Multi-Resource Allocation and Scheduling for Periodic Soft Real-Time Applications

Kartik Gopalan Tzi-cker Chiueh
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794
{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 multiple heterogeneous resources in the system. 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 for periodic soft real-time application. The principal feature of *IRS* is a heuristic multi-resource allocation algorithm that reserves multiple resources for real-time applications in a manner that can maximize the number of applications admitted into the system in the long run. More specifically, the algorithm takes an application's timing constraints into account and assigns a delay budget to each task in the application according to the current load and predicted demand on individual resources. At run-time, a *global scheduler* dispatches the tasks of the soft real-time application to individual resource schedulers according to the precedence constraints between tasks. The individual resource schedulers, which could be any deadline based schedulers, can make scheduling decisions locally and yet collectively satisfy a real-time application's performance requirements. The tightness of overall timing guarantees given to applications is ultimately determined by the properties of individual resource schedulers. *IRS* provides a framework to coordinate deadline assignment across multiple tasks in a soft real-time application, and exploits global knowledge to maximize overall system resource utilization efficiency.

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 periodically 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. In addition, to achieve reasonable perceptual quality, the entire sequence of tasks needs to be completed within a certain period of time. The above example typifies 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 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. In this paper, we present a framework called Integrated Real-time Resource Scheduler (*IRS*), that specifically addresses the problem of multi-resource allocation and scheduling. The framework supports application-level soft real-time performance guarantees while maximizing the overall resouce utilization efficiency of the system.

In the *IRS* model, application programmers specify the performance requirement on each resource that the application uses, the precedence ordering among the uses of resources, and its period. We call each use of a distinct system resource within an application a *task*. The precedence ordering among tasks is

specified by means of a *task precedence graph* (TPG). Given these information for each real-time application, *IRS* assigns deadlines to each task in the TPG such that the TPG's delay bound for periodic execution can be met and the overall system resource utilization efficiency is maximized. Since programmers are relieved of 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 a heuristic algorithm for multi-resource allocation and deadline assignment. Given the TPG and its period, the heuristic algorithm automatically computes delay budget (which determines deadline within each period) for each task such that the entire TPG can 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 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.

The deadlines assigned to different tasks in TPG by the deadline assignment algorithm are only as tight as are permitted by individual deadline-based resource schedulers. If the individual resource schedulers can provide only soft real-time guarantees, the *IRS* framework itself can provide only soft guarantees. In the current implementation, we use earliest deadline first (EDF) based schedulers which are known to be non-optimal for the case where deadline of a task is not the same as its period. Hence current *IRS* prototype provides only soft real-time guarantees that are sufficient for digital multimedia applications that can tolerate a few deadline misses.

The goal of this work is *not* to develop another comprehensive real-time operating system. Hence the current *IRS* implementation neither includes support for such features as priority inheritance and high-resolution timers nor does it provide real-time guarantees for multi-threaded or multi-process applications. *Rather, the main focus of 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 support application level performance guarantees.* Hence the current *IRS* implementation includes only basic features required to demonstrate our ideas.

The rest of the paper is organized as follows. In Section 2, we describe a new multi-resource allocation algorithm that *IRS* uses to efficiently allocate system resources among periodic soft 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 results and an outline of planned future research.

2 Resource Allocation and Deadline Assignment

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 task precedence graph (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 4). 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 quality of service (QoS) 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 early enough to leave time for Tasks 2 and 3 to complete before the end of the period. In other words, the 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. Note that task deadline assignment is in fact assignment of *delay budgets* to each task in TPG at admission time. These delay budgets in turn determine the deadlines of a TPG's tasks in each period at run-time.

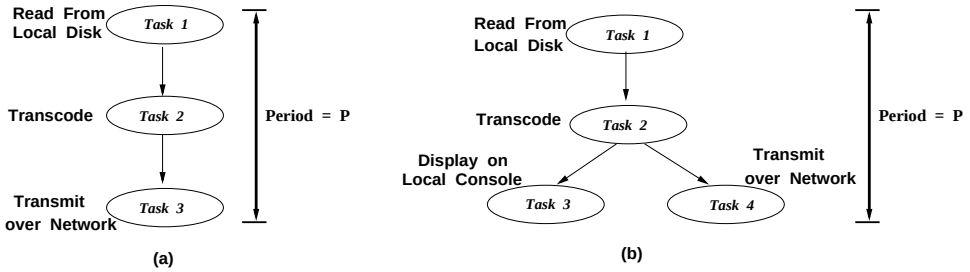

Figure 1: *Task precedence graphs (TPG) for video playback applications. (a) Linear TPG - video frames are read from the local disk, transcoded, and transmitted over the network. (b) General TPG - video frames are read from the local disk, transcoded, and then displayed on the local console as well as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

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 since a task's* ready *time is the same as the deadline of the task it depends on according to the TPG.* Therefore, assigning deadline to a task also entails a specification of the load on that task'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 efficiency by taking into account the current loads and predicted 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). For this case, we propose a *direct* heuristic solution to the task deadline assignment problem in Section 2.5. 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 which balances the loads across different resources. However, this approach is computationally more expensive than the direct solution for the linear TPG. We do not present the iterative algorithm here due to space considerations and its description can be found in the technical report version of this paper [8].

2.1 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.

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

L_{ir} The total minimum 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 $R_{ij} = r$.

2.2 Typical Real-time Application

Now we 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 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

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

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 resource demand pattern of earlier applications.

In current *IRS* implementation, 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 past applications admitted into the system. W_{typ_r} is then calculated as the average of resource demands over a small range of values around the median, the rationale being that a small range 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 2.5 that the values of T_{typ_r} do not need to be calculated. For the rest of this section, we assume that the information about typical workloads $\{W_{typ_1}, W_{typ_2}, \ldots, W_{typ_m}\}$ is readily available.

2.3 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 a 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.

2.4 Admission Control

The latitude in assigning deadline to a task depends on the remaining capacity of the resource that the task requires. The admission control module first decides 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.

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

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 $C_r - \sum_{i=1}^{N-1} \frac{W_{ir}}{T_{ir}}$. With respect to resource r, the minimum delay budget, L_{Nr}, could be assigned to A_N, when the remaining capacity of the resource r is dedicated to completing the work W_{Nr}. To simplify the analysis, we use the following *heuristic* to determine L_{Nr}.

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

The above expression is a heuristic since it it calculates minimum delay L_{Nr} solely on the basis of remaining available capacity at resource r and does not account for any scheduler induced delays. We will discuss this further in Section 2.6. To determine whether application A_N's timing requirements can be possibly met, the following *heuristic* based on minimum delay budgets is used:

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

If the above inequality holds, we assume that the task graph of A_N can be completed within its specified period; otherwise A_N should be rejected. Inequality 4 is a *necessary* but not *sufficient* condition for meeting the timing constraints of the application. The strictness of timing guarantees depends on how strictly the individual resource schedulers can guarantee task deadlines.

2.5 Deadline Assignment - A Direct Heuristic Solution

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 the slack to each task. Let's call this scheme *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 real-time applications*

that can be admitted in the future is maximized. We describe such a heuristic slack allocation scheme below and call it *Load-based Slack Allocation* (LSA). We call it a heuristic since its effectiveness depends on how accurately one can model the *typical real-time application* in a system.

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 minimum 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.,

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

The *optimization goal* of the deadline assignment is to maximize the number of typical 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 an equal number of typical applications will be admitted by each resource individually at the point which maximizes the minimum given by Equation 8. 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

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

In order to minimize n', we need to find the partial derivatives $\frac{\partial n'}{\partial T_{Nr}}$ for $0 < r < m$ and equate each derivative to 0. Here we skip the detailed derivation and present the results. The omitted details can be found in our technical report [8]. Using the m relations resulting from the differentiation step, the values of T_{Nr}, $1 \leq r \leq m$, that minimize n' (and hence maximize n) can be shown to be given by the following equation.

$$T_{Nr} = \frac{W_{Nr}}{Y_r} + \frac{k_{Nr}\frac{W_{Nr}}{Y_r}}{\sum_{i=1}^{m} k_{Ni}\frac{W_{Ni}}{Y_i}}\left(P_N - \sum_{i=1}^{m}\frac{W_{Ni}}{Y_i}\right) \tag{13}$$

where $k_{Ni} = \sqrt{\frac{W_{typ_i}}{W_{Ni}}}$, and P_N is the period of application A_N. From Equation 6, $\frac{W_{Ni}}{Y_i}$ represents the minimum delay budget L_{Ni} from resource i. From Equations 5 and 6, $P_N - \sum_{i=1}^{m}\frac{W_{Ni}}{Y_i}$ represents the slack in delay budget S_N. Replacing these values in Equation 13, we get the following expression.

$$T_{Nr} = L_{Nr} + \frac{k_{Nr}L_{Nr}}{\sum_{i=1}^{m} k_{Ni}L_{Ni}}S_N \tag{14}$$

Equation 14 essentially states that *the total delay budget assignment, T_{Nr}, to tasks in application A_N that use resource r, is the minimum delay budget allocation, L_{Nr}, plus a fraction of the slack, S_N, which 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

$$t_{Ni} = (w_{Ni}/W_{Nr}) * T_{Nr}, \quad where\ R_{Ni} = r. \tag{15}$$

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.

The above LSA algorithm for linear TPGs assumes simple summing of Equation 2 in its derivation of result in Equation 14. Therefore it cannot be directly applied to more general non-linear TPGs. To obtain a solution for non-linear TPGs, a 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. Fundamentally, the task deadline assignment scheme 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,

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 */
  $W_{typ_r}\ =\ update\_typical\_workload(\ W_{Nr},\ P_N,\ r\ );$
  $k_{Nr}\ =\ \sqrt{W_{typ_r}/W_{Nr}};$

$for\ \ (r = 1;\ r\ \leq\ m;\ r++)$
  $L_{Nr}\ =\ W_{Nr}/available\_capacity_r$; /* minimum 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 each task A_{Nj} of A_N
  $r\ =\ R_{Nj}$; /* 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}.*

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. An iterative algorithm to obtain a heuristic solution for non-linear TPGs is presented in our technical report [8].

2.6 Discussion

The LSA algorithm does a reasonably good job of maximizing resource utilization efficiency, as we will show later in Section 4. However, the deadlines assigned by LSA are not optimal because of three main reasons.

First of all, in the derivation of Equation 14, we make two simplifying assumptions, namely, (1) the minimum delay introduced by each resource is $Work/Residual\_Capacity$ (W/RC) and (2) the schedulability criterion is solely based on capacity. Both of these assumptions are not strictly accurate in practice. The individual resource schedulers, based on their specific policies, can introduce additional delays apart from that indicated by the W/RC expression. Further, a set of tasks may not be perfectly schedulable in spite of sufficient available capacity. For these two reasons, *IRS* framework does not provide hard real-time guarantees in its current form. Ideally, to address these issues, the global admission control mechanism needs to first ask each local resource scheduler the minimum delay it can support given its current residual capacity. The slack allocation algorithm then needs to apply relaxation to decrease the resource demands. In each relaxation step, the algorithm again needs to consult the local resource schedulers to determine how much increase in the delay will be caused by the decrease in the resource demand and whether the tasks would still be schedulable.

Secondly, the effectiveness of LSA depends on accurate characterization of *typical real-time application* in the system. If the future applications' workloads closely follow that of typical real-time application, then the 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 provide the desired performance improve-

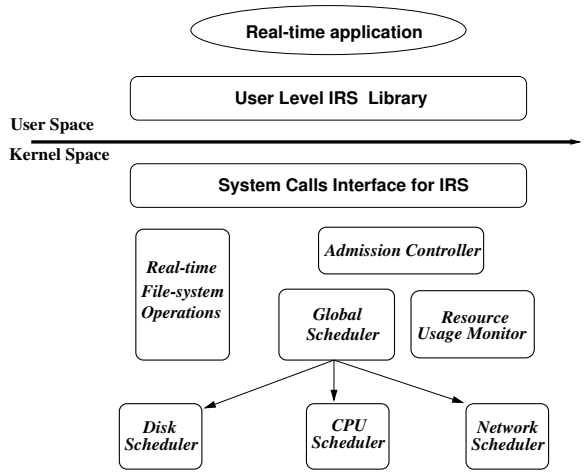

Figure 3: *The software architecture of the* IRS.

ment. We study this aspect further with experiments in Section 4.3.

Finally, LSA considers applications for admission *one at a time* and operates under the constraint that *when a new real-time application arrives, it cannot go back to revise earlier reservation decisions.* Hence LSA has the effect of leaving as much possible resource unreserved for use by future applications. In case we had the flexibility of revising the delay budgets of all applications admitted earlier, we might as well have allocated *all* system resources to current set of applications without leaving anything unreserved. This is because that we would anyway revise the delay budgets of each application once a new application arrives, so why leave any resource unreserved? Therefore, it is probable that a more efficient resource allocation may result if earlier admission decisions were revised with each new incoming application. However we believe that such back-tracking is computationally expensive and hence impractical.

3 Design and Implementation

A prototype of *IRS* was implemented as part of LINUX operating system. Figure 3 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. 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. The current *IRS* prototype implements EDF-like real-time CPU and disk schedulers. EDF is known to be non-optimal for the case when deadlines of tasks are not the same as their period. Since this is the case with deadlines assigned to tasks of a TPG, the current *IRS* prototype provides only soft real-time guarantees.

3.1 Application Programming Interface

The *IRS* API allows multimedia applications programmers to specify individual resource requirements and timing constraints. A feature of the *IRS* API is the support for implicit CPU 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, the current *IRS* implementation allows multimedia application programmers to leave their CPU resource requirements unspecified. Instead they are implicitly specified by the application's code. An application exploiting implicit resource requirement specification is first placed in the *probation* mode and its CPU requirements are dynamically measured for a certain number of periods. Once the application's implicitly specified resource requirements become clearer, the admission controller can determine whether the application should be promoted to *real-time* mode.

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 executes the `Graph_func()` function with specified period. The `Graph_func` function is written by application programmers and is responsible for the initialization and execution of TPG. `Graph_func` function first registers every task of the TPG with the *IRS* module in

the kernel. 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` operation has 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 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` identifies 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` system call.

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

This system call 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.

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

The control returns to `Exec_graph` once every period after completing execution of all tasks in the TPG.

3.2 Global Scheduler

The global scheduler is a kernel function that is invoked once every period by a call to `Exec_graph()` system call from the application. The global scheduler executes the TPG in the context of the calling application using the algorithm shown in Figure 4. 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 pointers to resource specific routines - `Start_exec()` and `Finish_exec()`. These two interface routines enable the global scheduler to communicate with local resource schedulers. The `Start_Exec()` routine submits a task to local resource scheduler and `Finish_Exec` performs resource specific post-processing (if any) when a task completes. For instance, a task `e` that performs disk read would be submitted to disk scheduler's queues by a call to `e->Start_exec()` and upon completion, the data read would be retrieved by a call to `e->Finish_exec()`.

In every iteration, the global scheduler dispatches each currently *eligible* task `e` in the TPG to its corresponding local resource scheduler by a call to `e->Start_Exec()` routine. A task becomes *eligible* when all its parent tasks in the TPG have finished execution. Following this, the global scheduler sleeps waiting for completion of any one of the dispatched tasks. Upon being woken up, the global scheduler invokes post-processing for each recently completed task `e` by a call to `e->Finish_Exec` routine.

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 process 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 wraps up 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 schedulers, such as one which would not allow non-real-time applications to starve. 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 [25]. 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 [15]. 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, 21]. 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 [22]. 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 respective deadlines. The *DS-SCAN* scheduler services the next I/O request in the SCAN ordered queue only 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.

3.5 Admission Control

When an application registers as a real-time process, the admission controller checks the condition given in Equation 4 to determine whether each individual resource has enough capacity to admit the application. *IRS* places applications with computation tasks,

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.*

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 disk and CPU for each workload type.*

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 whom to reclaim resources.

4 Performance

In this section, we first present some micro-benchmarks to show the overheads of the *IRS* prototype implementation. Next we study the effectiveness of resource allocation and deadline assignment algorithm presented in Section 2.5 and finally compare the performance of real-time tasks with and without *IRS*. All measurements in this section were performed on a machine with Intel Pentium III 650 MHz processor and a Seagate IDE hard disk.

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 overheads depend on the number of tasks in the TPG of each application and the number of applications in the system. We varied both the number of tasks per TPG and the number of applications from 1 to 10. Each application used in micro-benchmarking was a sample application consisting of an 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 overheads of various 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. Given 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.

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 [27] 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 a 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 are integrated in a single process. The modified application operates

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

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. In each workload set, LSA admits a significantly more number of real-time applications compared to ESA.*

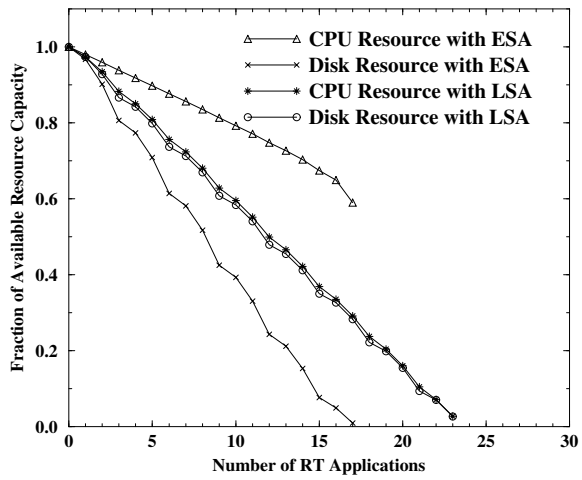

Figure 5: *Variation in available capacity of CPU and disk resources with number of processes for a mix of 25 workloads from Table 2. LSA maintains a balance between the loads on CPU and disk resources whereas the loads are increasingly divergent when ESA is used.*

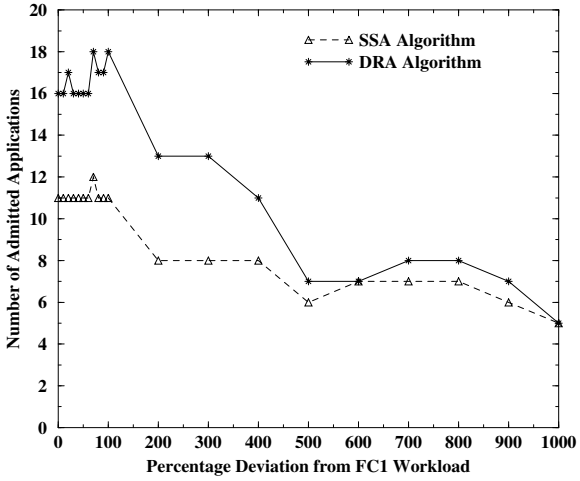

Figure 6: *Variation in number of admitted applications with increasing deviation from FC1 workload in ESA and LSA algorithms. For less than 100% deviation, LSA admits significantly higher number of applications than ESA. The difference becomes smaller as the maximum percentage deviation from FC1 workload increases.*

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 derived from this application with different data rates and filtering operations are listed in Table 2.

To test a wider variety of workloads than the MPEG filtering application, we wrote a *sample IRS* application with a period of 50 ms and with a TPG of one disk read and one computation task. The workload was 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 a workload titled 'FC1-X%' was 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 LSA

To study the effectiveness of Load-based Slack Allocation (LSA) algorithm given in Section 2.5, we compared the number of real-time applications admitted by LSA against the Equal Slack Allocation (ESA) scheme (also described in Section 2.5). Recall that the ESA scheme divides the slack in delay budget equally among all the tasks in a TPG, whereas LSA scheme divides the slack based on current resource loads and predicted future demands.

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 the slack to the the task which uses a resource that is 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 the LSA algorithm against the ESA algorithm. The curves in Figure 5 depict the evolution of available capacity of CPU and disk resources when ESA and LSA algorithms are applied while admitting a sequence of 25 MPEG filtering applications listed in Table 2. At any point in time, the available CPU and disk resources in LSA closely track each other. This shows that LSA does a good job of maintaining a balance of load on both resources at all times. On the other hand the loads on CPU and disk resources in ESA are widely imbalanced. Specifically, disk resource is consumed more quickly than CPU resource, leading to fewer admitted applications.

The third experiment presents the effects of variations in the resource demand patterns of admitted applications and demonstrates the importance for LSA to accurately estimate the typical resource demand pattern of future application. 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 6 plots the number of admitted applications with ESA and LSA algorithms as the percentage deviation from the FC1 workload is steadily increased. 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. The reason for the degradation in LSA's effectiveness with increasing variation in input workload is that LSA relies on the estimated *typical workload* to accurately reflect the nature of workload expected in future. If the deviation of future workload from the predicted typical workload is large, LSA algorithm will not produce the intended performance improvements.

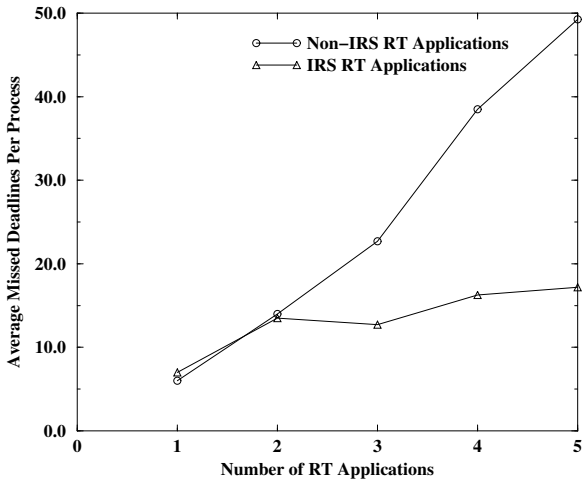

Figure 7: *Average number of missed deadlines per process for non-*IRS *and* IRS *real-time applications. For non-*IRS *applications, number of missed deadlines increase with increasing number of applications. For* IRS *applications, the number of missed deadlines remain small and occur mainly during probation mode.*

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 MPEG filter application had the SMT workload described 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 a 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 DS-SCAN 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 7 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 same application. The average number of missed deadlines for the *IRS* application is small and the deadline misses mainly during the probation mode. A small fraction of deadlines are missed in real-time mode due to the non-optimal nature of EDF-like algorithms when deadline of tasks is not the same as their period. On the other hand, the number of missed deadlines steadily increases for non-*IRS* applications due to the absence of a central deadline splitting mechanism.

5 Related Work

A vast amount of research work has been conducted in the area of resource allocation and scheduling in real-time systems 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 and scheduling in real-time system with the goal of maximizing the number of applications admitted into the system.

Q-RAM model [13, 20], 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 computes 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 minimum 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) [23] also recognizes the need to perform coordinated allocation and co-scheduling of multiple resources. CSS 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.

Spring Kernel [26] provides real-time support for multiprocessor and distributed environments using dynamic planning based scheduling. A computation is automatically broken into precedence related tasks and the worst case execution time is automatically derived from source code analysis. While Spring aims for absolute predictability in hard real-time environments, *IRS* provides QoS for soft real-time applications with efficient resource utilization.

Now we describe some real-time systems that have proposed sophisticated resource abstractions and real-time scheduling frameworks for different sys-

tem resources. In all these systems, each resource is allocated and scheduled independently of others.

Real-time Mach [22] from CMU is a microkernel based OS which provides real-time thread management, integrated time-driven scheduler and real-time synchronization. It supports *processor capacity reserves* abstraction to specify CPU timing constraints of applications. Another work from CMU has proposed the Resource Kernel [18] 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.

QLinux is an operating system derived from Linux kernel that provides QoS guarantees for CPU, disk and network subsystems independently. It features a Hierarchical Start Time Fair Queuing (H-SFQ) CPU [10] and network packet [11] scheduler and Cello disk scheduler [24]. QLinux also performs lazy receiver processing for network subsystem.

Eclipse/BSD [5] is derived from FreeBSD and provides flexible and fine-grained QoS support for applications. It implements hierarchical, proportional share CPU, disk and link schedulers [19]. The Rialto [12] operating system supports firm real-time applications by means of CPU reservations. A scheduling graph, determines which tasks execute in a given graph cycle. A resource planner object tries to maximize the perceived utility of system resource usage.

Nemesis [14] is a vertically structured operating system that supports real-time applications requiring consistent QoS from all resources. It employs a split level scheduler for CPU resource - kernel scheduling among application domains and user-level scheduling among threads of a domain. Nemesis device drivers are implemented as privileged domains.

Some significant research efforts that have focussed on CPU scheduling for real-time applications are discussed below. Scheduler for real-time and multimedia applications (SMART) [17] supports execution of multimedia applications in conjunction with non-real-time applications. SMART associates two attributes with tasks - urgency and importance. Applications with higher importance are favored and among applications of the same importance, proportional sharing is enforced. The ETI Resource Distributor [3] 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. Real-time LINUX (RT-Linux) [4] 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 [25] adds firm real-time capabilities to the standard Linux by running the hardware timer as an aperiodic device.

6 Summary

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 soft real-time applications. Specifically, this work makes the following contributions :

- A new heuristic algorithm for deadline assignment to tasks in periodic soft real-time applications that maximizes the number of real-time applications that can be admitted in the future. The algorithm achieves its goal by apportioning slack in delay budget according to current load and predicted demands on each resource.

- A two-level real-time resource scheduler framework 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. The framework allows the local resource schedulers to be any deadline based schedulers.

- 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.

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. Also, currently we have two heuristic algorithms for the task deadline assignment problem - a direct algorithm for linear TPGs and an iterative algorithm for non-linear TPGs. It would be interesting to investigate whether a simpler direct solution exists for non-linear TPGs.

Acknowledgements - The authors would like to thank Prashant Pradhan, Saurabh Sethia and anonymous referees for providing insightful comments that greatly improved the presentation of this paper.

References

[1] R.K. Abbot and H. Gracia-Molina, "Scheduling I/O Requests with Deadlines: A Performance Evaluation", Proc. of RTSS, 113-124, Dec. 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. Baker-Harvey, "ETI Resource Distributor: Guaranteed Resource Allocation and Scheduling in Multimedia Systems", Operating Systems Design and Implementation, February, 1999.

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

[5] J. Blanquer, et. al., "Resource Management for QoS in Eclipse/BSD", Proc. of the FreeBSD Conference, Oct. 1999.

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

[7] S. Chen, et. al., "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, Computer Science Department, 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, Computer Science Department, SUNY at Stony Brook, Jan 2001.

[10] P. Goyal, X. Guo and H.M. Vin,"A Hierarchical CPU Scheduler for Multimedia Operating Systems", Proc. of 2nd Symposium on Operating System Design and Implementation (OSDI'96), 107-122, October 1996.

[11] P. Goyal, H.M. Vin and H. Cheng,"Start-time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks", Proc. of ACM SIGCOMM'96, 157-168, August 1996.

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

[13] C. Lee, et. al., "A Scalable Solution to the Multi-resource QoS Problem", Proc. of IEEE RTSS'99, Dec. 1999.

[14] I.M. Leslie, et. al., "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications", IEEE Journal on Selected Areas In Communications, Vol. 14, No. 7, 1280-1297, September 1996.

[15] 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.

[16] A. Molano, K. Juvva, R. Rajkumar, "Real-time filesystems. Guaranteeing Timing Constraints for Disk Accesses in RT-Mach", Proc. of 18th IEEE Real-time Systems Symposium, December 1997.

[17] J. Neih and M.S. Lam, "The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications", Proc. of ACM Symposium on Operating Systems Principles, Oct. 1997.

[18] S. Oikawa and R. Rajkumar, "Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior", Proc. of IEEE RTAS'99, June 1999.

[19] A. Parekh, "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks.", Ph.D dissertation, Massachusetts Institute of Technology, February 1992.

[20] R. Rajkumar, et. al., "Practical Solutions for QoS-based Resource Allocation Problems", Proc. of IEEE RTSS'98, December 1998.

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

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

[23] S. Saewong and R. Rajkumar, "Cooperative Scheduling of Multiple Resources", Proc. of IEEE Real-time Systems Symposium, December 1999.

[24] P. Shenoy and H.M. Vin, "Cello: A Disk Scheduling Framework for Next Generation Operating Systems", Proc. of ACM SIGMETRICS Conference, 44-55, June 1998.

[25] B. Srinivasan, et. al., "A Firm Real-time System Implementation Using Commercial Off-The-Shelf Hardware and Free Software", Proc. of RTAS'98, June 1998.

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

[27] N. Yeadon, et. al., "Continuous Media Filters for Heterogeneous Internetworking", Proc. of SPIE - Multimedia Computing and Networking (MMCN'96), January 1996.

[28] D. Xu, et. al., "QoS and Contention-Aware Multi-Resource Reservation", Proc. of 9th IEEE International Symposium on High Performance Distributed Computing, August 2000.