Processor Scheduling Algorithms for Minimizing Buffer Requirements in Multimedia Applications

Center for Communications and Signal Processing
North Carolina State University
Technical Report TR 94/16

Sanjeev Rampal, Dharma P. Agrawal, and Douglas Reeves,
Dept. of Electrical & Computer Engineering,
North Carolina State University, Box 7911,
Raleigh, NC 27695-7911
Tel: 919-515-3984
Fax: 919-515-5523
E-mail: (sdrampal,dpa,reeves)@eos.ncsu.edu

Processor Scheduling Algorithms for Minimizing Buffer Requirements in Multimedia Applications.

Sanjeev Rampal, Dharma P. Agrawal, and Douglas S. Reeves

Abstract

The increasing use of audio, video and other multimedia applications on workstations, mandates the use of real-time processor scheduling techniques. Real-time scheduling algorithms have mostly concentrated on seeking to meet application *deadlines* (such as those required for video playback for example). Another important requirement of such applications is large buffer memory due to the high data rates involved. We investigate priority allocation algorithms for static priority based preemptive real-time scheduling of periodic task sets. These algorithms are directed at applications in which worst case execution latency is not critical so that priority allocation should be directed towards buffer minimization rather than meeting deadlines. Examples of such applications include video and audio playout/ recording, browsing through a database with audio/ video data and all types of non-interactive real-time applications (we refer to these as *throughput oriented real-time applications*). The techniques developed retain the simplicity of implementation of static priority scheduling while eliminating their drawback of not being able to achieve 100% processor utilization. In addition, we are able to obtain hard bounds on worst case execution time which are low enough for most practical deadline based applications also. We show that the standard *Rate-monotone* priority allocation algorithm is not optimal in terms of our goal of minimising input buffer size. Approximate algorithms are then derived, which perform better than the rate-monotone algorithm and bounds on their performance are derived. Average case performance is studied using simulation. The best performance is seen in an algorithm which combines rate-monotone and shortest job first type of priority orderings. Finally, it is shown that these approaches can also be used to obtain deadline-based scheduling algorithms for task sets with arbitrary deadlines. The existence of an optimal buffer minimization algorithm is left as an open problem.

Keywords: Static Priority Preemptive Scheduling, Real-time Applications, Buffer Minimization

1 Introduction

1.1 Motivation

Multimedia based systems are expected to dominate the workstation environment in the near future. Such systems will be expected to readily handle audio, video and other forms of time-constrained data often referred to as *real-time* data [1], [2]. Real-time data is typically required to be processed within a given *time bound or deadline* for it to be useful. For instance, in order to play out a movie stored on disk, the video processor has to set a playback point. Each frame from the video stream has to be received, processed and made available for display no later than its playback point in order for the user to perceive a continuous and smooth display. Conventional operating systems and processor scheduling techniques do not provide for such real-time constraints. Consequently, real-time processor scheduling techniques have been developed for deadline based task systems. In particular, processor scheduling algorithms for *periodic real-time systems* have been studied extensively [3], [4]. Such task systems consist of a number of jobs (say n jobs numbered as $J_i, i = 1, \ldots n$). Each job J_i arrives (i.e. is ready for execution) periodically with a period T_i. Each such invocation of a job is referred to as a task. Associated with each job J_i is a start time I_i and a deadline D_i. Thus the k^{th} task corresponding to job J_i is ready at time $I_i + (k-1) \times T_i$ and must complete execution by time $I_i + (k-1) \times T_i + D_i$. In the case of multimedia applications, the deadline of a job is set according to some user-dependent criteria. For instance, in a real-time conferencing application, the deadlines could be set according to the maximum allowable latency without having the participants' talk spurts start interfering with each other (as happens in an overseas telephone call for instance). Real-time scheduling techniques can directly be applied in such cases. However in non-interactive type of applications, it is not clear as to how the deadlines are to be set or if any deadlines should be set at all. For example, latency is not important to the user when viewing a movie stored on disk. The only requirement is that playout be smooth and continuous. We refer to such applications as *throughput oriented real-time applications*. In such cases, deadline-based real-time scheduling (by introducing artificial deadlines) may not be the best approach. However, we note that some kind of processor scheduling has to be performed even here because of the on-line nature of many of these applications e.g. a movie is being viewed on-line as it is being retrieved, decompressed etc. Even though latency is not critical, this processing cannot possibly be done offline because of the large amounts of data and high data rates involved. Different scheduling policies will differ in the amount of input buffer requirements. In this paper, we investigate the use of static priority based scheduling algorithms for such applications (Priority based real-time scheduling algorithms are briefly reviewed in the following subsection). The criterion which will be minimized will be the size of the buffer memory required to hold tasks waiting to be processed. We note that this can be an important issue considering that typical video images can have upto the order of a megabyte of data per frame and consequently, having to buffer even a few frames worth of input data will require a lot of workstation memory. Some other examples which may be classified as throughput-oriented real-time applications include encoding a real-time audio/video stream and outputting compressed data to disk, browsing through a audio/video database and all types of non-interactive multimedia applications involving real-time data such as audio and video. Outside the domain of multimedia systems, examples include instrumentation, real-time control and data acquisition systems in which input arrives periodically, and a statistical summary of the data is stored to disk for offline evaluation.

An important characteristic of our analysis is that *we are interested in the peak buffer requirement and not average buffer requirements.* We assume that buffer memory is statically allocated and must account for the maximum instantaneous buffer requirement. This corresponds to a hard real-time system where the buffer must not overflow in any case. This is also true in situations where it is not possible to dynamically allocate and deallocate memory because of the high data rates involved (as in video applications) and a static memory allocation must be made with peak requirements in mind. Consequently, we cannot use results from queueing theory which look at average buffer requirements. For average buffer minimization, shortest job first type allocations are known to be optimal. In contrast to queueing type approaches, the method of analysis used here is similar to that used for scheduling of hard real-time applications [3],[4],[5] i.e. examining a worst case interaction of task invocations.

1.2 Review of Preemptive Real-time Processor Scheduling Algorithms

Preemptive scheduling algorithms for real-time applications are generally classified as being either static priority based or dynamic priority based. Static priority allocation algorithms assign fixed priorities to jobs and at any instant of the schedule, the processor executes the task with the highest priority from among all ready tasks. In a dynamic scheme, the priorities of tasks are allowed to vary during the schedule. Dynamic priority based algorithms are able to obtain full processor utilization but this is not true in general for static algorithms. In this paper, we will always deal with *preemptive resume* type scheduling, in which a task which has been preempted earlier resumes execution at the point where it was preempted (and not re-executed from the beginning). For the case where deadlines are equal to the invocation periods (i.e. each task has to be completed before the arrival of the next task of the same job), the *rate-monotone* priority allocation algorithm has been shown [3] to be an optimal static priority allocation algorithm in the sense that if this algorithm does not lead to a schedule in which every deadline is met (i.e. a feasible schedule), then no priority allocation algorithm will result in a feasible schedule. This algorithm assigns priorities to jobs in increasing order of arrival rate, i.e. the lower the invocation period of a job, the higher is its assigned priority. This algorithm although optimal in the sense described above, cannot always achieve 100% processor utilization. For a large number of jobs, the achievable utilization may be as low as 69% [4]. In contrast, the optimal dynamic priority algorithm for the same problem viz. the *Earliest Deadline First* (EDF) algorithm is always able to find a feasible schedule as long as total utilization of the task set is no more than 100%. At every instant, this algorithm dynamically assigns highest priority to a task which has the earliest execution deadline from among all ready tasks. The *inverse deadline monotone* priority allocation algorithm, has been shown to be optimal for the case of deadlines less than or equal to the invocation periods [7]. Not much work has been reported on static priority scheduling for the case of arbitrary deadlines. Lehoczky [5] has analyzed the performance of the rate-monotone algorithm for the case where all deadlines are a constant multiple of the periods but has also shown that the rate-monotone algorithm is *not* an optimal algorithm even for this special case.

Clearly, using the EDF scheduling algorithm will always ensure schedulability and no buffering will be required because tasks will always complete execution no later than the arrival time of the next task of the same job. However static priority scheduling has some advantages over dynamic schemes such as ensuring that the timing requirements of the most important jobs are met during transient overloads, easier implmentation of task synchronization protocols such as the priority ceiling protocol [4] and most importantly, ease of implementation in processors, I/O controllers and communication switches. Analysis of static priority schemes is hence an important problem.

1.3 Overview of the Paper

In the next section we introduce some definitions which are used throughout the paper. Non-optimality of the standard rate-monotone algorithm for this problem is illustrated with a simple example. In section 3, bounds on the amounts of buffering are obtained and priority allocation algorithms are defined which minimize these bounds. The priority allocation algorithms turn out to be combinations of rate-monotone allocation with other allocation algorithms. Existence of an optimal algorithm is left as an open problem. Section 4 then introduces an approach for obtaining scheduling algorithms for the case of arbitrary deadlines using these approaches. Section 5 describes some simulation experiments which demonstrate the average case behaviour of these algorithms. The best performance is seen in an algorithm which combines rate-monotone and shortest job first type of priority allocation. The concluding section summarizes the results and suggests issues for further investigation.

2 Definitions and the performance of Rate-Monotone Priority Allocation

Consider a set of n periodic jobs. Job J_i has a computation time C_i and is invoked with a period T_i after a start time I_i. In this paper, we refer to each invocation of a job as a task. In case of scheduling with deadlines, each job has a deadline D_i associated with it. For most of this paper, we will consider throughput oriented applications as mentioned earlier and hence there are no task deadlines. Additionally, we will assume that all the start times (the I_i's) are equal to 0. Such job sets are also referred to as being synchronous [7]. *Input buffer size is determined by the number of tasks which have arrived but for which processing has not yet begun.* An inherent assumption is thus that buffer memory required for tasks being processed is not considered. Because of multiprogramming, any priority allocation algorithm will require n units of buffer space for tasks being processed, so only the buffering required for waiting tasks can be optimized. Another way of looking at this is that the input data corresponding to a task is consumed as soon as the task starts processing. We additionally assume that task preemption and context switching times are negligibly small.

Given a set of jobs and an input buffer size, if it is possible to obtain a priority assignment using which the peak buffer requirement can be accomodated, then we will refer to the set as being *buffer-schedulable* with respect to the given buffer size. A given job set can be treated as if it had deadlines equal to the periods. Now, *zero buffering is required if and only if the rate-monotone priority assignment is able to yield a feasible schedule for this job set with these deadlines.* Hence, a task set which requires no buffering will be referred to in the paper as being *LL-schedulable* (for "Liu-Layland Schedulable" [3]).

Lemma 2.1 *A necessary and sufficient condition for the buffered-schedulability of a job set using an appropriately large but finite buffer is that the total processor utilization ($\sum_{i=1}^{n}(C_i/T_i)$) be no more than 1.*

Proof: The necessary condition is straightforward while sufficiency follows from the fact that due to the periodicity of the application, any work conserving scheduler [1] will always be able to "empty out" any tasks accumulated in the buffer in finite time. Hence a *sufficiently large* but *finite* buffer will always do the job. This will be true even if the total utilisation is exactly 100% (in contrast to conventional queueing systems in which waiting times rise infinitely as total utilisation approaches

[1] One that never keeps the processor idle if there is any outstanding processing

100%). $\square$

For simplicity, let us assume that the input data requirement for each task of each job is the same, say c units. (We also look at the unequally weighted jobs problem in the paper). We use the term *late tasks* to denote all active tasks (i.e. those that have already been invoked) for which the previous task of the same job has not yet completed execution. Buffer allotment to incoming tasks can be either partitioned or shared. In a partitioned allotment, each job is assigned a buffer for itself, while in the shared case, a common memory is used for all incoming tasks irrespective of which job they belong to. For the case of equally weighted jobs and shared buffer allotment clearly, minimization of buffer space is equivalent to minimizing the maximum number of late tasks at any instant. For partitioned buffer allotment, it is equivalent to minimizing the sum over all jobs of the maximum number of late tasks of a job at any instant. Let NLT^t denote the total number of late tasks at some instant t and let MLT be its maximum value over the entire schedule. Similarly, let NLT_i^t denote the number of late tasks of job i at time t and MLT_i be its maximum value over the schedule. Clearly, for minimizing partioned buffer memory allotment, we need to minimize $\sum_{i=1}^{n} MLT_i$, while for the shared case, we need to minimize MLT. For most of the paper, we look at the equally weighted jobs case, so that the number of late tasks is itself taken to be a measure of the amount of buffering.

Let the jobs be numbered according to priority so that job J_1 has the highest priority, J_2 has the second highest and so on. Using terminology introduced by Lehoczky [5], we first define a level-i busy period.

<u>Definition :</u> A level-i busy period is a time interval [a,b] within which jobs of priority i or higher are processed throughout [a,b] but no jobs of level i or higher are processed in $(a-\epsilon, a)$ or $(b, b+\epsilon)$ for sufficiently small $\epsilon > 0$.

Lemma 2.2 *There must be at least one instant during the level-i busy period initiated by the critical instant $I_1 = I_2 = \cdots = I_i = 0$ at which there are MLT_i late tasks. Also, there must be at least one instant during the level-n busy period initiated by the critical instant $I_1 = I_2 = \cdots = I_n = 0$ at which there are MLT late tasks.*

<u>Proof:</u> The proof is similar to that of Theorem 1 in [5] and is not included here for lack of space. The rate-monotonic (RM) scheduling algorithm has been shown to be optimal for scheduling job sets when the deadlines equal the invocation periods [3]. The following example shows that it is not an optimal algorithm for minimizing the buffer requirements for buffer-scheduling a given job set.

Example 1: Consider a set of three periodic jobs with the parameters : $C_1 = 20$; $T_1 = 50$; $C_2 = 40$; $T_2 = 70$; $C_3 = 2$; $T_3 = 80$.

If we allocate priorities using RM-ordering, job J_1 will be assigned highest priority followed by J_2 and finally J_3. The completion times and the number of late tasks of job 3, NLT_3^t are shown below starting from the critical instant $I_1 = I_2 = I_3 = 0$. (NLT_1^t and NLT_2^t are zero for all t.)

Task No.	**Task arrival time**	**Task completion time**	NLT_3^t **at completion time** $- \epsilon$
1	00	342	4
2	80	344	3
3	160	346	2
4	240	348	1
5	320	350	0
6	400	692	3
7	480	694	2
8	560	696	1
9	640	698	0

The level-3 busy period ends at time 698, hence we need not examine further. $max_t(NLT_3^t) = MLT_3 = MLT = 4$. Hence, if each task requires c units of buffer space, $4c$ units of buffer memory are required. Consider however, a priority ordering in which J_1 has highest priority, J_3 is next and J_2 is lowest. (Note that this is neither rate-monotone ordering nor shortest job first). The corresponding completion times and NLT_2^t values are shown below (here NLT_1^t and NLT_3^t are zero for all t).

Task No.	**Task arrival time**	**Task completion time**	NLT_2^t **at completion time** $- \epsilon$
1	0	84	1
2	70	144	1
3	140	226	1
4	210	288	1
5	280	350	0
6	350	432	1
7	420	494	1
8	490	576	1
9	560	636	1

Here, only c units of buffer memory are required. Clearly, RM ordering is not an optimal algorithm since it required higher buffering.

Some important observations can be drawn from the above example in connection with obtaining an optimal algorithm for our buffer minimization problem.

- The optimal algorithm for the buffer minimization problem will have to behave like the rate-monotone algorithm for job sets which are LL-schedulable, since this is the only known optimal algorithm for such job sets. (In other words an optimal buffer minimization algorithm must result in zero buffering for LL-schedulable job sets). This eliminates many algorithms including shortest job first since these are not optimal for LL-schedulable job sets.
- For job sets which are *not* LL-schedulable, rate-monotone ordering does not yield minimal buffering as shown in the example above. This implies that the optimal priority allocation algorithm for our problem, if it exists, will have to produce rate-monotone ordering for LL-schedulable job sets and some different ordering for jobs sets which are not LL-schedulable.

3 Bounds on the Amount of Buffering and their Minimization

3.1 A First Upper Bound and a Priority Allocation Strategy

3.1.1 Equally Weighted Jobs

Consider first the case where the storage required to buffer any task of any job is the same. At some time t during the during the $level - i$ busy period initiated by the critical instant, let there be MLT_i late tasks of job i. A **necessary** condition for this to happen is given by relation 1.

$$(\lceil t/T_i \rceil - MLT_i)C_i + \sum_{j=1}^{i-1} \lceil (t)/T_j \rceil C_j > t \tag{1}$$

At time t, there have been $\lceil t/T_i \rceil$ invocations of job J_i. Clearly, since we are dealing with the necessary case, even if all tasks of higher priority have completed execution, there are at least MLT_i tasks of job J_i which have not even started execution (the MLT_i late tasks). Starting from the critical instant $I_1 = I_2 = \cdots = I_i = 0$, the entire period $[0, t]$ is part of the level-i busy period, so there is no processor idle time. Hence relation 1 follows.

Alongwith the necessary condition for schedulability, ...

$$(C_1/T_1) + (C_2/T_2) + \cdots + (C_i/T_i) \leq 1 - \sum_{k=i+1}^{n} (C_k/T_k)$$

...using the identity $\lceil x \rceil < x + 1$ and the fact that the instant t at which MLT_i occurs must be greater than T_i, we can simplify to obtain an upper bound on MLT_i.

$$MLT_i < \frac{\sum_{k=1}^{i} C_k - T_i \sum_{j=i+1}^{n} (C_j/T_j)}{C_i} \tag{2}$$

For the partitioned buffer allocation case, MLT is the sum of the MLT_i's over all i, while for the shared allocation case, the sum of all MLT_i is an upper bound for the maximum number of simultaneous late tasks. Hence, we choose as upper bound *UB1* given by the following expression.

$$MLT < UB1 = \sum_{i=2}^{n} \frac{\sum_{k=1}^{i} C_k - T_i \sum_{j=i+1}^{n}(C_j/T_j)}{C_i} \tag{3}$$

Theorem 3.1 *The upper bound* UB1 *in (3) is minimized if the priority based ordering of the jobs is such that higher priority is assigned to a job with a lower value of* (C_k^2/T_k) *i.e.*

$$C_1{}^2/T_1 \leq C_2^2/T_2 \leq \cdots \leq C_n^2/T_n$$

(We shall refer to this priority ordering as the **ICTM** *ordering for* inverse C^2 by T Monotonic *ordering.)*

<u>Proof :</u> Consider two different priority orderings which differ only in the relative priorities of jobs J_i and J_{i+1}. Let $Ci^2/T_i > Ci + 1^2/T_{i+1}$. Let ordering 1 have J_i at a higher priority than J_{i+1} while ordering 2 has J_{i+1} at a higher priority than J_i. All other jobs have the same priorities in both orderings. Let if possible, the value of *UB1* be lower for ordering 1 than ordering 2. Examination of the R.H.S. of equation 3 shows that each job contributes a term to the overall sum. Clearly, the contributions of jobs other than J_i and J_{i+1} will be the same for both orderings. Hence we have,

$$\begin{aligned} \frac{\sum_{j=1}^{i} C_j - T_i \sum_{j=i+1}^{n} \frac{C_j}{T_j}}{C_i} &+ \frac{\sum_{j=1}^{i+1} C_j - T_{i+1} \sum_{j=i+2}^{n} \frac{C_j}{T_j}}{C_{i+1}} < \\ \frac{\sum_{j=1}^{i-1} C_j + C_{i+1} - T_{i+1}(\sum_{j=i+2}^{n} \frac{C_j}{T_j} + \frac{C_i}{T_i})}{C_{i+1}} &+ \frac{\sum_{j=1}^{i+1} C_j - T_i \sum_{j=i+2}^{n} \frac{C_j}{T_j}}{C_i} \\ \Longrightarrow \frac{C_i^2}{T_i} &< \frac{C_{i+1}^2}{T_{i+1}} \end{aligned}$$

This contradicts our earlier assumption. Hence the value of *UB1* for ordering 1 must be greater than that for ordering 2. Now given any priority ordering, this shows that if the priorities of two jobs are swapped such the job with higher value of C_i^2/T_i is assigned lower priority, the value of *UB1* will always decrease. Hence the value of *UB1* is minimized by the **ICTM** priority ordering. □

We note that as a consequence of the integral nature of the quantities MLT_i, the expression for *UB1* can be simplified to

$$UB1 = \sum_{i=2}^{n} (\lceil \frac{\sum_{j=1}^{i} C_j - T_i \sum_{j=i+1}^{n} (C_j/T_j)}{C_i} \rceil - 1) \tag{4}$$

3.1.2 Unequally weighted Jobs

In general, different jobs will require differing amounts of storage per task buffered. Let $W_1, W_2, \ldots, W_n$ be the weights associated with the corresponding jobs representing the amount of storage required for buffering a task of the corresponding job.

Following a reasoning similar to the equally weighted case, the value of *UB1* can be derived as

$$UB1 = \sum_{i=2}^{n} W_i \times \frac{\sum_{k=1}^{i} C_k - T_i \sum_{j=i+1}^{n} (C_j/T_j)}{C_i} \tag{5}$$

Theorem 3.2 *The upper bound* UB1 *in (5) is minimized if the priority based ordering of the jobs follows the relation*

$$\frac{C_1^2}{W_1 T_1} \leq \frac{C_1^2}{W_1 T_1} \cdots \leq \frac{C_n^2}{W_n T_n}$$

Proof : Similar to that of the equally weighted jobs case (Not included here for lack of space).

We shall refer to this ordering as the *Weighted Inverse C^2 by T* or **W-ICTM** ordering.

From this point on we consider only the equally weighted jobs case as some of the derivations can be extended to the unequally weighted jobs case in a straightforward manner. (e.g. by using W-ICTM ordering wherever ICTM ordering is used and using the appropriate bounds etc.). However in many cases unequal weights make the problem much more difficult and further analysis is required.

3.2 A second upper bound

The upper bound derived in the previous subsection was taken as the sum of the upper bounds on the amount of buffering required by each job. While this is reasonable for the case of partioned buffer allocation, in general there may not be any instant in the schedule at which all jobs have the maximum number of late tasks. Thus for the shared buffer allocation case, a second bound is derived below which has a lower value for some job sets.

Let there be NLT_2^t late tasks of job 2, NLT_3^t late tasks of job 3 and so on up to NLT_n^t late tasks of job n at some time t in the level-n busy period after the starting the schedule at the critical instant at $t = 0$. Then, a necessary condition for this to happen is

$$\lceil t/T_1 \rceil C_1 + \sum_{j=2}^{n} (\lceil t/T_j \rceil - NLT_j^t) C_j > t$$

Making use of the identity $\lceil x \rceil < x + 1$, alongwith the necessary condition $\sum_{j=1}^{n} (C_j/T_j) \leq 1$, we have the necessary condition,

$$\sum_{j=1}^{n} C_j > (NLT_2^t + NLT_3^t + \cdots + NLT_n^t) \times min(C_2, C_3, \ldots, C_n)$$

Since this is true at every instant t, in the level-n busy period, we have the second upper bound.

$$MLT < UB2 = \frac{\sum_{j=1}^{n} C_j}{min(C_2, C_3, \ldots, C_n)} \tag{6}$$

3.3 Combined Priority Allocation Algorithms

In the previous two subsections, upper bounds were derived for the number of late tasks assuming all jobs except the highest priority one overflow (i.e. have some late tasks) at some time. In practice most of the tasks in a job system will require zero buffering. These jobs form a LL-schedulable set and must all execute at higher priorities than the remaining jobs. An optimal algorithm for allocating priorities to these LL-schedulable jobs is ofcourse the rate-monotone (**RM**) algorithm [3]. *The key point is how to partition a given job set into the set which can be LL-scheduled and another set consisting of jobs which will overflow so that the total amount of buffering is minimized.*

Fig. 1 lists an algorithm which we will call the **CP** (i.e. Combined Priority allocation) algorithm. Given a set of jobs S, the algorithm partitions it into sets S_{rm} and S_{other}. S_{rm} consists of jobs which can be LL-scheduled will be allocated priorities using the RM algorithm. S_{other} consists of jobs which will be allocated priorities using a possibly different algorithm. All jobs in S_{rm} will operate at higher priority than jobs in S_{other}. In particular, we will investigate two algorithms for allocation of priorities to the jobs in S_{other} viz. **ICTM** i.e. Inverse C^2 by T Monotonic and **ICM** i.e. Inverse Computation Monotonic algorithms (the corresponding CP algorithms being referred to as CP-I and CP-II).

The CP algorithm can be seen to be a greedy algorithm which picks the job with the largest value of *parameter* from the current set S_{rm} and tests the remaining jobs for LL-schedulability. In the next few subsections, we will see that with appropriate choice of the value of *parameter*, the CP algorithm will minimize the upper bounds derived in the earlier subsections. However, we need to modify the expressions for *UB1* and *UB2* to reflect the fact that only a few jobs can overflow.

/* S denotes the given job set;
S_{rm} denotes the set of jobs allocated priorities in RM order;
S_{other} denotes the set of jobs allocated priorities using the ICTM/ ICM algorithms */
$\mathbf{S_{rm}} = \mathbf{S}; \mathbf{S_{other}} = \phi$
Evaluate $parameter_i$, $i = 1, \ldots, n$

Step 1: Test the jobs in $\mathbf{S_{rm}}$ for LL-schedulability using the necessary and sufficient tests derived in [6]. If successful, goto Step 3.
Step 2: From the set S_{rm} move the job with the largest value of $parameter_i$ to the set S_{other}. Goto Step 1.
Step 3: Schedule jobs in S_{rm} using Rate monotonic priority ordering, and jobs in S_{icm} using ICTM/ ICM priority ordering.

Figure 1: **The Combined Priority Allocation Algorithm**

Upper bound *UB1* is simply the sum of the maximum amounts of buffering required by each job. If on the execution of the CP-I algorithm, k jobs are found to be in S_{rm} and the remaining $n-k$ in S_{other}, *UB1* for equally weighted jobs will become

$$UB1 = \sum_{i=k+1}^{n} (\lceil \frac{\sum_{j=1}^{i} C_j - T_i \sum_{j=i+1}^{n} (C_j/T_j)}{C_i} \rceil - 1) \tag{7}$$

Similarly it may be verified that the second bound *UB2* becomes

$$UB2 = \lceil \sum_{j=1}^{n} (C_j/min(C_{k+1}, \ldots, C_n) \rceil - 1 \tag{8}$$

We note that both bounds are tight. For instance in case of the example job set of section 2, both bounds yield a value of 1 for MLT which is its actual minimal value. In general for a given job set, $UB_{min} = min(UB1, UB2)$, (the minimum of the two upper bounds) should be used as the bound on the amount of buffering.

3.4 Some properties of the CP Algorithms

3.4.1 Minimization of upper bounds & tightness of bounds

Theorem 3.3 *The CP-II algorithm minimizes the upper bound* UB2 *defined in (8).*

Proof : The value of *UB2* can be seen to be inversely proportional to the minimum value of C_i from all jobs in the set S_{other}. If the algorithm terminates with k jobs in the set S_{rm} and the jobs are reordered according to their new priorities, a lower value for *UB2* would have been possible only if the algorithm had terminated after transferring one of the jobs J_{k+2} through J_n to the set

S_{other}. Clearly this is not possible else the CP-II algorithm would not have needed to move the job J_{k+1} to the set S_{other}. $\square$

3.4.2 Worst Case Buffer Requirement & Optimality for the 2 Job Case

Lemma 3.1 *If priorities are allocated using the CP-II algorithm, $MLT_i \leq i, \forall i$ the worst case buffer requirement is in $O(n^2)$ and can be as low as $O(n)$.*

Proof: From relation 2, when $C_i \geq C_j, j < i$, $MLT_i \leq i$. Hence, $MLT \leq \sum_{i=1}^{n} MLT_i = n(n-1)/2$. Also, from (8) $MLT \leq n\, C_{max}/\, C_{min}$ which is approximately in $O(n)$ when the ratio of maximun to minimum C values is small.

Lemma 3.2 *The CP-II algorithm is an optimal algorithm for minimizing the amount of buffering required by a two job application.*

Proof : The detailed proof is omitted for lack of space. Briefly however, this may be verified by examining equation 8 for $C_2 > C_1$ and checking that the CP-II ordering will result in only unit buffer requirement for non-LL-schedulable job sets which is the minimum possible.

3.4.3 Time Complexity & Polynomial time algorithms

The time complexity of both the CP-I and CP-II algorithms is in $O(nTmax)$ (Tmax is the maximum period of any job in the set). Hence these are pseudopolynomial algorithms (polynomial running time if the maximum period is bounded). This is mainly because of the pseudo-polynomial time complexity of the schedulability check in step 1 of fig 1. A CP-like priority allocation algorithm based on RM-ordering can also be derived. We will refer to this as the CP-RM algorithm. This algorithm would work exactly as in Fig 1, with the value of $parameter_i = T_i$ i.e. jobs are moved to the set S_{other} in order of decreasing periods.

Polynomial algorithms can also be derived along the same lines. If in step 1 of the CP algorithm (Fig. 1) , instead of using the necessary and sufficient conditions of [6] we use the sufficient conditions presented in [3] using only the worst-case total utilization bounds, the running times of the CP-I, CP-II and CP-RM algorithms become polynomial. Liu & Layland [3] have proved that an n job set is LL-schedulable if the total utilization is no more than the value $n(2^{1/n} - 1)$. Using this sufficient test at each step 1 will result in the CP algorithms having a running time in $O(n^2)$ ($O(n)$ operations for the evaluation of total utilization at most $O(n)$ times). We will refer to these algorithms as the P_CP-I, P_CP-II and P_CP-RM algorithms algorithms. These polynomial-time algorithms are expected to result in higher amounts of buffering than their pseudo-polynomial versions since in general more jobs will be transferred to the set S_{other} than necessary.

3.5 A third upper bound for the P_CP-RM algorithm

Lehoczky [5] has derived worst-case utilization bounds for scheduling jobs with deadlines given by $D_i = \Delta \times T_i$ using rate-monotone priority ordering. According to this formulation, a job set with deadlines of the form $D_i = \Delta \times T_i, \forall i$ is schedulable relation 9 below is satisfied.

$$\sum_{j=1}^{n} \frac{C_j}{T_j} \leq \Delta(n-1)((\frac{\Delta+1}{\Delta})^{1/(n-1)} - 1), \quad \Delta = 2, 3, \ldots \tag{9}$$

Clearly, if each task of a job J_i can execute for a maximum time ΔT_i, no more than $\Delta - 1$ tasks of the same job will need to be buffered. On execution of the P_CP-RM algorithm, if k jobs are found to be LL-schedulable, a sufficient amount of buffering for equally weighted jobs, is given by the upper bound below (multiplied by the amount of storage per task).

$$\begin{aligned} UB3 &= (n-k+1) \times (\Delta_{min} - 1) \qquad (10) \\ \Delta_{min} &= min\{\Delta | \sum_{j=1}^{n} \frac{C_j}{T_j} \leq \Delta(n-1)((\frac{\Delta+1}{\Delta})^{1/(n-1)} - 1), \Delta = 2, 3, \ldots\} \quad k = |S_{rm}| \end{aligned}$$

4 A Static Priority Scheduling Algorithm for High Utilization job sets

Here, by a high utilization job set, we imply a set which is not LL-schedulable since job sets with total utilisations below the Liu & Layland bound can always be scheduled by the rate-monotone algorithm. Currently, the only known approach to this problem is a sufficient (but not necessary) condition based on the worst case utilization bounds formulated by Lehoczky [5] as discussed above. The bounds in [5] have been derived assuming rate-monotone ordering of job priorities even though it is known that this is not an optimal ordering. In addition, it is valid only for the case where the job deadlines are a constant multiple of the invocation periods independent of the job ($D_i = \Delta T_i, \forall i$).

Consider the problem of finding a priority allocation for a job set in which the deadlines are arbitrary (less than, equal to or greater than the invocation periods) such that all deadlines are met in the resultant schedule. In using the worst case utilization bound based approach, one would find a minimal Δ such that $D_i \leq \Delta \times T_i, i = 1, \ldots n$. If then, total job utilization is no more than the worst case bound derived using Δ_{min} as found in 11, the job set will be deemed schedulable using rate-monotone priority ordering.

The buffer-minimization problem with equally weighted jobs is related to this scheduling problem. For instance, if we know that a job set is buffer-schedulable with some values for the MLT_i's then if each job J_i has a deadline $D_i \geq (MLT_i + 1) \times T_i$, all tasks will meet their deadlines (recall that MLT_i can be seen as the maximum number of full invocation periods that the execution of a task of job J_i can span). Relation 2 can be used to obtain the MLT_i. Note that this approach is also only a sufficient test for schedulability of job set with arbitrary deadlines.

5 Simulation Results

None of the algorithms discussed in this paper have been proved to be optimal in terms of yielding the minimal buffering required for a given job set. These algorithms minimize the upper bounds on buffering which we have derived and the upper bounds have also been shown to be tight. Also the CP-II algorithm has been shown to be optimal for the two job case. Hence we expect that heuristics based on these algorithms will also perform well in terms of minimizing the actual amount of buffering. Simulations can help in better characterizing their performance on an average over a large number of job sets.

5.1 Simulation Procedure

Job sets with up to 25 jobs were examined. For each value of n (the number of jobs), a total of 250 different job sets were generated with the total utilization varying uniformly between $WCUB$ and 1 where $WCUB$ is the worst case upper bound of Liu & Layland for LL-schedulability [3]. This was done since no buffering is required anyway for job sets with total utilization below this bound.

A random search algorithm (RAND_OPT) was additionally used to get some idea of the "optimal" amount of buffering. This algorithm *started with the minimum buffering already obtained by the other algorithms being compared and tried random permutations of the priority ordering in order to try and find a lower value for the buffering required.* For each priority ordering tried, this

Key: A: Random Search B: CP-II C: CP-I D: CP-RM

Figure 2: Buffering required versus number of jobs (Equally weighted jobs, Shared Buffer Allocation)

algorithm determined the exact amount of buffering by executing the entire schedule and computing the maximum instantaneous value of number of late tasks (executing each such schedule takes exponential time). Only $5n$ random permutations were examined for each set of n jobs in order to obtain an approximate estimate of the optimal amount of buffering without performing an exhaustive search.

5.2 Analysis of results

Fig. 2 shows the average values of (peak) buffering required by the CP-I, CP-II, CP-RM and RAND_OPT algorithms. For the first 3 algorithms, the average over all job sets of UB_{min} (the minimum of the upper bounds *UB1* and *UB2*) for the priority ordering produced by the algorithms, is plotted for each value of the number of jobs. The values obtained for the RAND_OPT algorithm are the actual values using the exponential time procedure described above.

- Among the three algorithms, the values of the bounds are lowest for the priority ordering produced by the CP-II algorithm. The values are generally the highest for the CP-RM ordering. This is in accordance with the fact that the CP algorithms were specifically designed using the formulations for the upper bounds in mind. The CP-II ordering minimizes the value of *UB2* while the CP-I ordering produces a low value of *UB1* in general as discussed earlier.

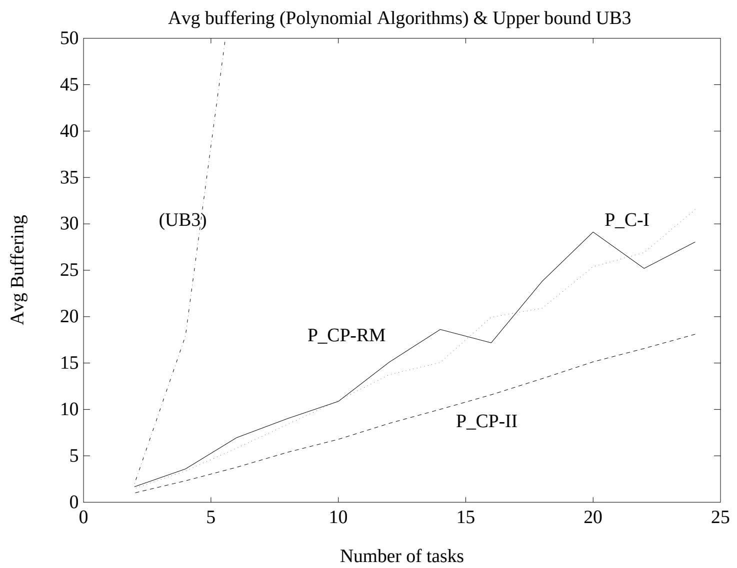

Figure 3: Averages of Peak Buffering required by the Polynomial-time Algorithms & Avg *UB3* values

- The average values of the bounds as produced by the various algorithms vary from 2 to 6 times the value obtained by the exponential time RAND_OPT algorithm indicating that there is some room for better algorithms. However, overall the values were quite low since even at high values of total utilization, there are many job sets which are LL-schedulable and hence require no buffering at all. (This is in accordance with the results in [4] in which the average total utilisation for LL-schedulability was found to be about 83%.) The average values produced by the CP-II algorithm are between 20 and 50 % of those produced by the other algoritthms.

Next the relative performance of the polynomial versions of the same algorithms was evaluated. Fig. 3 show the plots of the amounts of buffering required as predicted by the P_CP-I, P_CP-II and P_CP-RM algorithms. As mentioned in the previous section, a third upper bound *UB3* can be formulated for the CP-RM algorithm. The fourth curve shows the the values of the buffering required as predicted by this bound. The *UB3* values are very high and out of range for most of the values of n. The actual values can be obtained from table 5.2.

- Here too, the polynomial version of the CP-II algorithms viz. the P_CP-II algorithm results in the least amount of buffering (requiring on the average 40 to 60 % of the buffering required by the other algorithms). These figures are considerably higher than those in fig. 2. These

n	CP-I	CP-II	CP-RM	UB3
2	1.7	1.0	1.5	2.0
4	3.6	2.3	3.4	17.9
6	7.0	3.8	5.8	58.9
8	9.0	5.4	8.4	60.1
10	10.9	6.8	11.0	149.0
12	15.1	8.5	13.8	174.6
14	18.6	10.0	15.1	308.5
16	17.2	11.6	19.9	296.4
18	23.8	13.3	20.9	288.7
20	29.1	15.1	25.4	737.3
22	25.2	16.6	26.9	539.4
24	28.0	18.1	31.5	452.9

Table 1: Averages Values of Peak Buffering required by the Polynomial-time Algorithms & avg *UB3* values

figures seem to indicate that on the average the required buffering as predicted by the pseudo-polynomial CP-I, CP-II and CP-RM algorithms is approximately an order of magnitude less than that predicted by the polynomial time versions.

- The required buffering as predicted by *UB3* for RM-ordering is much more than that predicted by *UB1* or *UB2* for any ordering (P_CP-I, P_CP-II or P_CP-RM). This seems to indicate that the formulation based on worst case utilization is very pessimistic. These values would also seem to indicate that the scheduling algorithm introduced in section 4 would do much better than one based on worst case utilization bounds. A possible explanation of this observation can be made by examining relation 9. As the total job utilization approaches unity, the value of Δ required as predicted by this formula shoots up very rapidly. Since in our simulations, for each value of n, we simulate about 25 job sets for 10 different total utilization values uniformly distributed between the Liu & Layland bound and unity, the job sets with total utilization close to 1 will result in very large values of Δ, and dominate the resulting average values of required buffering (since the buffering required directly increases with the value of Δ).

6 Conclusions

We have looked at the problem of static priority allocation for the minimization of buffer space for a class of multimedia applications which we have referred to as being throughput oriented. Examples of such applications include audio and video playout/ recording, multimedia database browsing and any kind of non-interactive multimedia application involving periodic real-time data such as audio and video. Worst case execution latency is not critical in such applications and deadline based real-time scheduling should not be used. Instead we modify real-time processor scheduling techniques to result in reduced buffer memory requirement which is a more important criterion than latency in such applications. Using these techniques we are able to utilize the simplicity of implementation of static priority based scheduling techniques and also achieve 100% processor utilization (which is not true in general for conventional deadline based static priority schedulers). In addition the latencies are observed to be low enough for these techniques to be usable even for interactive multimedia applications. Upper bounds on the amounts of buffering required were obtained and heuristics derived which minimize these upper bounds. Some properties of these heuristics were analyzed. The complexity of these heuristics being pseudo-polynomial, polynomial time versions of the same algorithms were developed. The performance of these heuristics was then studied using simulation. Their performance was compared to that of an exponential time randomized "global" search based algorithm (which was expected to produce close to optimal values). The CP-II algorithm (a combination of rate-monotone and shortest job first priority orderings) was observed to have the best performance yielding buffer values between 50 and 75% of those required by the standard rate-monotone based priority ordering. The randomized search algorithm produced buffer values less than half of those produced by the approximate heuristics indicating that there is room for improvement of these heuristics. In case of the polynomial-time versions of these algorithms, buffer requirements were found to be much higher (of the order of the number of jobs). Here too, the polynomial-time version of the CP-II algorithm had the best performance.

The priority allocation algorithms for buffer minimization were also shown to be usable to obtain real-time scheduling algorithms for job sets with arbitrary deadlines (not necessarily related to the invocation periods in any way). Analysis of the buffering requirements of these approaches indicate that these require much less buffer space than that required by existing scheduling algorithms that have been proposed for this case, indicating much lower response times and consequently improved schedulability. Scheduling algorithms based on input buffer minimization techniques will also be applicable to a more general class of applications since the deadlines will not be restricted to be a multiple of the task periods as in [5]. A detailed evaluation of this approach was beyond the scope of this paper and is a topic for further investigation. The question of intractability of the problem (obtaining an optimal priority allocation in polynomial time, which minimizes buffering requirements) is open. Getting an optimal algorithm seems difficult even for the case of a synchronous job system and equally weighted jobs. Detailed analysis of the unequally weighted jobs case is also an issue for further work.

References

[1] E.A. Fox, "The Coming Revolution in Interactive Digital Video," *Commun. of the ACM*, July 1989, pp.794-801.

[2] P.V. Rangan, H.M. Vin and S. Ramanathan, "Designing an On-Demand Multimedia Service," *IEEE Communications Mag.*, July 1992, pp.56-65.

[3] C.L. Liu and J.W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, **20**, 1973, pp. 46-61.

[4] L. Sha and J. Goodenough, "Real time scheduling theory and ADA," *IEEE Computer*, April 1990, pp. 53-62.

[5] J.P. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," *Proc. IEEE Real Time Systems Symposium*, 1990, pp. 201-209.

[6] J.P. Lehoczky, L. Sha, and Y. Ding,"The rate monotonic scheduling algorithm: Exact characterization and average case behaviour," *Proc. IEEE Real Time Systems Symposium*, 1989, pp. 166-171.

[7] J. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic real-time tasks," *Performance Evaluation*, **2**, 1982, pp. 237-250.