Integrated Real-Time Resource Scheduling

Kartik Gopalan Tzi-cker Chiueh

Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400

{kartik, chiueh}@cs.sunysb.edu

Abstract

Real-time applications that utilize multiple system resources, such as CPU, disks, and network links require *coordinated* scheduling of these resources in order to meet their end-to-end performance requirements. Most state-of-the-art operating systems support at best independent resource allocation and deadline-driven scheduling but lack coordination among individual heterogeneous resources. This paper describes the design and implementation of an Integrated Real-time Resource Scheduler (*IRS*) that performs coordinated allocation and scheduling of multiple heterogeneous resources on the same machine. The novel feature of *IRS* is a multi-resource allocation mechanism that maximizes overall resource utilization efficiency while ensuring that end-to-end timing constraints of the real-time application be satisfied. More specifically, *IRS* assigns a delay budget to each task in an application according to the current load and predicted demand on individual resources. In the *IRS* model, real-time applications are modeled as a periodic execution of their Task Precedence Graph (TPG). Each task in the TPG corresponds to the consumption of a particular resource. The TPG, its period, jitter and "bandwidth" requirements of individual tasks, are specified explicitly when an application registers itself as a real-time process. At run-time, a *global scheduler* dispatches the tasks of the real-time application to corresponding resource schedulers according to the precedence constraints between tasks. Hence, individual resource schedulers can make scheduling decisions locally and yet collectively are able to satisfy a real-time application's end-to-end performance requirement.

1 Introduction

A network video server reads a group of compressed video frames from local disk, processes the video data (e.g., transcoding or frame skipping), and transports the processed data across the network to the requesting client. The entire process repeats itself over the lifetime of the application. In this example multimedia application, there are several tasks each of which uses a different system resource, and is dependent upon the successful completion of previous tasks in the sequence. In other words, there is a *precedence ordering* among the tasks in the application. Finally, to achieve reasonable perceptual quality, the entire sequence of tasks needs to be completed within a certain period of time. Similar behavior can be observed in aperiodic applications such as client request processing in web server clusters where each request may require multiple system resources and a guaranteed response time. The above examples typify the following distinct properties shared by many soft real-time applications:

- Uses of multiple heterogeneous resources each have a specific performance requirement.
- Uses of resources within an application are strictly ordered and form a dependency graph.
- Execution of a repetitive sequence of tasks requires guaranteed time-bound completion.

While real-time scheduling for a single system resource, such as CPU, disk, and network, has been studied extensively in the real-time and more recently multimedia computing community, the issues of efficient resource allocation and coordinated scheduling of multiple heterogeneous system resources on a single machine have not received the attention they deserve. This paper presents the design and implementation of an Integrated Real-time Resource Scheduling system called *IRS* that is designed specifically to address the multi-resource allocation problem, in order to support true end-to-end application-level real-time performance guarantees.

In the *IRS* model, application programmers specify the performance requirement of each resource that the application uses, the precedence ordering (TPG) among the uses of resources, and the TPG's delay bound. We call each use of a distinct system resource as a *task*. Given these information for each real-time application, *IRS* assigns deadlines to each task in the TPG such that the TPG's delay bound is met and the overall system resource utilization efficiency is maximized. The *IRS* model is equally applicable to periodic and aperiodic real-time applications. However for simplicity of presentation, we will consider only periodic real-time applications in the rest of the paper. Since programmers are relieved from the responsibility of managing deadlines of individual tasks, *IRS* also simplifies the development of real-time applications.

The novel and most important component of *IRS* is its resource allocation and deadline assignment algorithm. Given the TPG and its delay bound for periodic execution, *IRS* automatically computes delay budget (deadline in each period) for each task such that the entire TPG must be completed within its delay bound. More concretely, the deadline assignment algorithm apportions the *slack* in the delay budget among an input TPG's tasks according to the current and predicted load of each of the resources in the system. The goal of this slack allocation algorithm is to maximize the number of real-time applications that can be admitted into the system by reducing the extent of load imbalance among different resources.

Another feature of *IRS*'s programming model is the support for implicit resource requirement specification. Although it may be reasonable to expect application programmers to specify the disk bandwidth and network bandwidth requirement of their applications, it is relatively difficult for application programmers to accurately estimate the application's CPU requirement. To alleviate this problem, *IRS* allows multimedia application programmers to leave their CPU resource requirements unspecified, and transparently determines the CPU requirements through dynamic measurements. The CPU requirements in this case are implicitly specified by the application's code, rather than through explicit specifications. An application exploiting implicit resource requirement specification is first placed in the *probation* mode. After a number of periods, the application's implicitly specified resource requirements become clearer, and *IRS* can determine whether the application should be promoted to *real-time* mode according to the admission control decision. In addition, a resource usage monitor constantly keeps track of the resource usage of each real-time application, and those applications whose resource consumption significantly deviates from the explicit or implicit specifications will be informed to take proper actions and potentially down-graded to the *best-effort* class.

Unlike many real-time operating systems discussed in Section 5, the design of *IRS* is *not* meant to be a comprehensive real-time operating system. So it does not include support such features as priority inheritance, high-resolution timers, or real-time synchronization primitives. Rather, the main emphasis of

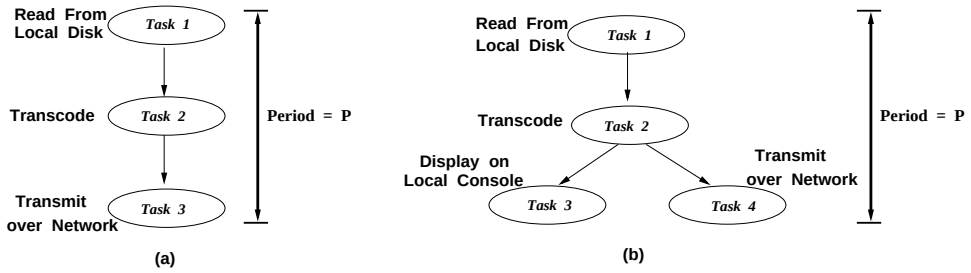

Figure 1: *Task precedence graphs (TPG) for video playback applications. (a) Linear TPG - video frames are read from the local disk, transcoded, and transmitted over the network. (b) General TPG - video frames are read from the local disk, transcoded, and then displayed on the local console as well as transmitted over the network. The entire TPG has to be completed within period P.*

this work is to advocate the importance of an integrated framework for allocation and scheduling of multiple resources in a real-time system in order to provide end-to-end performance guarantees.

The rest of the paper is organized as follows. In Section 2, we describe a new resource allocation algorithm that *IRS* uses to efficiently allocate system resources among periodic real-time applications. Section 3 explains *integrated* scheduling in *IRS* based on *global* and *local* resource schedulers, the programming model and the software architecture of *IRS* in our prototype implementation. Section 4 presents performance results demonstrating the effectiveness of *IRS*. Section 5 reviews related work in this area. Section 6 provides a summary of main research results and Section 7 gives an outline of the future research that we plan to do.

2 Resource Allocation and Deadline Assignment

In this section we first present the motivation for addressing the efficient resource allocation and deadline assignment problem. Next, we propose a load-based slack allocation scheme for solving this problem.

2.1 Motivation

A typical real-time multimedia application periodically executes a set of *tasks* that are related to each other through *precedence ordering constraints*. For instance, Figure 1(a) shows the linear TPG of a video playback application that executes three tasks every period. Task 1 is a *disk read* operation which reads a video frame from local disk. This data is then transcoded (Task 2) and transmitted over the network to a remote client (Task 3). Figure 1(b) shows a more general TPG in which the transcoded video is displayed on the local console (Task 3) and also transmitted over the network to a remote client (Task 3). In the latter case, Tasks 3 and 4 cannot begin till Task 2 completes the transcoding operation. However, once Task 2 is completed, Tasks 3 and 4 can proceed concurrently since one does not depend on the completion of the other.

A task precedence graph describes only the partial-ordering but not the timing relationships among tasks. For instance, the video playback application may need to display video frames once every periodic interval. This application-level performance requirement imposes the timing constraint that all tasks in the TPG must complete within each period or cycle. To guarantee application-level QoS for such an application requires more than real-time scheduling for each individual resource, which can only guarantee task-level QoS.

For the multimedia application shown in Figure 1(a), Task 2 cannot begin till Task 1 completes and Task 3 cannot begin till Task 2 completes. Therefore in each cycle Task 1 must be completed sufficiently early to leave enough time for Tasks 2 and 3 to complete before the end of the period. In other words, total time budget for Tasks 1, 2 and 3 must be smaller than P. But how does one assign time budgets to tasks in a TPG so that the system can satisfy both the precedence and timing constraints among the tasks, and at the same time maximize the overall resource utilization efficiency? This problem is called the *Task Deadline Assignment* problem.

It is well known in real-time resource scheduling literature that for a given throughput, a tighter latency bound requirement leads to higher resource requirement. Assigning a deadline to a task is equivalent to imposing a latency bound because a task's *ready* time is the same as the deadline time of the task it depends on according to the TPG. Therefore, assigning deadline to a task also entails a specification of the load

requirement on that tasks's corresponding resource. The key to maximize the overall utilization efficiency of a multi-resource system is to balance the load on individual resources. As a result careful task deadline assignment can significantly improve the overall system resource utilization by taking into account the current loads and predicted future demands on different resources.

In general, most real-time applications can be modeled with linear TPGs, i.e., tasks are ordered one after another, as in Figure 1(a). In this case, the task deadline assignment problem has a *direct solution* that will be presented in Section 2.3. The more general case of TPG being a directed acyclic graph, as in Figure 1(b) is less common and does not admit to a simple direct solution. For this case, we have developed an iterative algorithm that converges to a *heuristic solution* of task deadline assignment. However, this approach is computationally more expensive than the direct solution for the linear TPG. The iterative solution is presented in Section 2.4.

2.2 Notations

Before we formally present the task deadline assignment problem and its solution, let us introduce the following notations:

m The number of resources in the system.

C_r The capacity of resource r, such as CPU mips rating, disk bandwidth, or network link bandwidth.

A_i The i-th application.

u_i The number of tasks in the i-th application.

P_i The period of the i-th application.

A_{ij} The j-th task of the i-th application.

R_{ij} The ID of the resource used by the task A_{ij}.

w_{ij} The amount of work (or resource) the task A_{ij} performs every period.

W_{ir} The total amount of work required from resource r by application A_i every period. $W_{ir} = \sum_{\{j|R_{ij}=r\}} w_{ij}$.

t_{ij} The delay budget, in seconds, that is assigned to the task A_{ij}.

T_{ir} The total delay budget, in seconds, that is assigned to tasks in A_i that use resource r. $T_{ir} = \sum_{\{j|R_{ij}=r\}} t_{ij}$. Conversely, $t_{ij} = (w_{ij}/W_{ir}) * T_{ir}$, where $R_{ij} = r$.

l_{ij} The minimal delay budget, in seconds, that could be allocated to the task A_{ij} when the application A_i is admitted.

L_{ir} The total minimal delay budget, in seconds, that could be allocated to tasks in A_i that use resource r. $L_{ir} = \sum_{\{j|R_{ij}=r\}} l_{ij}$. Conversely, $l_{ij} = (w_{ij}/W_{ir}) * L_{ir}$, where $R_{ij} = r$.

2.3 Linear Task Precedence Graphs

In this section, we describe the task deadline assignment problem in the case of linear TPGs and present a direct solution.

2.3.1 Typical Real-time Application

We first define the concept of a *typical real-time application* that will be used in following exposition. A *typical real-time application* in a system is defined as one which closely models the resource demand pattern of actual applications that may arrive in the future. A typical real-time application is represented by the set of typical workloads $\{W_{typ_1}, W_{typ_2}, \ldots, W_{typ_m}\}$ and corresponding delay budgets $\{T_{typ_1}, T_{typ_2}, \ldots, T_{typ_m}\}$ such that

$$T_{typ_1} + T_{typ_2} + \cdots + T_{typ_m} = 1 \quad (1)$$

In other words, W_{typ_r} is the typical amount of work required from resource r every second.[1] A typical application in a system could be known statically if all arriving real-time applications are known to be identical. In case they are not known statically, the typical application's characteristic could be dynamically estimated from the history of resource demand pattern of earlier applications.

In *IRS* we dynamically estimated the typical application's characteristics as follows. To determine W_{typ_r}, we first calculate the median of the normalized demands on resource r, W_{ir}/P_i, $1 \leq i \leq K$, of K applications admitted into the system. W_{typ_r} is then calculated as the average of resource demands over a small window of applications around the median, the rationale being that a small window around the median would contain the most representative values of resource demands for future applications. The size of the window is configurable according to expected workload variations. We will show later in Section 2.3.4 that the individual values of T_{typ_r} do not need to be calculated. In the rest of this section, we assume that the information about typical workloads $\{W_{typ_1}, W_{typ_2}, \ldots, W_{typ_m}\}$ is available in the system.

2.3.2 Task Deadline Assignment Problem

Assume there are $N-1$ applications already in the system and application A_N, with period P_N, arrives for admission into the system. Further assume that application A_N has a linear TPG, i.e., tasks are ordered one after another, and that A_N requires works $\{W_{N1}, W_{N2}, \ldots, W_{Nm}\}$ from each of the m resources. We need to find delay budget assignments $\{T_{N1}, T_{N2}, \ldots, T_{Nm}\}$ such that

$$T_{N1} + T_{N2} + \cdots + T_{Nm} \leq P_N \tag{2}$$

and the number, n, of *typical applications* admitted into the system after application A_N is admitted, is maximized.

2.3.3 Admission Control

The latitude in assigning deadline to a task depends on the availability of the resource that the task requires. The admission control decides first whether the application A_N could be admitted without affecting existing real-time applications' performance guarantees. If so, the delay budgets, i.e., t_{Nj}'s, assigned to individual tasks in application A_N, are computed so that A_N can be completed within its period P_N.

The amount of resource r consumed by any application A_i is W_{ir}/T_{ir}. Before application A_N arrives, the available capacity remaining in resource r is $C_r - \sum_{i=1}^{N-1} \frac{W_{ir}}{T_{ir}}$. The minimal delay budget, L_{Nr}, could be assigned from resource r to A_N, if the remaining capacity of the resource r is dedicated to completing the work W_{Nr}. That is,

$$L_{Nr} = \frac{W_{Nr}}{C_r - \sum_{i=1}^{N-1} \frac{W_{ir}}{T_{ir}}} \tag{3}$$

To determine whether the available resource is sufficient to accommodate the application A_N's resource requirements, the following check based on minimal delay budgets is performed:

$$\sum_{r=1}^{m} L_{Nr} \leq P_N \tag{4}$$

If the above condition holds, the system has enough resources to complete the task graph of A_N within its specified period. Otherwise it does not have sufficient resources and A_N should be rejected.

2.3.4 Deadline Assignment

In the case that the period, P_N, is larger than the sum of L_{Nr}'s, it means there is a slack in the delay budgets assigned to A_N, given by

$$S_N = P_N - \sum_{i=1}^{m} L_{Ni} \tag{5}$$

One can divide this slack among individual tasks of A_N to reduce each task's actual resource demand. A simple scheme to divide this slack would be to give equal share of slack to each task. Let's call this scheme

[1] We choose one second as the RHS of Equation 1 in order to normalize the period lengths of different applications.

Equal Slack Allocation (ESA). However, ESA ignores the current load and possible future demand on each resource thus leading to possible load imbalance among resources. For example, under ESA some resources may be exhausted much earlier than others. Another way is to apportion the slack based on current load and predicted demands on each resource such that the number of typical applications that can be admitted in the future is maximized. We describe such a slack allocation scheme below and call it *Load-based Slack Allocation* (LSA).

Before application A_N's arrival, the available capacity, Y_r, of each resource r can be expressed as

$$Y_r = \frac{W_{Nr}}{L_{Nr}} \tag{6}$$

This is because L_{Nr} is the minimal delay budget allocated to A_N from resource r if all the remaining capacity of resource r is assigned to the task A_N. After A_N is admitted, the available capacity, Y'_r, of each resource r would be

$$Y'_r = Y_r - \frac{W_{Nr}}{T_{Nr}} \tag{7}$$

The number of typical applications, n, that can be admitted in the system, after A_N is admitted, is given by the minimum of the number of typical applications that each resource can admit individually, i.e.,

$$n = \min_{r=1}^{m}(\frac{Y'_r}{(W_{typ_r}/T_{typ_r})}) \tag{8}$$

The *optimization goal* of the deadline assignment is to maximize the number of applications admitted, n, given by Equation 8. In conjunction with Equation 1, it can be shown that at the point which *maximizes the minimum* in Equation 8, the following equality holds.

$$n = \frac{Y'_1}{\frac{W_{typ_1}}{T_{typ_1}}} = \frac{Y'_2}{\frac{W_{typ_2}}{T_{typ_2}}} = \cdots = \frac{Y'_m}{\frac{W_{typ_m}}{T_{typ_m}}} \tag{9}$$

Equation 9 states that at the point of maximum admitted typical applications, equal number of typical applications will be admitted by each resource individually. From Equations 1 and 9, we can solve for n as follows.

$$n = \frac{1}{\frac{W_{typ_1}}{Y'_1} + \frac{W_{typ_2}}{Y'_2} + \cdots + \frac{W_{typ_m}}{Y'_m}} \tag{10}$$

From Equations 7 and 10, we have

$$n = \frac{1}{\frac{W_{typ_1}}{Y_1 - \frac{W_{N1}}{T_{N1}}} + \frac{W_{typ_2}}{Y_2 - \frac{W_{N2}}{T_{N2}}} + \cdots + \frac{W_{typ_m}}{Y_m - \frac{W_{Nm}}{T_{Nm}}}} \tag{11}$$

Since the number of typical applications that can be admitted, n, depends on the delay budget allocations T_{N1}, T_{N2}, ..., T_{Nm}, we need to find the combination of these delay budget allocations that maximizes n. In other words, we need to minimize the inverse of n, given by

$$n^{-1} = \frac{W_{typ_1}}{Y_1 - \frac{W_{N1}}{T_{N1}}} + \frac{W_{typ_2}}{Y_2 - \frac{W_{N2}}{T_{N2}}} + \cdots + \frac{W_{typ_m}}{Y_m - \frac{W_{Nm}}{T_{Nm}}} \tag{12}$$

From Equations 12 and 2 we have

$$n^{-1} = \frac{W_{typ_1}}{Y_1 - \frac{W_{N1}}{T_{N1}}} + \frac{W_{typ_2}}{Y_2 - \frac{W_{N2}}{T_{N2}}} + \cdots + \frac{W_{typ_m}}{Y_m - \frac{W_{Nm}}{P_N - T_{N1} - T_{N2} - \cdots - T_{N(m-1)}}} \tag{13}$$

In order to minimize n^{-1}, we need to find the partial derivatives $\frac{\partial n^{-1}}{\partial T_{Nr}}$ for $1 \leq r \leq (m-1)$ and equate each derivative to 0. Setting the partial derivative $\frac{\partial n^{-1}}{\partial T_{Nr}}$ to 0 means that, given T_{Ns}, for each resource $s \neq r$, we want to determine the value of T_{Nr} that will minimize n^{-1}. Hence, for $1 \leq r \leq (m-1)$, we have

$$\frac{\partial n^{-1}}{\partial T_{Nr}} = \frac{W_{typ_r} W_{Nr}}{t_{Nr}^2 (Y_r - \frac{W_{Nr}}{T_{Nr}})^2} - \frac{W_{typ_m} W_{Nm}}{(P_N - T_{N1} - T_{N2} - \ldots - T_{N(m-1)})^2 (Y_m - \frac{W_{Nm}}{P_N - T_{N1} - T_{N2} - \cdots - T_{N(m-1)}})^2} \tag{14}$$

Input : (1) Capacity C_r $\forall 1 \leq r \leq m$, (2) Period P_N, (3) Works w_{Nj} for all tasks of application A_{Nj}.
LSA algorithm :

$for\ \ (r = 1;\ r\ \leq\ m;\ r++)$
  $W_{Nr} = \sum_{\{j\ |\ R_{Nj}\ =\ r\}} w_{Nj}$; /* accumulated work on resource r */
  $W_{typ_r}\ =\ update\_typical\_workload(\ W_{Nr},\ P_N,\ r\ )$;
  $k_{Nr}\ =\ \sqrt{W_{typ_r}/W_{Nr}}$;

$for\ \ (r = 1;\ r\ \leq\ m;\ r++)$
  $L_{Nr}\ =\ W_{Nr}/available\_capacity_r$; /* minimal delays from each resource*/

$if\ (\ \sum_{r=1}^{m} L_{Nr}\ >\ P_N)$ /* check admissibility */
  task cannot be admitted; return;

$denom\ =\ \sum_{r=1}^{m} L_{Nr} * k_{Nr}$;
$S_N\ =\ P_N\ -\ \sum_{r=1}^{m} L_{Nr}$; /* slack */
$for\ \ (r = 1;\ r\ \leq\ m;\ r++)$
  $T_{Nr}\ =\ L_{Nr}\ +\ \frac{L_{Nr} * k_{Nr}}{denom} * S_N$; /* per-resource delay budget*/
  $available\_capacity_r = available\_capacity_r - \frac{W_{Nr}}{T_{Nr}}$;

$for\ \ (j = 1;\ j\ \leq\ u_N;\ j++)$
  $r\ =\ R_{Nj}$; /* id of resource used by A_{Nj} */
  $t_{Nj}\ =\ \frac{w_{Nj}}{W_{Nr}} * T_{Nr}$; /* delay budget of task A_{Nj} */

Figure 2: *Load-based Slack Allocation (LSA) algorithm computes the delay budget allocation for the Nth application A_N with a linear TPG. It apportions the slack S_N, based upon (a) predicted demand on each resource, captured by W_{typ_r}, and (b) current load on each resource, captured by L_{Nr}. The routine update_typical_workload takes application A_N's demand on resource r and updates the typical workload W_{typ_r}.*

Replacing $T_{Nm} = P_N - T_{N1} - T_{N2} - \ldots - T_{N(m-1)}$, we have

$$\frac{\partial n^{-1}}{\partial T_{Nr}} = \frac{W_{typ_s} W_{Nr}}{t_{Nr}^2 (Y_r - \frac{W_{Nr}}{T_{Nr}})^2} - \frac{W_{typ_m} W_{Nm}}{t_{Nm}^2 (Y_m - \frac{W_{Nm}}{T_{Nm}})^2} \tag{15}$$

Equating $\frac{\partial n^{-1}}{\partial T_{Nr}}$ to 0, we have

$$\frac{W_{typ_s} W_{Nr}}{t_{Nr}^2 (Y_r - \frac{W_{Nr}}{T_{Nr}})^2} = \frac{W_{typ_m} W_{Nm}}{t_{Nm}^2 (Y_m - \frac{W_{Nm}}{T_{Nm}})^2} \quad for\ 1 \leq r \leq (m-1) \tag{16}$$

In general, we can replace the subscript m on the RHS of Equation 16 with any s, s.t. $1 \leq s \leq m$. This is due to the fact that Equation 12, from which Equation 16 is finally derived, is perfectly symmetrical w.r.t. each resource s, $1 \leq s \leq m$. Hence we can generalize Equation 16 as follows.

$$\frac{W_{typ_s} W_{Nr}}{t_{Nr}^2 (Y_r - \frac{W_{Nr}}{T_{Nr}})^2} = \frac{W_{typ_s} W_{Ns}}{t_{Ns}^2 (Y_s - \frac{W_{Ns}}{T_{Ns}})^2} \quad for\ 1 \leq r \leq m\ \ and\ for\ 1 \leq s \leq m \tag{17}$$

Upon simplifying, Equation 17 yields

$$\frac{\frac{Y_s T_{Ns}}{W_{Ns}} - 1}{\frac{Y_r T_{Nr}}{W_{Nr}} - 1} = \pm \sqrt{\frac{W_{typ_s}/W_{Ns}}{W_{typ_r}/W_{Nr}}} \quad for\ 1 \leq r \leq m\ \ and\ for\ 1 \leq s \leq m \tag{18}$$

Note that the negative sign in the RHS of Equation 18 is not possible. This is due to the fact that for $1 \leq r \leq m$, $Y_r \geq \frac{W_{Nr}}{T_{Nr}}$, Y_r being the total remaining capacity of resource r, and $\frac{W_{Nr}}{T_{Nr}}$ being the capacity of resource r used by application A_N. Hence both the numerator and denominator of LHS of Equation 18 are positive. We define constant k_{Nr} as follows.

$$k_{Nr} = \sqrt{\frac{W_{typ_r}}{W_{Nr}}} \quad for\ 1 \leq r \leq m \tag{19}$$

Ignoring the negative sign on RHS, Equation 18 can be rewritten as follows.

$$\frac{\frac{Y_s T_{Ns}}{W_{Ns}} - 1}{\frac{Y_r T_{Nr}}{W_{Nr}} - 1} = \frac{k_{Ns}}{k_{Nr}} \quad for\ 1 \leq r \leq m \ \ and\ for\ 1 \leq s \leq m \tag{20}$$

Solving for T_{Ns}, we get the following.

$$T_{Ns} = \frac{W_{Ns}}{Y_s}(1 + \frac{k_{Ns}}{k_{Nr}}(\frac{Y_r T_{Nr}}{W_{Nr}} - 1)) \quad for\ 1 \leq r \leq m \ \ and\ for\ 1 \leq s \leq m \tag{21}$$

Given the above equations and Equation 2, we can solve for T_{Nr}, for $1 \leq r \leq m$. Let us solve for T_{N1} and later generalize the solution for T_{Nr}. Expressing each T_{Nr} in Equation 2 in terms of T_{N1} we get the following.

$$T_{N1} + \sum_{r=2}^{m}(\frac{W_{Nr}}{Y_r}(1 + \frac{k_{Nr}}{k_{N1}}(\frac{Y_1 T_{N1}}{W_{N1}} - 1))) = P_N \tag{22}$$

Solving for T_{N1}, we get the following.

$$T_{N1} = \frac{P_N - \sum_{r=2}^{m} \frac{W_{Nr}}{Y_r}(1 - \frac{k_{Nr}}{k_{N1}})}{1 + \frac{Y_1}{W_{N1}} \sum_{r=2}^{m} \frac{W_{Nr}}{Y_r} \frac{k_{Nr}}{k_{N1}}} \tag{23}$$

The above equation can be simplified as follows when we introduce additional terms in the two summations on numerator and denominator, based on the fact that $k_{N1}/k_{N1} = 1$.

$$T_{N1} = \frac{W_{N1}}{R_1}(\frac{P_N - \sum_{r=1}^{m} \frac{W_{Nr}}{Y_r}(1 - \frac{k_{Nr}}{k_{N1}})}{\sum_{r=1}^{m} \frac{W_{Nr}}{Y_r} \frac{k_{Nr}}{k_{N1}}}) \tag{24}$$

After simplifying the above equation further and generalizing the result to all values of r, the values of T_{Nr}, $1 \leq r \leq m$, that minimize n^{-1} (and hence maximize n) are given by the following equation.

$$T_{Nr} = \frac{W_{Nr}}{Y_r} + \frac{k_{Nr} \frac{W_{Nr}}{Y_r}}{\sum_{s=1}^{m} k_{Ns} \frac{W_{Ns}}{Y_s}}(P_N - \sum_{s=1}^{m} \frac{W_{Ns}}{Y_s}) \tag{25}$$

where $k_{Ns} = \sqrt{\frac{W_{typ_s}}{W_{Ns}}}$, and P_N is the period of application A_N. Note from Equation 6 that $\frac{W_{Ns}}{Y_s}$ represents the minimal delay budget L_{Ns} from resource s. Also note from Equations 5 and 6 that $P_N - \sum_{s=1}^{m} \frac{W_{Ns}}{Y_s}$ represents the slack in delay budget S_N. Replacing these values in Equation 25, we get the following expression.

$$T_{Nr} = L_{Nr} + \frac{k_{Nr} L_{Nr}}{\sum_{s=1}^{m} k_{Ns} L_{Ns}} S_N \tag{26}$$

Equation 26 essentially states that *the total delay budget assignment, T_{Nr}, to tasks in application A_N that use resource r, is the minimal delay budget allocation, L_{Nr}, plus a fraction of the slack, S_N, that is proportional to a weighted percentage of L_{Nr}*. The delay budget assignment t_{Ni} for each individual task A_{Ni} of application A_N, that uses resource r, can be calculated as

$$t_{Ni} = (w_{Ni}/W_{Nr}) * T_{Nr}, \quad where\ R_{Ni} = r. \tag{27}$$

The complete LSA algorithm is presented in Figure 2. Given a constant number of resources, m, the complexity of the algorithm is linear in the number of tasks in the TPG of the application. As would be expected, the effectiveness of the LSA scheme presented above depends on accurate characterization of the typical real-time application in the system. If the incoming applications' workloads closely follow that of typical real-time application, then LSA algorithm provides significant gains. On the other hand, if the deviation from typical real-time application's workload is large, LSA algorithm may not perform as well. We study this aspect further with experiments in Section 4.3.

2.4 Non-linear Task Precedence Graphs

The LSA algorithm for linear TPGs assumes simple summing of Equation 2 in its derivation of result in Equation 26. This simple summing cannot be directly applied to more general non-linear TPGs. Instead, more sophisticated way of combining delay budgets of individual tasks, than simple summing, is needed to arrive at the end-to-end delay of a complete TPG. In this section, we describe the task deadline assignment problem in the case of non-linear TPGs and present an iterative solution.

2.4.1 Accumulating Delay Budgets

First we describe how to accumulate delay budgets of non-linear TPGs. Fundamentally, the accumulation operation should ensure that for every leaf in the TPG, the sum of delay budgets on the longest-delay path from the root to the leaf is smaller than the application's period. Operationally, the following three rules are used to accumulate a TPG's total delay when traversing the TPG from its root in a breadth-first order:

- When visiting a task, add the task's delay budget to the accumulated delay sum.
- Propagate the accumulated delay sum to all the current task's children tasks.
- If a task has multiple parent tasks, it inherits the longest-delay sum among those propagated from its parents.

This end-to-end delay calculation algorithm takes into account the topology of an application's TPG, and is thus a generalization of the simple sum used in linear TPGs. In the rest of this section, we will use $\mathcal{T}_\mathcal{N}$ to represent the accumulated sum of delay budgets, t_{Nj}s, of application A_N. Similarly, we use $\mathcal{L}_\mathcal{N}$ to represent the accumulated sum of *minimal* delay budgets, l_{Nj}s, of application A_N.

2.4.2 Task Deadline Assignment Problem

Assume there are $N-1$ applications already in the system and application A_N, with period P_N, arrives for admission into the system. The application A_N has a non-linear TPG, i.e., tasks are arranged as a directed acyclic graph. Each task A_{Nj} of application A_N requires work w_{Nj} from resource R_{Nj}. We need to find a set of delay budget assignments t_{Nj}s such that

$$\mathcal{T}_N \leq P_N \tag{28}$$

and the load imbalance among different resources is reduced. The reason behind trying to reduce the load imbalance is that the number of typical applications admitted into the system would be maximized if the load imbalance between different resources is kept to a minimum.

2.4.3 Admission Control

The amount of resource R_{ij} consumed by task A_{ij} of application A_i is w_{ij}/t_{ij}. Before application A_N arrives, the available capacity remaining Y_r in any resource r is

$$Y_r \;=\; C_r - \sum_{i=1}^{N-1} \sum_{\{j|R_{ij}=r\}} (\frac{w_{ij}}{t_{ij}}). \tag{29}$$

The minimal delay budget, l_{Nj}, could be assigned from resource r to task A_{Nj}, if the remaining capacity of the resource r is dedicated to completing the work w_{Nj}. That is,

$$l_{Nj} \;=\; \frac{w_{Nj}}{Y_r} \quad where\ R_{Nj} = r \tag{30}$$

To determine whether the available resources are sufficient to accommodate the application A_N's resource requirements, we need to check if the accumulated sum of minimal delay budgets of the TPG lies within the period of the TPG, i.e.,

$$\mathcal{L}_N \quad \leq \quad P_N \tag{31}$$

Recall that $\mathcal{L}_\mathcal{N}$ in the above equation is the accumulated sum of minimal delay budgets l_{Nj}s of the non-linear TPG. If the above condition holds, the system has enough resources to complete the task graph of A_N within its specified period. Otherwise it does not have sufficient resources and A_N should be rejected.

/* $\beta_r * Y$ is the remaining capacity on each resource r after assigning deadlines to all tasks in A_N */

$Y = Y_{initial}$;
while (1) {
    for ($j = 1$; $j \leq u_N$; $j++$)
        $r = R_{Nj}$; /* id of resource used by A_{Nj} */
        $\alpha_{Nj} = \frac{1}{1 - \frac{Y * \beta_r * l_{Nj}}{w_{Nj}}}$;
        $t_{Nj} = \alpha_{Nj} * l_{Nj}$;

    $remains = P_N - accumulate\_delays(A_N)$;
    if ($remains \geq 0$ && $remains \leq Threshold$)
        exit;
    if ($remains > 0$ && $remains > Threshold$)
        $Y = Y + Y_\delta$;
    else
        $Y = Y - Y_\delta$;
}

Figure 3: *The deadline assignment algorithm for non-linear TPGs computes the delay budget for each task t_{Nj}, by iteratively searching for a Y value such that the remaining capacity of each resource r is $Y * \beta_r$, and the task graph could be completed within the application period, P_N.*

2.4.4 Relaxation of Deadlines

In the case that P_N is larger than the accumulated minimal delay budget $\mathcal{L}_\mathcal{N}$, this means that there is spare room in the delay budget assigned to A_{Nj}, and one can exploit this latitude to reduce each task's actual resource demand. Before A_N's arrival, the available capacity of the resource required by A_{Nj}, can be expressed as $\frac{w_{Nj}}{l_{Nj}}$, since l_{Nj} is the minimal delay budget allocated to A_{Nj} if all the remaining capacity of resource R_{Nj} is assigned to completing the task A_{Nj}. Assume the actual delay budget assigned to A_{Nj}, $t_{Nj} = \alpha_{Nj} * l_{Nj}$, where α_{Nj} is the *relaxation parameter*. Then the reduction in the resource requirement on resource R_{Nj} by increasing the delay budget of A_{Nj} from l_{Nj} to t_{Nj} is

$$Y_{Nj} = \frac{w_{Nj}}{l_{Nj}} - \frac{w_{Nj}}{t_{Nj}} = (1 - \frac{1}{\alpha_{Nj}}) * \frac{w_{Nj}}{l_{Nj}} \tag{32}$$

Y_{Nj} is also resource R_{Nj}'s remaining capacity after A_N is admitted. Assume Y_{Nj} is of the form $\beta_{R_{Nj}} * Y$. If the optimization criterion for task deadline assignment is to minimize the difference in the loads on distinct resources, $\beta_{R_{Nj}}$ should be the same value (say 1) for all j. If the optimization objective is to ensure that a resource's remaining capacity is proportional to its current load, $\beta_{R_{Nj}}$ should be set to $CL_{R_{Nj}}$, which is A_N's accumulated load on resource R_{Nj}. In this case, the more loaded a resource is, the proportionally higher relaxation it should be given while assigning deadlines to T_N.

The detailed algorithm to calculate α_{Nj}'s and thus t_{Nj}'s that achieve target Y_{Nj}'s is shown in Figure 3. This algorithm iteratively searches for such a Y value that the remaining capacity in resource R_{Nj}, after A_N is admitted, is $Y * \beta_{R_{Nj}}$, and the TPG could be completed within the specified application period, P_N. During the search, the Y value is increased (decreased) by a Y_δ amount when the accumulated value of delay budgets, t_{Nj}'s, is smaller (larger) than P_N. The function *accumulate_delays(A_N)* calculates the accumulated value of delay budgets, t_{Nj}'s, according to the method described in Section 2.4.1. The algorithm stops when the accumulated delay budget of A_N is smaller than P_N, and the difference is below a given threshold.

As a performance optimization, whenever multiple paths in a TPG share the same starting and end nodes, delay budgets of the tasks on the shorter-delay paths can be relaxed so that the accumulated delay on these shorter-delay paths are lengthened to be the same as that of the longest-delay path. Such relaxation does not affect the end-to-end application delay, but could reduce the resource requirements of the tasks on the shorter-delay paths for free.

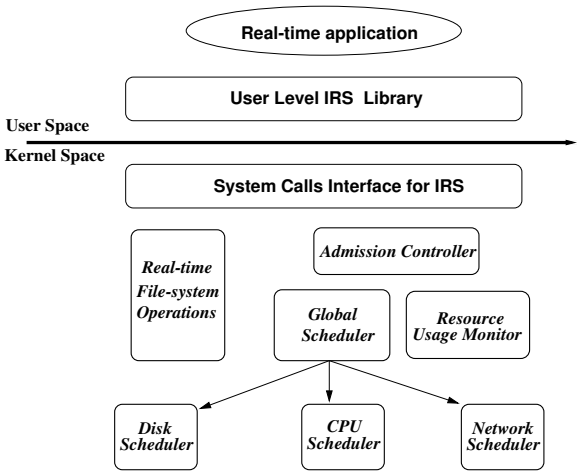

Figure 4: *The software architecture of the* IRS. *Real-time applications are linked to a user-level* IRS *library that uses* IRS *specific system calls to communicate resource requirements to the OS. An admission controller makes admission decisions and allocates resources to applications. A global scheduler dispatches an application's tasks to individual resource schedulers. A resource usage monitor tracks applications' resource usage.*

3 Design and Implementation of IRS

A prototype of *IRS* has been implemented as part of LINUX operating system. Figure 4 shows the software architecture of *IRS*. Real-time applications are programmed using an *IRS* API and linked with a user level *IRS* library that uses a set of *IRS* specific system calls to interact with the kernel subsystem. An *admission controller* makes admission decisions and performs task deadline assignments at the time a new application registers itself as a real-time process. A usage monitor constantly keeps track of the resource consumption of individual tasks to detect misbehaved applications.

IRS is based on a two-level resource scheduling architecture, which consists of a *global scheduler* that has a complete knowledge of each application's task graph and resource requirements, and a set of *local schedulers*, each corresponding to a particular resource. The *global scheduler* is responsible for making sure that a task's dependencies are all satisfied before it is placed in the request queue of the corresponding resource. *Local schedulers* manage individual resources and perform scheduling decisions based on both task deadlines and resource utilization efficiency. The global scheduler acts as a glue between local resource schedulers using its global knowledge of task deadlines and resource requirements for a given application. To the global scheduler, individual resource schedulers are black-box building blocks whose specific scheduling algorithm is completely hidden. Current *IRS* prototype implements real-time CPU and disk schedulers.

3.1 Application Programming Interface

The *IRS* API allows multimedia applications programmers to specify individual resource requirements and timing constraints. An *IRS* application creates a TPG by first invoking the `Create_graph` function.

```
Graph_id = Create_graph(Graph_func, Period);
```

This call starts a new thread, which is responsible for initialization and execution of the new TPG, having the specified period, inside the `Graph_func()` function. The `Graph_func` function is written by application programmers. It first registers with the *IRS* module in the kernel every task that is to be executed each period. Registration of a task includes making reservation for that task's required resource and invoking resource-specific admission control checks to ensure that the resource required by each task is indeed available. A task can be either a *read* operation, *write* operation or *computation* operation. *Read* operations are registered with the *IRS* via `Register_read` (`Register_write` operations have an identical format).

```
T_id1 = Register_read(File_descriptor, Bytes_per_period, Buffer_pointer, Graph_id);
```

`Bytes_per_period` refers to the number of bytes that are read/written into the user buffer within each period. `Graph_id` refers to the id of the graph to which this task belongs and `T_id1` identifies the task

```
while(not all tasks in TPG have finished) {
        for each eligible task e in graph
            e− > Start_exec();

        sleep till some task completes execution;

        for each task e that has just completed
            e− >Finish_exec();
}
```

Figure 5: *The* `Exec_graph` *routine schedules each task in a TPG according to dependency constraints using* kernel select *mechanism.*

subsequently. Computation tasks are also modeled as concurrent threads. A computation task is registered via `Register_compute`.

```
T_id2 = Register_compute( Compute_func, Graph_id);
```

This function starts off a new thread that initially goes to sleep, waiting for *IRS* to wake it up when it becomes eligible. On being woken up, the thread executes `Compute_func()` and then goes back to sleep waiting for the next wake up call from *IRS*. `T_id2` is used to identify the compute task in further operations.

The registration of a task does not execute the task immediately. Rather, it informs the kernel of *how* to execute this task *when* the kernel is asked to do so later. This separation of *how* to execute a task from *when* to execute it is a departure from conventional operating system designs, and provides more flexibility in application programming and resource scheduling. After task registration, the `Graph_func` routine specifies dependencies among tasks with the `Depend` function.

```
Depend( T_id2, T_id1, Graph_id);
```

This function tells *IRS* that within each period the execution of task `T_id2` can start only upon the completion of task `T_id1`. Multiple `Depend` calls can be used to specify precedence among tasks in a pairwise manner. The complete precedence graph constructed as a result of calls to `Depend` must be *directed* and *acyclic*. The `Depend` function performs necessary checks to ensure this property. The main body of the `Graph_func` routine is a while loop with the `Exec_graph` function in the loop body, which begins periodic execution of the TPG just specified .

```
while(some_condition_is_true)
    Exec_graph(Graph_id);
```

On each call to `Exec_graph`, *IRS* strives to complete the execution of all tasks in the task graph according to their precedence constraints within the period. The control returns to `Exec_graph` once every period after completing the execution of all tasks in the TPG.

3.2 Global Scheduler

Each task of a real-time application is represented in *IRS* as a *task* data structure in the kernel. The *task* data structure includes task dependency information, such as its parents and children tasks in the task graph. It also includes task specific routines used to start and finish executing the task, to make resource specific admission decisions, to set up parameters for task execution at the beginning of every period, and to check if the task has completed execution.

The global scheduler is a kernel function that is invoked once every period by a call to `Exec_graph` system call. The global scheduler executes the TPG in the context of the calling application using the algorithm shown in Figure 5. A task is scheduled for execution only when it becomes *eligible*, i.e. when all its parent tasks in the TPG have finished execution. Since the global scheduler has complete knowledge of an application's entire TPG, it could dispatch all *eligible* tasks simultaneously, thus improving the system throughput. However, for *read* and *write* tasks for disk and network subsystems, *blocking is inevitable*. In LINUX, an application can only block on a single event inside the kernel, because a system call performs at most one I/O operation and thus could prevent the global scheduler from proceeding to other eligible

tasks. This model is not optimal for *IRS*, because the global scheduler has the necessary information to issue multiple I/O operations simultaneously, and therefore should be allowed to exploit this concurrency.

The global scheduler processes an eligible *read* or *write* task in a task graph by calling the corresponding task's `Start_exec()` routine, which inserts the task in the corresponding resource's request queue, and returns the control to the global scheduler. Instead of blocking immediately, the global scheduler moves on to processing other eligible tasks in a similar way, and puts itself to sleep only when *all* immediately eligible tasks in the task graph have been dispatched. Essentially, *IRS* allows a task graph thread to be blocked on multiple I/O events in the kernel. This is a *kernel select* mechanism that is similar in nature to the `select` call at the user level. When a *read* or *write* task completes, the associated global scheduler is woken up, which completes the task just completed by calling its `Finish_exec` routine. If more tasks become eligible as a result of the task just completed, the global scheduler again dispatches them and then goes back to sleep.

3.3 CPU Scheduler

The CPU scheduler in the current *IRS* prototype uses a simple Earliest Deadline First scheduling policy for real-time tasks and a priority-based scheduling policy for non real-time tasks. This scheduler is in no way tied to the *IRS* model and could easily be replaced by more sophisticated deadline based schedulers. The global scheduler dispatches a computation task together with its deadline to the CPU scheduler after the task's parents are all completed. In addition to deadlines, tasks can also have jitter requirements, which impose additional constraints on the scheduling eligibility of a task. The CPU scheduler picks the real-time task with the earliest deadline that has entered its jitter window. If the CPU scheduler does not find any eligible real-time task, it picks a non real-time task using the default scheduling policy of LINUX.

To prevent non real-time processes from hogging the CPU, a simple check is made in the timer interrupt service routine as to whether the current CPU task is non real-time and another real-time eligible CPU task is waiting in the scheduler queue. If so, the CPU scheduler is invoked so that the non real-time task is preempted in favor of the most eligible real-time task. This improves the maximum latency seen by a real-time task from default 100 ms in LINUX to 10 ms. Even better latency bounds with microsecond resolution can be achieved by reprogramming the timer chip for every scheduling decision [26]. However, this entails additional overhead for each timer chip reprogramming operation.

3.4 Disk I/O Subsystem

Traditional disk subsystems use the SCAN algorithm to schedule disk I/O requests. SCAN algorithm sorts requests according to their track positions on the disk and services them in the sorted order to reduce unnecessary seeks. SCAN is designed to maximize the disk throughput by minimizing seek time and does not take into account any deadline constraints on the I/O requests. The simplest algorithm for deadline based scheduling is EDF [14]. However it does not take into account the relative positions of requested data on the disk. Hybrids of SCAN and EDF algorithms for deadline based I/O are described in [1, 6, 7, 23]. All of them use EDF schedule as the basic scheme and reorder requests so as to reduce seek and rotational latency overhead.

To achieve a performance level as close to the SCAN algorithm as possible while meeting all disk requests' deadlines, *IRS* uses a *Deadline Sensitive SCAN Algorithm* (*DS-SCAN*). *DS-SCAN* is similar in spirit, but simpler in formulation, to Just-in-Time scheduler [16] used in RT-Mach [24]. Due to space considerations, here we present a brief description of the *DS-SCAN* algorithm. Further details of *DS-SCAN* can be found in our technical report [9].

DS-SCAN places each disk I/O request in two queues - one queue ordered by scan positions, and another queue ordered by *start deadlines.* The scan-ordered queue corresponds to a queue based on desirability of I/O request in terms of the seek distance between the target position of a disk request and the current disk head position. The *start deadlines* based queue orders requests by the latest time the requests can be dispatched to the disk and still complete before their deadline. The *DS-SCAN* scheduler services the next I/O request in the scan-ordered queue if this would not cause the request with the earliest start-deadline to miss its deadline. Otherwise, the scheduler services the disk request with the earliest start-deadline and then re-arranges the scan-order queue accordingly.

Component	Overhead (microsecs)
Admission Control	50 to 70
Global Scheduler	73 to 90
CPU Scheduler	0.1 to 1
Disk Scheduler	5 to 11.5 (Insertion) 3 to 7 (Scheduling)

Table 1: *Component overheads in* IRS *implementation. The overheads are relatively small considering that typical soft real-time applications have period lengths in the order of milliseconds.*

3.5 Admission Control and Usage Monitoring

When an application registers as a real-time process, the admission controller checks the condition given in Equation 3 to determine whether each individual resource has sufficient capacity to admit the application. *IRS* places applications with computation tasks, whose CPU bandwidth requirements are unknown, in a *probation* state at first, dynamically measures their CPU bandwidth requirements for a period of time, applies admission control based on the measurements, and performs deadline assignment. For admitted applications, *IRS* constantly monitors their resource usage to ensure that their resource consumption is in line with their original reservations. When an application consistently misses its deadlines or when the available capacity of a resource falls below a threshold, the usage monitor identifies misbehaving applications from which *IRS* reclaims resources, and deregisters them from real-time category.

4 Performance

In this section, we first present some micro-benchmarks to show the overheads of the current prototype implementation of *IRS*. Next we study the effectiveness of resource allocation and deadline assignment algorithm presented in Section 2.3.4 and finally compare the performance of real-time tasks with and without the *IRS* framework. All measurements in this section were performed on a machine with Intel Pentium III 650 MHz processor and a Seagate IDE hard disk with an average measured data transfer rate of 30 Mbps.

4.1 Micro-benchmarks

The principal sources of runtime overhead in *IRS* are due to admission control, the global scheduler, and local CPU and disk schedulers. These overhead times depend on the number of tasks in TPG of each of the applications and the number of applications in the system. We varied both number of tasks per TPG and the number of applications from 1 to 10. Each *IRS* application used in micro-benchmarking was a sample application consisting of equal number of read and computation tasks. For instance, an application with 6 tasks consisted of 3 read tasks and 3 computation tasks interleaved with each other. The overhead times of various *IRS* components are given in Table 1.

Admission control is invoked once at the start of each real-time process to perform slack allocation. The time taken for execution of this step varied between 50 to 70 microseconds. The global scheduler executes once per period per real-time application and its execution overhead varied between 73 to 90 microseconds per period. The CPU scheduler always took less than 1 microsecond, even with 10 real-time applications, to select the next process to execute.[2] The disk scheduler's overhead depends on the number of I/O requests in its queues and, in the course of measurements, this number ranged from 0 to 26 requests. The time taken by the operation of inserting each request in the queue varied between 5 and 11.5 microseconds. The time taken to reach a scheduling decision for the next I/O request to dispatch varied between 3 to 7 microseconds. Considering that typical soft real-time applications have periods in the order of milliseconds, the table shows that the overheads due to *IRS* implementation are relatively insignificant.

[2] This time excludes the standard LINUX overheads for performing context switch between processes, which can take around 10 microseconds.

Workload Type	Data Rate (Mbps)	Computation per period (ms)
Full Color 1 (FC1)	1.5	0.5
Full Color 2 (FC2)	1.0	0.5
Full Color 3 (FC3)	0.5	0.5
Smoothing (SMT)	1.5	5.0
Low Pass (LOP)	1.5	7.0
Mono Color (MON)	1.5	8.5

Table 2: *Different workloads generated by MPEG filtering application. The application has a period of* 50 *ms and places the above specified resource requirements on CPU and disk for each workload type.*

Workload Type	Number admitted by ESA	Number admitted by LSA	Percentage improvement
Mix from Table 2	17	23	29.4 %
25 of FC1-10%	11	16	45.5 %
25 of FC1-20%	11	17	54.5 %

Table 3: *Comparison of number of real-time applications admitted by Equal Slack Allocation (ESA) versus Load-based Slack allocation (LSA) for three different mix of workloads. First row of the table has mix of 25 workloads from MPEG filtering application listed in Table 2. The second and third rows have 25 workloads each obtained by taking FC1 workload as the base and applying a random deviation up to* 10% *and* 20% *respectively on the two resources. It can be seen that in each workload set, LSA admits a significantly more number of real-time applications compared to ESA.*

4.2 Workload Description

In order to measure the effectiveness of *IRS* in allocating resources among applications and meeting real-time guarantees, we took an MPEG video filtering tool available from Lancaster University [17] and rewrote it to work using *IRS* API. The tool features an MPEG file server, a filter server, a network based MPEG Player and a filter control process. The file server reads an MPEG stream from disk and ships it at user-specified rate to a remote client. The filter process can intercept this stream and perform filtering operations such as frame dropping, low-pass filtering, color reduction, etc., before shipping the filtered stream to a remote mpeg player.

For our experiments, we needed the file server and the filter server to be part of the same process so that we could experiment with different data rates from disk and filtering computation loads. Hence we first modified the application such that the file serving and the video filtering operations could be performed from the same process. The modified application operates with a period of 50 ms, and in each period reads an appropriate amount of data (depending on user-specified data rate) and performs various filtering operations. Next we changed this modified application to use *IRS* API to specify QoS guarantees for disk read and computation operations. The workloads obtained from this application with different data rates and filtering operations are listed in Table 2.

In order to obtain a better variation of workloads than what the MPEG filtering application provided, we wrote a *sample IRS* application with a period of 50 ms and with a TPG consisting one disk read and one computation operation. The kind of workload obtained could be controlled by varying the data rate requested from disk and by varying the length of a `for` loop performing dummy computation. In the following text each set of workloads titled 'FC1-X%' were obtained by taking the FC1 workload listed in Table 2 and applying random deviations between 0 and X percent to the resource demands on CPU and disk.

4.3 Effectiveness of Load-based Slack Allocation

In order to study the effectiveness of our Load-based Slack Allocation (LSA) algorithm given in Section 2.3.4, we compared the number of real-time applications admitted by LSA against the Equal Slack Allocation (ESA) scheme (also described in Section 2.3.4). Recall that the ESA scheme divides the slack in delay budget equally among all the tasks in the TPG of a real-time application, whereas LSA scheme divides the slack based on current resource loads and predicted future demands.

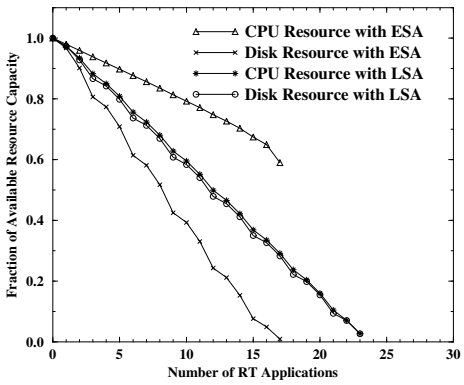

Figure 6: *Variation in available capacity of CPU and disk resources with number of processes in ESA and LSA algorithm for a mix of 25 workloads from Table 2. LSA maintains a balance between the loads on CPU and disk resources whereas the loads are highly divergent in when ESA is used.*

The first experiment is meant to demonstrate that LSA scheme indeed admits more real-time applications than a ESA scheme. Table 3 lists the result of this experiment. Each row in the table corresponds to a set of 25 applications that were started one after another. Row 1 consists of a mix of 25 applications from Table 2 (8 of FC1, 7 of FC2, 7 of FC3, and one each of SMT, LOP and MON). Row 2 consists of 25 FC1-10% applications and Row 3 consists of 25 FC1-20% applications. As can be seen from the table, with each type of workload, LSA admits a significantly more number of real-time applications than ESA. The reason that LSA can admit more applications is that it gives higher proportion of slack to the the task which uses a resource in higher demand. In contrast, ESA completely ignores the demands being placed on the resources while apportioning the slack in delay budget. As a result, LSA scheme achieves a better balance of resources compared to ESA scheme.

The second experiment contrasts the resource usage pattern of LSA scheme against the ESA algorithm. The graphs in Figure 6 plot the variation in percentage available capacity of CPU and disk resources when ESA and LSA algorithms are applied while admitting a mix of 25 MPEG filtering applications listed in Table 2. It can be observed from the graphs that at any point in time, the graphs for percentage available CPU and disk resources in LSA closely track each other. This shows that LSA attempts to maintain a close balance of load on both resources at all times. On the other hand the graphs for CPU and disk resources in ESA are widely imbalanced. Specifically, disk resource is consumed more quickly than CPU resource, leading to fewer number applications that can be admitted overall.

The third experiment presents the effect of difference between the resource demand patterns of past admitted applications and those of the applications that arrive later on. For this experiment, we generated FC1-X% workloads by applying a random deviation between 0 and X percent to the FC1 workload from Table 2. The value of X was varied from 0 percent to 1000 percent. For each value of X, we generated 25 FC1-X% workloads, applied the workloads to the *sample IRS* application described in Section 4.2, and executed the application under both ESA and LSA schemes. Figure 7 plots the number of admitted applications with ESA and LSA algorithms as the percentage deviation from the FC1 workload is steadily increased. As can be seen from the graph, when the percentage deviation is below 100%, LSA admits a significantly more number of applications than ESA. As the deviation becomes larger than 100%, the number of applications admitted by LSA steadily falls to roughly the same values as ESA. This result demonstrates the importance for LSA to accurately estimate the resource demand patterns of typical applications in the future.

4.4 Performance of Real-Time Applications with IRS

This section compares the performance of real-time applications with and without the *IRS* framework. The goal of these measurements was to compare the deadline characteristics of MPEG filter application mentioned in Section 4.2 (a) with *IRS*, i.e., with task deadline assignment, global scheduling, and real-time CPU and disk scheduling and (b) without *IRS*, i.e., with just real-time CPU and disk scheduling. The

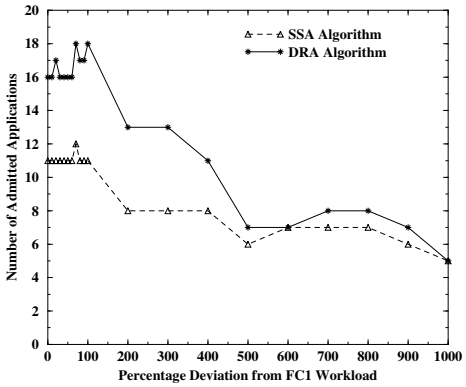

Figure 7: *Variation in number of admitted applications with increasing deviation from FC1 workload in ESA and LSA algorithms. For less than* 100% *deviation, LSA admits significantly higher number of applications than ESA. The difference becomes smaller as the maximum percentage deviation from FC1 workload increases.*

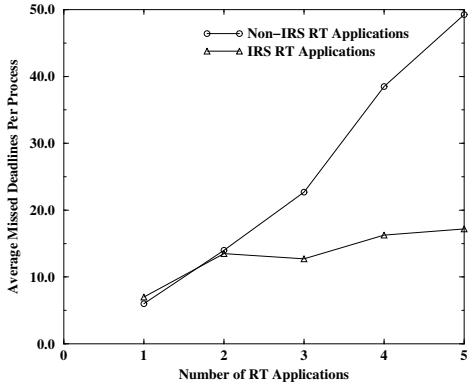

Figure 8: *Average number of missed deadlines per process for non-*IRS *and* IRS *real-time applications, having SMT workload, with increasing number of processes. For non-*IRS *applications, number of missed deadlines increase with increasing number of applications. For* IRS *applications, the number of missed deadlines remain small and occur mainly during probation mode.*

MPEG filter application had the SMT workload requirement given in Table 2 and a periodicity of 50 ms. During each period, the application would read 9375 bytes of data from an MPEG file on disk, and then perform computations for smoothing operation for duration of roughly 5 ms.

In the version of the application using *IRS*, this was implemented as a TPG of a disk read task followed by a smoothing computation task. The kernel performed deadline assignment, global scheduling of tasks in TPG and local real-time scheduling at CPU and disk resources using EDF and DSSCAN respectively.

In the non-*IRS* version of application, no TPG was constructed. Instead, the application sequentially performed the disk read followed by smoothing computation and slept for the remaining time in the period. The deadlines for disk read and smoothing computation tasks were considered to be end of the corresponding periods and these deadlines were communicated to the kernel by the application using a special system call. Real-time scheduling for CPU and disk were based on EDF.

For both *IRS* and non-*IRS* versions, we ran five instances of the application simultaneously, each application reading a different disk file to avoid buffer caching effects. The system was also stressed at the same time by running a compilation process and a "`while(1);`" loop in the background to provide non real-time CPU and disk I/O load.

Figure 8 shows the comparison of the average number of missed deadlines per process for *IRS* and non-*IRS* applications with an increasing number of instances of the application. The average number of missed deadlines for the *IRS* application is small and the deadline misses mainly during probation mode. On the other hand, the number of missed deadlines steadily increases for non-*IRS* applications.

5 Related Work

A vast amount of research work exists in the area of providing operating systems support for real-time applications. Research has been conducted in the area of multi-resource allocation and scheduling from different perspectives. In this section we survey some works relevant to this paper. To the best of our knowledge, our work is the first one to address the issue of multi-resource allocation in real-time system with the goal of maximizing the number of applications admitted into the system.

Q-RAM model [12, 22], considers the problem of allocating multiple resources along single or multiple QoS dimensions such that the overall system utility is maximized. The Q-RAM model attempts to maximize a general utility function whereas *IRS* focuses on real-time applications and attempts to maximize the number of applications admitted. Also, the Q-RAM model tries to allocate maximum system resources to maximize instantaneous system utility, at the expense of re-computing the resource allocation of all admitted application when a new application arrives. In contrast, IRS leaves as much capacity as possible in each resource, and only needs to compute the resource allocation for the new application without touching admitted ones' allocations. Further, Q-RAM model does not consider precedence constraints among tasks of an application which is important in the context of real-time systems.

Continuous Media Resource (CM-resource) model [2] is a theoretical framework that provides end-to-end performance guarantees to applications using *continuous media* (such as digital audio and video). Clients make resource reservation for worst-case workload. The meta-scheduler coordinates with the CPU scheduler, network and file-system and negotiates end-to-end delay guarantees and buffer requirements on behalf of clients. Like the *IRS* model, CM-resource model addresses resource allocation and delay guarantees for applications that use multiple resources. However, unlike the *IRS* model, which handles multiple resources usage related by arbitrary *directed acyclic graph* of precedence constraints, the framework of CM-resource model only handles compound sessions consisting of linear chain of sessions. The definition of cost functions for each resource and exact algorithm to solve minimal cost delay assignment problem is unspecified.

Xu et. al. [28] present a framework and simulation results for a QoS and contention aware, multi-resource reservation algorithm. This work addresses the problem of how to determine the best end-to-end QoS level for an application under the constraint of current resource availability. This goal is different from that of *IRS* which tries to maximize the number of admitted applications in the system.

The work on Cooperative Scheduling Server (CSS) [25] also recognizes the need to perform coordinated allocation and co-scheduling of multiple resources. CSS is a dedicated server that manages one specific controlled resource, such as disk, while using CPU as the controlling resource. It performs admission control for disk I/O requests by reserving both raw disk bandwidth as well as the CPU bandwidth required for processing the disk requests. Similar to *IRS* timing constraints are partitioned into multiple stages each of which is guaranteed to complete before its deadline on a particular resource. However unlike *IRS* the timing

is partitioned based on a fixed slack sharing scheme rather than resource utilization efficiency. As we have demonstrated in this paper, a fixed slack sharing scheme can lead to wide load imbalance between different resources, leading to fewer number of applications admitted. Further, the effect of precedence constraints with other tasks in the real-time application is not addressed.

Real-Time Mach [24] from CMU is a microkernel based OS which provides real-time mechanisms such as real-time thread management, integrated time-driven scheduler and real-time synchronization. It supports an abstraction called *processor capacity reserves* [15] to specify CPU timing constraints of applications. Another work from CMU has proposed the Resource Kernel [19, 21] approach. The resource kernel allows applications to specify only their resource demands leaving the kernel to satisfy those demands using hidden resource management schemes. It uses the Q-RAM model mentioned earlier for QoS management and has been integrated with Linux and Windows NT.

Eclipse/BSD [5] is an operating system derived from FreeBSD that provides flexible and fine-grained QoS support for applications. Eclipse/BSD implements hierarchical, proportional-share CPU, disk and link schedulers [20, 4]. Unlike *IRS*, each resource is allocated and scheduled essentially independently of others.

Spring Kernel [27] provides real-time support for multiprocessor and distributed environments using dynamic planning based scheduling. In Spring, a computation is automatically broken into precedence related tasks (which are the schedulable entities), and scheduled according to precedence constraints. The worst case execution time is automatically derived from source-code analysis based on the target machine. Spring aims for absolute predictability in hard real-time environments and uses specialized hardware to achieve its goals. Our approach in *IRS* is geared towards providing QoS for soft real-time applications with efficient resource utilization.

The Rialto [11] operating system supports firm real-time applications by means of CPU reservations of the form "reserve X units of time out of every Y units". A scheduling graph, determines which tasks execute in a given graph cycle. Applications negotiate the desired QoS parameters with a resource planner object which tries to maximize the user perceived utility of system resource usage. Rialto does not addresses reservations across multiple resources, nor does it provide coordinated real-time scheduling across multiple resources.

Nemesis [13] is a vertically structured operating system that supports real-time applications requiring consistent QoS from all system resources. Nemesis employs a split-level scheme for scheduling CPU resource. The scheduling among application domains is done at a low-level by the kernel and among threads of a domain by the domain itself at user-level. Nemesis device drivers, are implemented as privileged domains. Unlike *IRS*, Nemesis does not perform coordinated allocation and scheduling among domains of multiple resources.

Many other research efforts have focussed on operating system support for real-time applications. We discuss a representative few works below which make significant contributions. However none of these address the issue of real-time scheduling of resources other than CPU.

Scheduler for real-time and multimedia applications (SMART) [18] supports execution of multimedia applications in conjunction with non real-time applications. SMART associates two attributes with executing tasks, namely, urgency, which reflects time-constraints, and importance, which reflects priority. Applications with higher importance are favored and among applications of the same importance, proportional sharing is enforced. The ETI Resource Distributor [10] is a CPU resource allocation and scheduling mechanism for multimedia applications on MAP1000 processor. It uses EDF scheduling to guarantee the delivery of CPU resource to admitted tasks even under system overload while ensuring liveness of applications that are not real-time. The ETI Resource distributor decouples QoS-decision making and scheduling process. Real-Time LINUX (RT-Linux) [3] provides hard real-time support by inserting a real-time kernel layer between the LINUX kernel and the hardware interrupts. LINUX kernel is just another real-time task having the lowest priority. KURT Linux [26] adds firm real-time capabilities to the standard Linux by running the hardware timer as an aperiodic device, thus supporting increased temporal resolution for applications. It implements an explicit planning based CPU scheduler in which applications explicitly specify the event deadlines.

6 Contributions

In this paper, we presented the design and implementation of an Integrated Real-time Resource Scheduler (*IRS*). *IRS* provides a framework for integrated allocation and scheduling of multiple resources among dynamic periodic real-time applications. Specifically, this work makes the following contributions :

- A new task deadline assignment algorithm for tasks in periodic real-time applications that maximizes the number of real-time applications admitted in the system by reducing load imbalance among multiple resources.
- A two-level real-time resource scheduler architecture that respects task dependencies when dispatching tasks, performs coordination of local resource schedulers, and supports the ability to dispatch multiple concurrent tasks to local schedulers before blocking.
- A programming API, using which the application programmer can specify resource and timing requirements of the application in a declarative fashion without having to worry about explicit deadline management for individual tasks.

7 Future Work

Currently we are planning to apply the *IRS* framework to provide QoS guarantees for web server clusters, where the clients can request not only throughput guarantees (requests/sec) but also per-request response time guarantees. Servicing each web request typically involves access to multiple resources such as CPU, disk and network link. *IRS* is an essential building block for providing bounded response time guarantees in such a system where it is important to maximize the number of clients admitted by the web server cluster. Further research could also be conducted along the following lines. Currently we have two solutions for the task deadline assignment problem - a direct solution for linear TPGs and an iterative heuristic for non-linear TPGs. It would be interesting to investigate whether a simpler direct solution exists for non-linear TPGs. It is also worth investigating whether it is worthwhile to re-compute deadline assignments for all the tasks in the system every time an application leaves or joins the system. The current *IRS* prototype never looks back on its previous decisions, and we may need to reconsider this design decision in order to adapt better to widely fluctuating workloads.

References

[1] R. K. Abbot and H. Gracia-Molina, "Scheduling I/O Requests with Deadlines: A Performance Evaluation", In Proceedings of RTSS, 113-124, December 1990.

[2] D. P. Anderson, "Meta-Scheduling For Distributed Continuous Media", Technical Report CSD-90-599, Computer Science Division, EECS Department, University of California at Berkeley, 1990.

[3] M. Barabanov and V. Yodaiken, "Real-Time Linux", Linux Journal, February 1997.

[4] J. C. R. Bennett and H. Zhang, "WF^2Q: Worst-case Fair Weighted Fair Queuing", In Proceedings of IEEE INFOCOM, San Francisco, pp 120-128, March 1996.

[5] J. Blanquer, J. Bruno, E. Gabber, M. Mcshea, B. Ozden, and A. Silberschatz, "Resource Management for QoS in Eclipse/BSD", Proceedings of the FreeBSD 1999 Conference, Berkeley, California, October 1999.

[6] M. J. Carey, R. Jauhari, and M. Linvy, "Priority in DBMS Resource Scheduling", In Proceedings of the 15th VLDB Conference, 1989.

[7] S. Chen, J. A. Stankovic, J. F. Kurose, and D. Towsley, "Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems", Journal of Real-Time Systems, Vol. 3, 307-336, 1991.

[8] K. Gopalan, and T. Chiueh,"Integrated Real-Time Resource Scheduling", Technical Report TR-56, Experimental Computer Systems Labs, Department of Computer Science, SUNY at Stony Brook, June 1999.

[9] K. Gopalan, and T. Chiueh,"Real-time Disk Scheduling Using Deadline Sensitive SCAN", Technical Report TR-92, Experimental Computer Systems Labs, Department of Computer Science, SUNY at Stony Brook, Jan 2001.

[10] M. Baker-Harvey, "ETI Resource Distributor: Guaranteed Resource Allocation and Scheduling in Multimedia Systems", In Operating Systems Design and Implementation, February, 1999.

[11] M. B. Jones, D. Rosu, and M. Rosu, "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities", In Proc. of the 16th ACM Symposium on Operating System Principles, St-Malo, France, pp. 198-211, Oct. 1997.

[12] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen, "A Scalable Solution to the Multi-resource QoS Problem", Proceedings of IEEE RTSS'99, December 1999.

[13] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barharn, D. Evers, R. Fairbairns, and E. Hyden, "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications", IEEE Journal on Selected Areas In Communications, Vol. 14, No. 7, pp. 1280-1297, September 1996.

[14] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming environment in a hard real-time environment," Journal of the ACM, 20(1), 47-61, (1973).

[15] C. W. Mercer, S. Savage, and H. Tokuda, "Processor Capacity Reserves: Operating System Support for Multimedia Applications", In Proceedings of the IEEE International Conference on Multimedia Computing and Systems May 1994

[16] A. Molano, K. Juvva, R. Rajkumar, "Real-time filesystems. Guaranteeing timing constraints for disk accesses in RT-Mach", Proceedings of the 18th IEEE Real-Time Systems Symposium, December 1997, San Fransico, CA.

[17] N. Yeadon, F. Garcia, D. Hutchison, and D. Shepherd , "Continuous Media Filters for Heterogeneous Internetworking", Proceedings of SPIE - Multimedia Computing and Networking (MMCN'96), San Jose, CA, January 1996.

[18] J. Neih and M. S. Lam, "The design, implementation and evaluation of SMART: A scheduler for multimedia applications", In Proc. ACM Symposium on Operating Systems Principles, St.Malo, France, Oct. 1997.

[19] S. Oikawa and R. Rajkumar, "Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior", In Proceedings of the IEEE Real-Time Technology and Applications Symposium, Vancouver, June 1999.

[20] A. Parekh, "A generalized processor sharing approach to flow control in integrated services networks." Ph.D dissertation, Massachusetts Institute of Technology, February 1992.

[21] R. Rajkumar, K. Juvva, A. Molano and S. Oikawa, "Resource Kernels: A Resource-Centric Approach to Real-Time Systems", In Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking January 1998.

[22] R. Rajkumar, C. Lee, J. Lehoczky, D. Siewiorek, "Practical Solutions for QoS-based Resource Allocation Problems", Proceedings of IEEE RTSS'98, December 1998.

[23] A. L. N. Reddy and J. Wyllie, "Disk Scheduling in Multimedia I/O System". In Proceedings of ACM Multimedia'93, Anaheim, CA, 225-234, August 1993.

[24] H. Tokuda, T. Nakajima and P. Rao, "Real-Time Mach: Towards a Predictable Real-Time System", Proceedings of USENIX Mach Workshop October 1990

[25] S. Saewong and R. Rajkumar, "Cooperative Scheduling of Multiple Resources", In Proceedings of the IEEE Real-Time Systems Symposium, December 1999.

[26] B. Srinivasan, S. Pather, R. Hill, F. Ansari, D. Niehaus, "A Firm Real-Time System Implementation Using Commercial Off-The-Shelf Hardware and Free Software", In Proc. of Real Time Technology and Applications Symposium, June 1998.

[27] J. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Systems", IEEE Software, Vol. 8, No. 3, pp. 62-72, May 1991.

[28] D. Xu, K. Nahrstedt, A. Viswanathan, D. Wichadakul, "QoS and Contention-Aware Multi-Resource Reservation", In Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC-9), August 2000.