explain why the 90 msec RTUs were missing their deadlines when we used immediate preemption (IP) but were able to meet their deadlines when we used non-preemptive (NP)

	Pentium	Sparc-1
Inter-process	35	373
Intra-process	14	114

Figure 5: Measured Context Switch Time (microseconds)

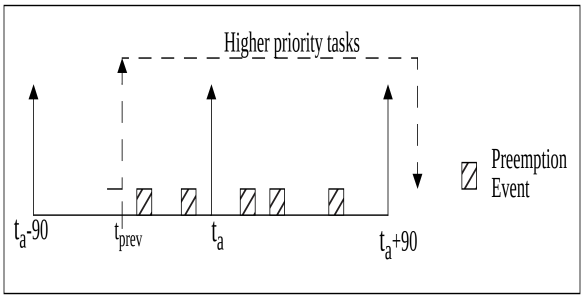

Figure 6: Delays due to Context Switching

scheduling. Consider the IP case shown in Figure 6. A 90 msec RTU J_a arrives at t_a but is unable to complete before $t_a + 90$. Let t_{prev} be the time at which the previous invocation of J_a completes. J_a can be affected due to preemption in the following two ways:

- If P_1 preemptions occur between t_{prev} and t_a, the higher priority RTUs (dashed lines) that are in the run queue at t_a get delayed by the time for P_1 preemptions. Since J_a can run only after the higher priority tasks complete, J_a also gets delayed by P_1 preemptions.
- If P_2 preemptions occur between t_a and t_a+90 then J_a gets further delayed by time for P_2 preemptions.

We need to measure P_1 and P_2 in the worst case. We measured t_{prev} to be $t_a - 70$ msec. Therefore we need to measure the number of preemptions that occur in the worst case in an interval of length $70 + 90 = 160$ msec. We measured this number to be 110. Using the values shown in Table 5 for Pentium, each preemption costs 0.07 msec which results in a delay of $110 * .07 = 7.7$ msec.

Interpreting Experimental Results. The above measurements show that preemptions can cause delays which could lead to missed deadlines. By reducing the number of preemptions using a scheduling scheme such as RMDP, we can offset the reduction in utilization that is predicted by our analysis. In the experiment above, the reduction due to blocking given by Equation 1 was $0.54(1/40-1/90) = 0.75\%$. On the other hand by avoiding the overhead due to preemption we can increase available utilization by up to $7.7/90 = 8.5\%$. Our goal is to be able to predict the gain due to reduction in preemptions not only for the non-preemptive case shown above, but also for other cases. We are currently looking into ways to do this.

9 Conclusions

With the rapid deployment of multimedia applications, the need for real-time scheduling in general purpose computers continues to increase. We have implemented the RMDP scheduling mechanism that lowers the costs of context switching, and eliminates the need for locking operations in protocol processing. To be able to use RMDP and other practical scheduling schemes that operate in the region between strict preemption and non-preemption, we have developed analytical tools based on the idealized scheduler simulation methodology. Using ISS we derive simple schedulability tests for RM and EDF scheduling that apply in cases when preemption is not immediate. The use of preemption policies such as delayed preemption has the potential to reduce context switching costs. To quantify these reductions, we measured context switching costs on the Sparc-1 and Pentium platforms. From the measured values we conclude that the Sparc-1 platform will perform poorly specially when clock interrupts occur every 1 msec. Our results show that schemes such as RMDP that eliminate per packet system calls, and reduce context switches will scale well to high speed networks.

References

[1] Chen, J.B., et. al., "The Measured Performance of Personal Computer Operating Systems," 15^{th} ACM SOSP, Dec. 1995.

[2] Gopalakrishnan R., Parulkar G.M., "Real-time Upcalls: A Mechanism to Provide Real-time Processing guarantees," Tech. Rep. WUCS-95-06, Washington University, St.Louis, 1995.

[3] Gopalakrishnan R., Parulkar G.M., "A Generalized Preemption Model for Real-time Scheduling," Tech. Rep. WUCS-96-04, Washington University, St.Louis, 1996.

[4] Gopalakrishnan R., Parulkar G.M., "A Real-time Upcall Facility for Protocol Processing with QoS Guarantees," (Poster) 15^{th} ACM SOSP, Dec. 1995.

[5] Jeffay, K., Stanat, D.F., Martel, C.U., "On Non-Preemptive Scheduling of Periodic and Sporadic Tasks," 12^{th} *IEEE Real-Time Systems Symposium*, Dec 1991.

[6] Jeffay, K., Stone, D.L., "Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems," 14^{th} *IEEE Real-Time Systems Symposium*, Dec 1993.

[7] Katcher et. al., "Engineering and Analysis of Fixed Priority Schedulers," *IEEE Transactions on Software Engineering*, Sep 1993.

[8] Katcher, D.I., "Engineering and Analysis of Real-time Operating Systems," PhD Thesis, Carnegie Mellon University, 1994.

[9] Khanna, S., et. al., "Realtime Scheduling in SunOS5.0," *USENIX*, Winter 1992, pp.375-390.

[10] Lehoczky, J.P., Sha, L., "Performance of Real-Time Bus Scheduling Algorithms," *ACM Performance Evaluation Review*, Vol.14, No.1, May 1986.

[11] Liu, C.L. and Layland, J.W., "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *JACM*, Vol. 20,No. 1, January 1973.

[12] Pingali, S., "Protocol and Real Time Scheduling Issues for Multimedia Applications," PhD Thesis, University of Massachusetts, Amherst, Sep 1994.

[13] Sha, L. et. al., "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, Vol.39, No.9, Sep 1990.

[14] Sha, L., Lehoczky, J.P., Rajkumar, R., "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling," *IEEE Real-Time Systems Symposium*, Dec 1986.

[15] Tokuda, H., Nakajima, T., Rao, P., "Real-Time Mach: Towards Predictable Real-time Systems," *USENIX Mach Workshop*, Oct 1990.