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

Kartik Gopalan* and Tzi-cker Chiueh†

Dept. of Computer Science, State University of New York, Stony Brook, NY, 11794

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. 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 is ultimately determined by the properties of individual resource schedulers. However, *IRS* maximizes overall system resource utilization efficiency by coordinating deadline assignment across multiple tasks in a soft real-time application.

Keywords: Real-time, resource allocation, scheduling, multi-resource

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 three distinct properties shared by many soft real-time applications : (1) uses of multiple heterogeneous resources each have a specific performance requirement, (2) uses of resources within an application are strictly ordered and form a dependency graph, and (3) 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

*kartik@cs.sunysb.edu; phone: 1 631 632-8436; †chiueh@cs.sunysb.edu; phone: 1 631 632-8449; Address: Dept. of Computer Science, State University of New York, Stony Brook, NY, 11794

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 those 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 does not include support for such features as priority inheritance, high-resolution timers, and 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 our implementation includes only basic features required to demonstrate our ideas.

The rest of the paper is organized as follows. Section 2 reviews related work in this area. In Section 3, we describe a new multi-resource allocation algorithm that *IRS* uses to efficiently allocate system resources among periodic soft real-time applications. Section 4 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 5 presents performance results demonstrating the effectiveness of *IRS*. Section 6 provides a summary of main results and an outline of planned future research.

2. RELATED WORK

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 systems with the goal of maximizing the number of applications admitted into the system. Q-RAM model[1, 2] considers the problem of allocating multiple resources along single or multiple QoS dimensions such that a general system utility function is maximized. It tries to allocate maximum system resources to maximize instantaneous system utility, at the expense of re-computing the resource allocation of all admitted applications when a new application arrives. In contrast, *IRS* focuses on real-time dimension and attempts to maximize the number of applications admitted. *IRS* leaves as much capacity as possible unreserved in each resource and computes the resource allocation for a new application without touching admitted ones' allocations. CM-resource model[3] aims to provide end-to-end performance guarantees to applications using *continuous media*. 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. Cooperative Scheduling Server (CSS)[4] 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 of disk I/O are partitioned into multiple stages each of which is guaranteed to complete before its deadline on a particular resource. Unlike *IRS*, the timing is partitioned based on a fixed slack sharing scheme rather than resource utilization efficiency. As we will show later, a fixed slack sharing scheme can lead to poor resource utilization efficiency, leading to fewer number of applications

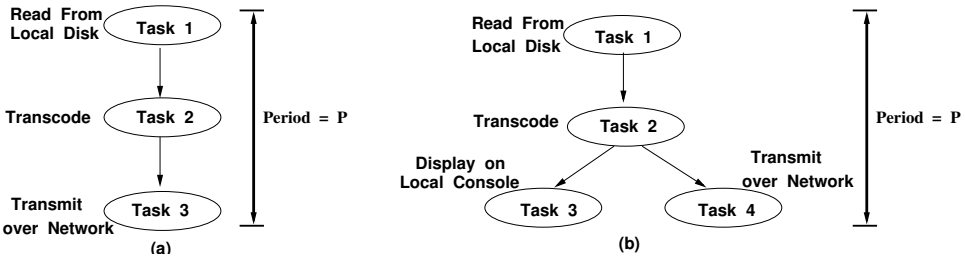

Figure 1: Task precedence graphs (TPG) for video playback applications. (a) Linear TPG (b) General TPG.

admitted. Spring Kernel[5] provides real-time support for multiprocessor and distributed environments using dynamic planning based scheduling. A computation is 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. Xu et. al.[6] present a framework and simulation results for a QoS and contention aware, multi-resource reservation algorithm. They address 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. Many other real-time systems have sophisticated resource abstractions and scheduling frameworks for different system resources. These include Real-Time Mach,[7] QLinux,[8,9] Eclipse/BSD,[10] Rialto,[11] Nemesis,[12] SMART,[13] and RT-Linux.[14] In all these systems, each resource is allocated and scheduled independently.

3. 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. 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.

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

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.
A_{ij}	The j-th task of the i-th application.
P_i	The period 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 possibly be allocated to the task A_{ij}.
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 $R_{ij} = r$.

Table 1: Notations used in Section 3.

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 3.4. 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.[15] The notations used in rest of this section are listed in Table 1.

3.1. 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

$$T_{typ_1} + T_{typ_2} + \cdots + T_{typ_m} = 1 \tag{1}$$

In other words, W_{typ_r} is the typical amount of work required from resource r every second.* 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 the 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. We will show later in Section 3.4 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.

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

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

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

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

3.3. 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. 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 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 3.5. To determine whether application A_N's timing requirements can be possibly met, the following *heuristic* based on minimum delay budgets is used:

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

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.

3.4. 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 algorithm to divide this slack would be to give equal share of the slack to each task. Let's call this algorithm *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 algorithm 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

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

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

$$Y'_r = Y_r - (W_{Nr}/T_{Nr}) \tag{7}$$

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 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 n, the number of typical applications admitted. From Equation 7 and Equation 8, it follows that n depends on the delay budget allocations $T_{N1}, T_{N2}, \ldots, T_{Nm}$. Hence we need to find the combination of the delay budget allocations that maximizes n. Due to space considerations, we skip the detailed derivation and present the results. The omitted details can be found in our technical report.[15] The values of T_{Nr}, $1 \leq r \leq m$, that maximize n can be shown to be given by the following equation.

$$T_{Nr} = L_{Nr} + \frac{k_{Nr} L_{Nr}}{\sum_{i=1}^{m} k_{Ni} L_{Ni}} S_N \tag{9}$$

where $k_{Ni} = \sqrt{\frac{W_{typ_i}}{W_{Ni}}}$. Equation 9 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 $t_{Ni} = (w_{Ni}/W_{Nr}) * T_{Nr}$, where $R_{Ni} = r$. 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 9. Therefore it cannot be directly applied to more general non-linear TPGs. To obtain a solution for non-linear TPGs, the algorithm needs to consider the topology of the TPG to arrive at its end-to-end delay. Fundamentally, the task deadline assignment algorithm 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. (1) When visiting a task, add the task's delay budget to the accumulated delay sum. (2) Propagate the accumulated delay sum to all the current task's children tasks. (3) If a task has multiple parent tasks, it inherits the longest-delay sum among those propagated from its parents. An iterative algorithm to obtain a heuristic solution for non-linear TPGs is presented in our technical report.[15]

3.5. Discussion

The LSA algorithm does a reasonably good job of maximizing resource utilization efficiency, as we will show later in Section 5. However, the deadlines assigned by LSA are not optimal because of three main reasons. First of all, in the derivation of Equation 9, 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 consult each local resource scheduler about the minimum delay it can support for a task given its current residual capacity and the schedulability of a task given its delay budget.

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 improvement. We study this aspect further with experiments in Section 5.3.

Finally, LSA operates under the constraint that *when a new real-time application arrives, it cannot go back to revise earlier reservation decisions*. 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.

4. DESIGN AND IMPLEMENTATION

Figure 3 shows the software architecture of *IRS*, implemented as part of the LINUX operating system. 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. The *IRS* API provides interfaces to create a TPG, to specify dependency constraints among its tasks, and to periodically execute it. An *admission controller* makes admission decisions and performs task deadline assignments as described in Section 3. A usage monitor constantly keeps track of the resource consumption of individual tasks. *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. Since EDF is known to be non-optimal for the case when deadlines of tasks are not the same as their period, the current *IRS* prototype provides only soft real-time guarantees.

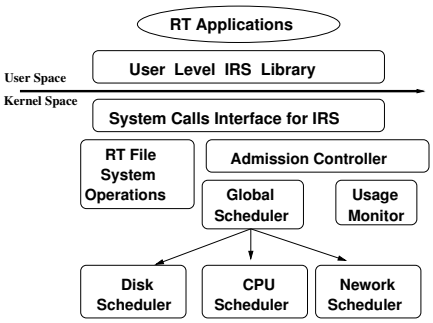

Figure 3: The software architecture of *IRS*.

```
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 global scheduler that dispatches tasks in a TPG according to dependency constraints.

4.1. Global Scheduler

The global scheduler is a kernel function that is invoked once every period by a 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. Essentially, *IRS* allows a task graph thread to be blocked on multiple events in the kernel. This is a *kernel select* mechanism that is similar in nature to the `select` call at the user level. Upon being woken up, the global scheduler invokes post-processing for each recently completed task `e` by a call to `e->Finish_Exec()` routine.

4.2. 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. 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.

4.3. Disk I/O Subsystem

Traditional disk subsystems use the SCAN algorithm to schedule disk I/O requests. 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*).[16] We briefly describe *DS-SCAN* which is similar in spirit, but simpler in formulation, to Just-in-Time scheduler[17] used in RT-Mach.[7] *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

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 with a period of 50 ms.

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.

5. PERFORMANCE

5.1. Micro-benchmarks

In this section, we first study the overheads of the *IRS* implementation. All measurements were performed on a machine with Pentium III 650 MHz processor and a Seagate IDE hard disk. The principal sources of runtime overhead in *IRS* are due to admission control, the global scheduler, and local CPU and disk schedulers. Admission control overhead, which includes slack allocation, varied between 50 to 70 microseconds. The global scheduler overhead, which executes once per period per real-time application, varied between 73 to 90 microseconds per period. Each CPU scheduling decision always took less than 1 microsecond (excluding the standard LINUX context switch overhead of around 10 microseconds), even with 10 real-time applications. The disk scheduler's overhead depends on the number of I/O requests in its queues. In the course of measurements, this number ranged from 0 to 26 requests. The operation of inserting each request in the queue consumed between 5 and 11.5 microseconds. The 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 overheads due to *IRS* implementation are relatively insignificant.

5.2. Workload Description

We took an MPEG video filtering tool[18] from Lancaster University which 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 specified rate to a remote client. The filter process intercepts this stream and performs filtering operations such as frame dropping, low-pass filtering, color reduction, etc., before shipping the filtered stream to a remote MPEG player. In order to experiment with different workloads, we first modified the application such that the file serving and the video filtering operations are integrated in a single process. The modified application periodically (every 50 ms) reads an appropriate amount of data (depending on specified data rate) and performs various filtering operations. Next we changed the application to use *IRS* API to specify QoS guarantees for disk read and computation operations. The workloads derived from this application are listed in Table 2. To test a wider variety of workloads, we wrote a *sample IRS* application with a period of 50 ms and with a TPG of one disk read and one computation task. In the following text workloads titled 'FC1-X%' were obtained from the sample application 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.

5.3. Effectiveness of LSA

We compared the number of real-time applications admitted by LSA against ESA (both described in Section 3.4). Recall that the ESA algorithm divides the slack in delay budget equally among all the tasks in a TPG, whereas LSA algorithm divides the slack based on current resource loads and predicted demands. The first experiment in Table 3 demonstrates that LSA indeed admits more real-time applications than ESA. Each row in the table

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 ESA versus LSA. 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.

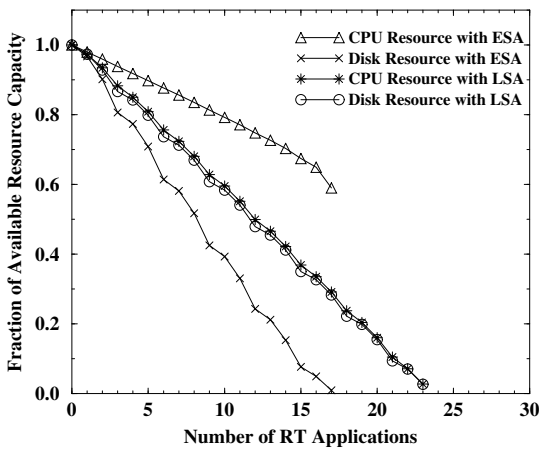

Figure 5. Variation in available capacity of CPU and disk resources with number of processes for a mix of 25 workloads from Table 2.

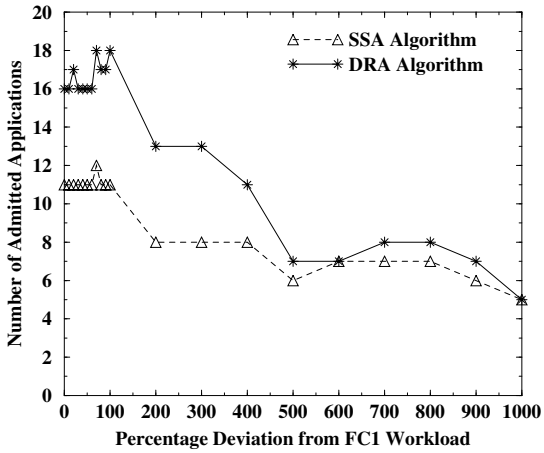

Figure 6. Variation in number of admitted applications with increasing deviation from FC1 workload.

corresponds to a set of 25 applications that were started one after another. With each type of workload, LSA admits a significantly more number of real-time applications than ESA, the reason being that LSA gives higher proportion of the slack to the the task which uses a resource that is in higher demand. In contrast, ESA ignores the demands placed on the resources while apportioning the slack. Figure 5 contrasts the resource usage pattern of LSA against ESA and plots the evolution of available capacity on CPU and disk resources while admitting a sequence of 25 MPEG filtering applications from Table 2. LSA does a good job of maintaining a balance of load on both resources since the available CPU and disk resources closely track each other. 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. Figure 6 demonstrates the importance for LSA to accurately estimate the typical resource demand pattern of future applications. We generated 25 FC1-X% workloads, applied the workloads to the *sample IRS* application described in Section 5.2, and executed the application under both ESA and LSA algorithms. Figure 6 plots the number of admitted applications with ESA and LSA algorithms as the percentage deviation, X, from the FC1 workload is steadily increased. When the 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 value as ESA. LSA's effectiveness degrades with increasing variation in input workload since LSA relies on the predicted *typical workload* to accurately reflect the real future workloads. If the deviation of future workload from the predicted typical workload is large, LSA algorithm will not produce the intended performance improvements.

5.4. Performance of Real-time Applications with IRS

This section compares the deadline characteristics of MPEG filter application mentioned in Section 5.2 (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 (in Table 2) that performed smoothing on MPEG frames. The *IRS* application was implemented as a TPG

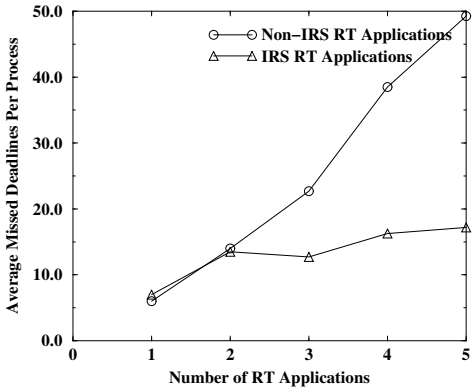

Figure 7. Average number of missed deadlines per process with increasing number of real-time applications. Both cases had non-real-time CPU and disk I/O load in the background.

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. For non-*IRS* 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. Figure 7 shows that the average number of missed deadlines for the *IRS* application is small and the deadlines miss 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.

6. SUMMARY

In this work we proposed the *IRS* framework for integrated allocation and scheduling of multiple resources among dynamic periodic soft real-time applications. Specifically, this work makes the following contributions : (1) A new heuristic algorithm for deadline assignment to tasks that maximizes the number of real-time applications that can be admitted in the future, and (2) 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. Apart from video playback applications, *IRS* framework can also be applied for QoS guarantees in 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. Hence *IRS* framework could be used to maximize the number of clients hosted 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.

ACKNOWLEDGMENTS

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

REFERENCES

1. C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen, "A scalable solution to the multi-resource QoS problem," in *Proc. of 20th IEEE Real-Time Systems Symposium, Phoenix, AZ, USA*, pp. 315–326, Dec. 1999.

2. R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, "Practical solutions for QoS-based resource allocation problems," in *Proc. of 19th IEEE Real-Time Systems Symposium, Madrid, Spain*, pp. 296–306, Dec. 1998.
3. D. Anderson, *Meta-Scheduling for Distributed Continuous Media*, Technical Report CSD-90-599, Computer Science Division, EECS Department, University of California at Berkeley, Berkeley, CA, USA, Oct. 1990.
4. S. Saewong and R. Rajkumar, "Cooperative scheduling of multiple resources," in *Proc. of 20th IEEE Real-Time Systems Symposium, Phoenix, AZ, USA*, pp. 90–101, 1999.
5. J. Stankovic and K. Ramamritham, "The Spring Kernel: A new paradigm for real-time systems," *IEEE Software* **8(3)**, pp. 62–72, May 1991.
6. D. Xu, K. Nahrstedt, A. Viswanathan, and D. Wichadakul, "QoS and contention-aware multi-resource reservation," in *Proc. of 9th IEEE International Symposium on High Performance Distributed Computing, Piitsburgh, PA, USA*, pp. 318–327, Aug. 2000.
7. H. Tokuda, T. Nakajima, and P. Rao, "Real-time Mach: Towards a predictable real-time system," in *Proc. of 1st USENIX Mach Workshop, Burlington, VT, USA*, pp. 73–82, Oct. 1990.
8. P. Goyal, X. Guo, and H. Vin, "A hierarchical CPU scheduler for multimedia operating systems," in *Proc. of 2nd Symposium on Operating System Design and Implementation, Seattle, WA, USA*, pp. 107–121, Oct. 1996.
9. P. Shenoy and H. Vin, "Cello: A disk scheduling framework for next generation operating systems," in *Proc. of ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems, Madison, WI, USA*, pp. 44–55, June 1998.
10. J. Blanquer, J. Bruno, E. Gabber, M. Mcshea, B. Ozden, and A. Silberschatz, "Resource management for QoS in Eclipse/BSD," in *Proc. of FreeBSD'99 Conf., Berkeley, CA, USA*, Oct. 1999.
11. M. Jones, D. Rosu, and M. Rosu, "CPU reservations and time constraints: Efficient, predictable scheduling of independent activities," in *Proc. of 16th ACM Symposium on Operating System Principles, St. Malo, France*, pp. 198–211, Oct. 1997.
12. I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, 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* **14(7)**, pp. 1280–1297, Sept. 1996.
13. J. Neih and M. Lam, "The design, implementation and evaluation of SMART: A scheduler for multimedia applications," in *Proc. of 16th ACM Symposium on Operating Systems Principles, St. Malo, France*, pp. 184–197, Oct. 1997.
14. M. Barabanov, *A Linux-based Real-Time Operating System*, Master Thesis, Dept. of Computer Science, New Mexico Institute of Mining and Technology, Socorro, NM, USA, June 1997.
15. K. Gopalan and T. Chiueh, *Integrated Real-time Resource Scheduling*, Technical Report TR-56, Experimental Computer Systems Labs, Dept. of Computer Science, SUNY at Stony Brook, Stony Brook, NY, USA, Jan 2001.
16. K. Gopalan and T. Chiueh, *Real-time Disk Scheduling Using Deadline Sensitive SCAN*, Technical Report TR-92, Experimental Computer Systems Labs, Dept. of Computer Science, SUNY at Stony Brook, Stony Brook, NY, USA, Jan 2001.
17. A. Molano, K. Juvva, and R. Rajkumar, "Real-time filesystems: Guaranteeing timing constraints for disk accesses in RT-Mach," in *Proc. of 18th IEEE Real-time Systems Symposium, San Francisco, CA, USA*, pp. 36–46, Dec. 1997.
18. N. Yeadon, F. Garcia, D. Shepherd, and D. Hutchinson, "Continuous media filters for heterogeneous internetworking," in *Proc. of SPIE Conf. on Multimedia Computing and Networking, San Jose, CA, USA*, March 1996.