Integrated Scheduling of Multimedia and Hard Real-Time Tasks*

Hiroyuki Kaneko, John A. Stankovic,
Subhabrata Sen and Krithi Ramamritham
Computer Science Department
University of Massachusetts
LGRC, Box 34610
Amherst MA 01003-4610

UMass Computer Science Technical Report 96-45
August 1996

Abstract

An integrated platform which is capable of meeting the requirements of both traditional real-time control processing and multimedia processing has enormous potential for accommodating various kinds of new applications. However, except for the simplest of situations, few, if any, research or commercial systems successfully provide architectural and OS mechanisms which can efficiently support both hard real-time computation and multimedia soft real-time computation. In this paper, we propose a multimedia server executing on multiprocessor real-time operating systems to provide different classes of guarantee to support both types of processing. The multimedia server supports multiple periodic multimedia streams with a capability for graceful QoS degradation during system overload. In this paper we (i) discuss realistic system implementation issues on the SGI IRIX/REACT/PRO operating system, (ii) develop several multimedia server scheduling algorithms, and (iii) present a performance evaluation. We chose the SGI system as an implementation platform because it is being used more and more for multimedia applications. Our performance evaluation demonstrates that a multimedia server algorithm based on a flexible, proportional *allocation scheme provides the best performance and that simple iterative scheduling is adequate for handling graceful degradation of the multimedia streams. We consider issues such as server size and period as well as the impact of context switch overhead on the performance. We also show that for applications which require* integrated *resource sharing, neither the frame scheduler nor the deadline scheduler supplied in the IRIX/REACT/PRO OS are suitable. We propose an implementation solution that is appropriate.*

*This work was supported by the National Science Foundation Grant No. IRI-9208920 and CDA-9502639, and Mitsubishi Electric Corporation.

1. Introduction

Many hard real-time applications such as automated manufacturing and attack helicopters are being designed to take advantage of audio and video information. This information has real-time requirements such as delay and jitter tolerance, requires suitable real-time operating system support, and is less critical than hard real-time control information. However, support for processing this information must co-exist with the hard real-time control information. For example, in attack helicopters such as the Comanche, control tasks have to be executed within their deadlines otherwise the helicopter will not fly. Audio and video sensors can provide monitoring and sophisticated control of the helicopter. To do this requires flexible and dynamic scheduling that includes various types of interaction between the hard and soft real-time control tasks.

Accommodating multimedia and traditional real-time applications which have interaction requirements is a challenging research issue. However, little attention has been paid to the coexistence of these applications. For example, the Mercuri system [?] is one of the few research projects targeting this objective, where data from remote video cameras are transferred through an ATM network and displayed in X windows, but they fail to provide any guarantees and end up with providing best effort services.

This paper presents a mechanism to support the coexistence of multimedia applications and traditional hard

real-time applications that interact via shared use of CPUs, using the SGI Challenge multiprocessor and its IRIX/REACT/PRO OS. It develops various real-time scheduling algorithms to provide the necessary scheduling support, and presents a performance evaluation that demonstrates the value of the solutions. We chose the SGI system as an implementation platform because it is being used more and more for multimedia applications.

The rest of this paper is organized as follows. Section 2 introduces several applications which can benefit from direct integration of hard real-time control and multimedia. Section 3 presents the integrated scheduling algorithms and a solution that can be used on the IRIX/REACT/PRO OS. In the simplest of applications, a solution can be based on complete partitioning of the two classes of work. In this case, the frame scheduler or deadline scheduler of the IRIX/REACT/PRO OS can be used. For more complicated applications, integrated solutions are necessary, and we show that the standard IRIX/REACT/PRO schedulers cannot be used. The QoS degradation solution is discussed in Section 4. In Section 5, simulation results are presented. These show that a multimedia server based on a flexible, proportional allocation scheme is highly effective and that a simple iterative policy is adequate for handling QoS degradation in overload. Section 6 summarizes the work.

2. Applications

With technology like high performance CPUs, memory, disks and high speed networks becoming less expensive and more easily available, a number of multimedia applications have emerged both in the commercial world and in research. At the same time, traditional real-time computing is still one of the major applications being used in various fields. In order to motivate the need for a platform which is capable of supporting these two types of computations at the same time, consider the following applications.

First, even the coexistence of a simple video stream display and real-time control processes requires new solutions. For example, suppose that in a power plant or industrial manufacturing plant, plant operators (i) monitor situations in different locations of the plant via cameras and (ii) control actuators based on this monitoring. Currently these analog video monitoring systems and digitized controlling systems are implemented on completely separate platforms. Replacing these redundant systems with an integrated digitized system can reduce the cost since the reduction in the number of cables, display equipment, etc. is significant. In addition to the reduction in cost, the integrated digitized system can provide more functionality. For example, processing of the audio and video by on-line algorithms may then directly control various actuators for more effective and faster response to problems. Also several video streams can be shown on a single screen, information for them can be fused, and automatic control of actions might be triggered, allowing faster and more accurate response. Many companies, including Honeywell and Mitsubishi, are pursuing applications with similar characteristics.

Second, many examples can be found in military applications such as controlling the fly-by-wire Comanche helicopter through trees and telephone wires and "looking for" enemy soldiers or vehicles based on processing video and audio data. Upon detection of various situations from the video and audio processing, direct control of the helicopter may occur. The workload presented by this application is highly dynamic and subject to both hard and multimedia real-time constraints.

Third, *computer-participative* multimedia applications are another emerging trend in multimedia research [?]. As opposed to *computer-mediated* multimedia applications such as online encyclopedias and video-conferencing systems, in which the computer acts as a mediator between the application author and user or between two users, computer participative multimedia applications perform analysis on their audio and video data input, and take actions based on the analysis. For example, a system in which a program watches television news shows and maintains an online database of stories organized by subject is introduced in [?]. In this type of application, input data have to be manipulated or filtered by software rather than hardware because of the flexibility required in the design. Similar applications can be seen in [?] and [?]. It is possible to make use of these techniques for traditional real-time systems. For example, we may want to know if there is any intruder in an isolated area by filtering the data from the remote monitoring camera with a motion detection filter. The detection can be directly connected to the alarm system or controlling functions such as shutting the valves or closing the gates.

We assume that single processor systems will not be used for these applications since multimedia processing is sometimes very computation-intensive (e.g., the Comanche helicopter uses a multiprocessor as the main processing engine). In some of the ongoing multiprocessor-based research approaches, some processors are dedicated to multimedia processing and others to traditional real-time processing, e.g., [?]. Although this approach can provide good isolation of one type of processing from another, it has several disadvantages:

- It cannot achieve high utilization of system resources in a dynamic environment.

  It is not effective to dedicate three processors for multimedia processing when there is only one multimedia

session and the rest of the processors are overloaded with real-time processing. Allowing both types of tasks to exist in the same processor makes the system more adaptable. This is the main type of interaction among hard real-time and multimedia tasks that is explicitly addressed in this paper.

- Correctness may be jeopardized when the various types of processing interact over shared data resources.

  If the two classes of work interact over shared resources, treating them independently may cause tasks to miss their deadlines due to possible blocking over these shared resources. The solutions presented in this paper solve the blocking problem among hard real-time tasks themselves, but assume that there is no read-write shared resources between the hard real-time and multimedia tasks.

Our approach is therefore, to accommodate both multimedia and traditional real-time processes in a multiprocessor system and allow both types of processes to reside in any processor. Our solution is described in the context of the SGI Challenge multiprocessor and its OS.

3. Multimedia Server

3.1. Background

The multimedia server is a periodic task that is dynamically created and scheduled along with hard real-time tasks. We use a planning-based scheduler, as exemplified by the Spring scheduling algorithm [?], to perform this level of scheduling. The server then executes the multimedia tasks themselves. A planning-based scheduler dynamically generates schedules or plans in which every task included in the schedule is guaranteed its required resources (including a processor) for its worst case execution time. When a new set of tasks arrives at the system, it attempts to assign execution windows for the new tasks and every task in its current schedule such that every task completes by its deadline and there are no resource conflicts between any tasks scheduled to execute at the same time. If a feasible schedule cannot be found, the new set of tasks is rejected and the previous schedule remains intact. This planning allows admission control and results in a reservation-based system. The key aspect of this scheduler is its ability to schedule not only the CPU but also the other required resources in an integrated fashion.

On the basis of this planning-based scheduling algorithm, we integrate multimedia and hard real-time processes using a multimedia server. The server is given a fraction of CPU time and is responsible for controlling the execution of multimedia tasks. Task executions of multiple multimedia streams are multiplexed into one multimedia server instance. Hard real-time tasks are executed in the rest of the CPU time. Of course, it is possible that we regard each multimedia task instance as a hard real-time task and schedule it without having the multimedia server. However, the cost involved in individually scheduling these task instances using the planning-based scheduler would be too high. Its capability to provide more precise guarantees per task instance is essential for the hard real-time control tasks, but this level of determinism is not needed for multimedia sessions which can do with more statistical types of guarantees.

3.2. Multimedia Task Allocation Policies

In this paper, we investigate both static and flexible allocation schemes as well as proportional and individual allocation schemes. This gives rise to four different combinations.

- Static proportional allocation
- Static individual allocation
- Flexible proportional allocation
- Flexible individual allocation

This section discusses these different schemes and Section ?? describes how these various combinations are integrated with the planning-based scheduler.

Static versus Flexible Allocation. Obviously, there can be many different policies for allocating a fraction of CPU time to the multimedia server. One clear distinction is between a static allocation and a flexible allocation. With static allocation, the start time and duration of each multimedia server instance are fixed beforehand. Then the on-line scheduler tries to guarantee the hard real-time tasks by scheduling them into the CPU time not used by the multimedia server. Therefore, scheduling of multimedia streams is separated from scheduling of hard real-time tasks and is not directly related to the planning-based scheduling algorithm. The static allocation can be considered a baseline and is not expected to perform very well. On the other hand, with the flexible allocation, each multimedia server instance is treated as one real-time task and dynamically scheduled with the planning-based scheduling algorithm. The start time of each multimedia server instance can be moved between its release time and its deadline minus server computation time. The release time and deadline are calculated based on the period of the multimedia server as described below. The scheduling overhead of the flexible approach is higher than

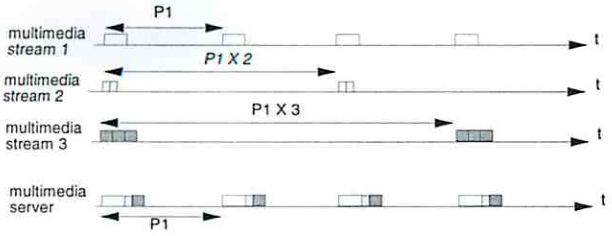

Figure 1. Proportional allocation of multimedia streams.

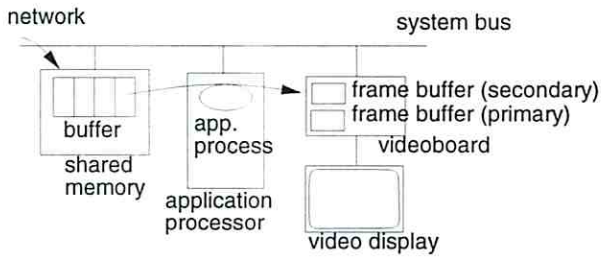

Figure 2. An example application system.

that of the static approach because the planning-based scheduler has to schedule multimedia server instances in addition to hard real-time tasks. However, schedulability of hard real-time tasks is much lower with the static allocation than with the flexible allocation since the former is much more restrictive in timing.

Proportional versus Individual Allocation. For both static and flexible allocation schemes, there are two ways to assign each multimedia task instance to the multimedia server instance. One is called *proportional allocation* where each task instance is split proportionally into the multimedia server. Suppose there are n different multimedia streams in the system. Let P_S be the period of the multimedia server, P_i be the period of the i-th multimedia stream, L_S be the time duration of each multimedia server instance and L_i be the estimated execution time of the task instance in the i-th multimedia stream. Then, since each task instance is divided into $\frac{P_i}{P_S}$ server instances, the computation time of the multimedia server instance L_S is given as

$$L_S = \sum_{i=1}^{n} (L_i \frac{P_S}{P_i}).$$

Figure 1 illustrates this allocation scheme. As the multimedia stream 1 has the shortest period P_1, the server has the same period as stream 1, namely P_1. In this example, the length of each server instance is the sum of the execution times of the task of stream 1, half of stream 2 and one third of stream 3.

Flexibility in task execution is needed especially when several multimedia streams are multiplexed. For example, Figure 1 only illustrates how the computation time of the server instance is decided, not the order in which tasks are executed within the server. In practice, due to the high variability in multimedia stream processing, it is virtually impossible to execute the tasks within the server in the way the figure shows. The only thing that the system has to guarantee is that every multimedia stream gets its requested fraction of time in the server. Although this lack of determinism is intolerable for hard real-time tasks, for multimedia tasks, some amount of jitter caused by the execution delay can be tolerated.

For example, in the architecture considered here, a typical type of processing of the multimedia task can involve taking frame data out of the buffer, processing it and putting it into the secondary frame buffer on the videoboard (Figure 2). At the end of the processing, the task issues the draw command to the videoboard, then the videoboard transfers the data on the secondary buffer with some processing into the primary buffer. The frame data written in the primary buffer will be displayed on the screen by hardware. Here, as long as the display commands are issued at some requested rate, the specific deadline of each issue does not necessarily have to be defined. At the same time, the transfer of frame data to the videoboard can be started just after the display command of the previous frame is issued. Therefore, the release time of the tasks do not have to be strictly enforced either. In fact, the execution time of a multimedia task depends largely on the amount of data it processes, thus it is sometimes difficult to estimate a priori the worst case execution time of the task. The amount of execution time needed to play back a single frame varies a lot and even the average execution time needed over a group of pictures shows considerable variations as a result of changes in scene or video content [**?**]. The adaptable scheduling introduced by the proportional allocation scheme is well suited for these various application requirements.

Another multimedia task assignment approach is to assign each multimedia task instance individually to a server instance. We call this the *individual allocation* scheme. Here again, the period of the server is the same as the minimum period of all multimedia streams multiplexed into the server. For example, in Figure 3, there are three multimedia streams and the stream with the shortest period is stream 2, thus the server has the same period as stream 2 and all the tasks in stream 2 are allocated to the server instances with their locations unchanged. Then the tasks in stream 1 and stream 3 are allocated to their nearest server instances. The server instance to which the task is assigned has to be located between the task's release time and deadline. If such

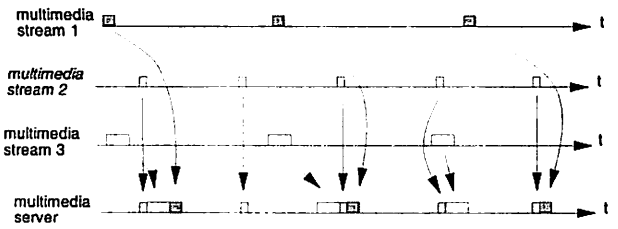

Figure 3. Individual allocation of multimedia streams.

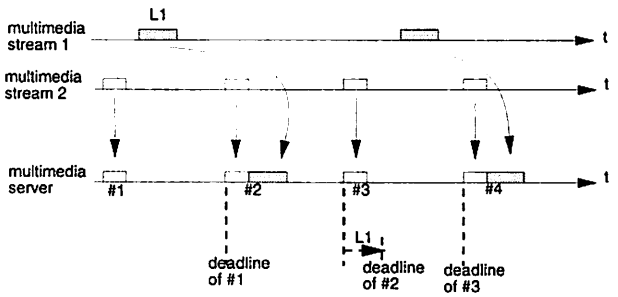

Figure 4. Deadlines of the flexible individual allocation.

a server instance cannot be found, a new server instance has to be created. The order in which tasks are executed inside the server can be decided by using the earliest deadline first algorithm. Each server instance has to keep state information on which tasks it is responsible for and in what order it has to execute them.

As opposed to the proportional allocation scheme, the individual allocation method can provide deterministic guarantee for each execution of the multimedia task instance. Each task instance is executed exactly in the allocated time when the static allocation approach is taken. Even if we take the flexible allocation scheme for the multimedia server instances, we can execute each task instance deterministically within its deadline by choosing the deadline of each multimedia server instance in the following way. Suppose we have two multimedia streams, stream 1 and stream 2 (Figure 4), and the computation time of a task instance in stream 1 is L_1. At first, we make a deadline of a server instance the same as the start time of its next server instance. For example, in the figure, the deadline of the server instance #1 is the start time of the server instance #2. Then we multiplex the stream 1 with the server. The first task instance of stream 1 is attached to the server instance #2 and the deadline of the server instance #2 is extended by L_1 because as long as the order of execution is maintained, the execution of the task instance of stream 2 within its deadline is guaranteed.

3.3. Scheduling Algorithm

In the previous section, we discussed four different multimedia server assignment policies. In general, any one of them may be chosen based on the system requirements and its performance for that system. Regardless of which one is chosen, at runtime, the scheduler takes the following steps.

We have to consider separately the cases when a multimedia stream comes into the system and when a hard real-time task enters. In the former case, if no multimedia stream already exists in the system, the scheduler creates a new multimedia server whose computation time and period are the same as those of the incoming stream. On the other hand, if one or more multimedia streams already exist in the system, the scheduler merges the new stream into the server using the chosen assignment policy (proportional or individual). If the period of the incoming multimedia stream is smaller than that of the current server, a new server is created with period equal to that of the new stream. Instances of the new server will replace those of the old one at the earliest possible time at which the changeover can occur without violating the QoS guarantees of the existing multimedia streams. An upper bound on this changeover time delay is the LCM of the periods of the existing multimedia sessions. After setting up the multimedia server for the incoming stream, the scheduler tries to schedule the hard real-time tasks that reside in the system. If we take the static allocation approach, we just try to put the hard real-time tasks into the unused CPU time outside the multimedia server using the planning-based scheduling algorithm described before. If we take the flexible allocation approach, we regard each multimedia server instance as one hard real-time task and schedule it along with other server instances and hard real-time tasks. If the scheduling is not successful, the incoming multimedia stream is rejected to ensure the executions of already guaranteed multimedia streams and the hard real-time tasks.

In both cases, the scheduler attempts to schedule all server instances whose period start times are before the latest deadline of the existing hard real-time tasks. If the scheduling is not successful, the incoming multimedia stream is rejected to ensure the executions of already guaranteed multimedia streams and the hard real-time tasks.

In the case of the arrival of a hard real-time task, the procedure is slightly different. If the deadline of the incoming task is earlier than the latest deadline of the existing tasks, the scheduler attempts to schedule the the current task set plus the new task. Otherwise, the scheduler needs to create more multimedia server instances whose period start times are before the deadline of the incoming task. After the creation of these server instances, the new task set will be tested for scheduling. If the scheduling is not successful, the

new hard real-time task is rejected. This scheduling procedure ensures that the already admitted multimedia streams or *real-time* tasks are always guaranteed to be executed no matter how many tasks arrive later. Of course, a different approach is possible here. If hard real-time tasks have higher priority over multimedia streams, we can reduce the QoS guarantees provided to existing multimedia streams so that a subsequent attempt at building a feasible schedule is more likely to succeed and the incoming hard real-time task is guaranteed. We will discuss this issue in Section 4.

Before actually running the scheduling algorithm, making a preliminary admission test with the estimated execution time may be helpful. That is, if the sum of the task's execution time is greater than the amount of CPU time that the system can provide, there is no way that the scheduler can create a feasible schedule. With this test, the system can take some actions much more quickly since the cost of this test is much less than that of the actual scheduling test. In order to make this admission test, the scheduler has to calculate the percentage of CPU time multimedia tasks use in a scheduling time period l and the percentage of CPU time hard real-time tasks use. Now let us call the ratios *multimedia server ratio* and *hard real-time task ratio* and denote them by Rs and Rr, respectively. In the case of the proportional scheme, since all the server instances have the same computation time and the same period, Rs is equal to (server computation time / server period). For example, if we have 20 ms of server computation time and 100 ms of server period, the multimedia server ratio Rs is 20% and that means 20% of the CPU time will be allocated to the multimedia tasks. Rs of the individual allocation scheme is sum of the computation times of the server instances divided by the scheduling period which is the length of time from the current time to the latest deadline of the hard real-time tasks. The hard real-time task ratio Rr is also sum of the execution times of the hard real-time tasks divided by the scheduling period. In order for the schedule to be successful, $Rs + Rr$ has to be at least less than 100% $\times$ (the number of processors). If $Rs + Rr$ is greater than 100% $\times$ (the number of processors), the incoming request is immediately rejected or a degradation approach is taken, depending on the policy in effect at that time.

3.4. Implementing the scheduling algorithm on an SGI Challenge Multiprocessor

The SGI Challenge multiprocessor system [?], Figure 5, is a shared memory, symmetric multiprocessor architecture. This architecture has four levels of memory hierarchy - two levels of cache (on-chip cache and cache on the CPU board), main memory and disk. There is a 100:1 access speed difference between successive levels of the memory hierarchy.

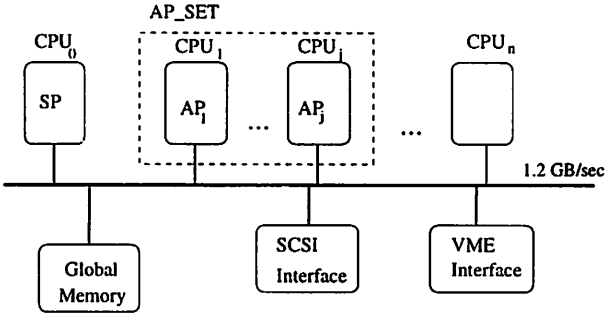

Figure 5. Architecture of SGI/Challenge Multiprocessor.

The processors and global main memory are connected via a 1.2 Gigabytes per second processor bus.

Details of Scheduling Mechanisms built into SGI's OS. $IRIX^{TM}$ [?, ?] is a commercial, UNIX based OS which has been optimized for multiprocessor performance. It offers many interesting features which are useful for supporting real-time applications. These include memory mapped I/O, asynchronous I/O, the option to lock pages in memory to avoid unpredictable page fault delays, the facility to direct interrupts to or away from specific CPUs, and to isolate and restrict subsets of CPUs to execute only specific processes using specific scheduling disciplines etc. The IRIX/REACT/PRO facilities which are capable of supporting real time and multimedia applications in certain scenarios are the the Frame Scheduler and the Deadline Scheduler.

The REACT/PRO Frame Scheduler isolates a CPU and uses a cyclic executive to schedule and dispatch selected processes on that CPU. It supersedes normal IRIX scheduling for this CPU and directs all other processing, daemons, and interrupt handling overheads away from it. Given a set of real time processes with periods and worst case execution times, the user has to compute the major and minor frame rates for the scheduler, and queue the processes for service in one or more minor frames. The frame scheduler then services the minor frames in order, once every major frame. In each minor frame, the processes queued for service in that frame are served in queue order, possibly multiple times, until the minor frame ends. It is possible to use multiple synchronized frame schedulers for concurrently executing a set of real time processes on a subset of CPUs. The frame scheduler is more suitable for static scenarios characterized by a fixed set of tasks whose requirements do not change over time. In more complex environments characterized by dynamic event arrivals, the frame scheduler is not suitable since the minor and major frame values may have to be potentially recomputed and allocation of processes to minor frame queues re-determined. To do this, the frame scheduler would have to be paused, its parameters modified and processes reassigned to minor frame queues, before the

scheduler can be restarted. This temporary pause in the scheduler would cause unacceptable disruption in service.

The Deadline Scheduler [?] attempts to guarantee execution rates to sessions. The admission control checks if the total CPU-time allocation for all the processes over a predefined frame interval is below a maximum limit and if so the requesting process is admitted. The processes are arranged in the scheduling queue according to time-to-deadline and are serviced in round-robin order. Although IRIX does implement the basic priority inheritance protocol [?] to prevent the unbounded priority inversion problem, blocking by lower priority tasks can still occur [?]. Unless the resources required by the real-time applications are carefully isolated, they may also be delayed due to blocking over a resource held by a process in the time-sharing class. However, the analysis for admission control does not account for the delay terms due to this blocking. Deadlocks can also occur and can result in violation of the guaranteed QoS. Also this scheduler is primarily suited to handling periodic tasks. The only way aperiodic hard real time tasks can be accommodated is by treating them as periodic for admission control (this makes the admission control very pessimistic) and explicitly removing the task from the scheduling queue at the end of its execution. It is also not clear how the deadline scheduling can provide the guaranteed rates over multiple processors.

Based on this discussion we can say that neither the Frame Scheduler nor the Deadline Scheduler built into IRIX/REACT/PRO suits our needs. In the following section, we outline how our planning based solution can be implemented on top of IRIX in the SGI/Challenge architecture, by using a different mechanism.

Details of implementing a Planning-based Approach. A supervisor process, called MASTER, executing at a very high priority on CPU_0, which is designated the system processor (SP), will group processors into a processor set called AP_SET. The processors in this set, called application processors (AP), will be actually executing the hard real time and multimedia application tasks and for predictable performance, need to be spared from unpredictable interrupt-driven workloads. MASTER isolates and restricts each AP in the set using the following steps:

- Specify that the system processor will perform all the overhead processing related to the scheduling clock interrupts.
- Isolate the APs from sprayed interrupts.
- Assign I/O interrupts to either the SP or a separate I/O processor.
- Restrict each AP from executing processes that are not explicitly assigned to it.
- Isolate each AP from TLB misses. As long as an isolated CPU executes only processes whose pages are locked into memory, it will receive no broadcast/TLB interrupts from other CPUs as actions by processes in other CPUs cannot change the address space mapping of any process on this CPU.

Now the system is configured to provide integrated support for multimedia and hard real time tasks. The supervisor running on the SP, executes the server allocation algorithm and the planning-based scheduler on dynamically arriving tasks. Initially the system is idle and then some tasks (multimedia sessions and aperiodic hard real time tasks) arrive. If MASTER is successful in finding a feasible schedule for the incoming workload, the outcome is a dispatch table with one column for each AP - this is the dispatch list for that AP. The list consists of a series of tuples (i, SST_i, SFT_i), where i is the process identifier, SST_i and SFT_i are the scheduled start time and scheduled finish time, respectively. (SST_i, SFT_i) defines a scheduling interval over which i is guaranteed to execute on the corresponding AP.

1. The master now allocates a dispatch task D_j to each AP_j using the $runon()$ command. D_j is given a priority in the real time class and is restricted to run only on that AP. In IRIX, the real time class is a band of priorities in the range 30-39. Processes allocated to this range do not have their priorities degraded, and the system accords them the highest importance next to kernel processes.
2. At this point, the dispatch task is the only ready-to-run eligible task on each AP and therefore the OS starts it.
3. The dispatch task D_j does the following:
   (a) It goes to the dispatch list in main memory for AP_j, finds the next tuple (i, SST_i, SFT_i). This indicates that process i has to be executed next from time SST_i to SFT_i.
   (b) If the scheduled start time $SST_i > T_{current}$, the current time, it will start a timer to expire at SST_i and go to sleep on that timer. The timer will be executing on the system processor SP. When it times out, an interrupt is sent to AP_j.
   (c) When the timer interrupt arrives, the ISR on the AP awakens the dispatcher D_j and returns. The dispatcher runs immediately, being the only eligible runnable task on that processor.
   (d) D_j now checks if the process i is already in the real time queue of the AP. If so it must have

executed at least once before on this AP, and been suspended at the end of its allocated time. The dispatcher then uses the *resume()* system call to make the process ready, starts a timer to expire at SFT_i and goes to sleep.

(e) If the process i is being executed for the first time on processor AP_j, the dispatcher D_j will

i. allocate the process to the real time priority class at a lower priority than D_j, and restrict the process to execute only on processor AP_j; and

ii. then start a timer to expire at SFT_i and go to sleep.

Note that since the dispatcher is executing at a higher nondegrading real-time priority than the process i, it will not be preempted by the latter, before it voluntarily suspends itself.

(f) Now i is the only eligible ready-to-run process on AP_j and so it is dispatched next. It now executes until it either finishes, or the time advances to SFT_i when a timer interrupt occurs.

(g) When the timer goes off, the ISR suspends process i if it has not yet finished, wakes up the dispatcher and returns. D_j now fetches the next tuple from the dispatch list and the whole protocol repeats itself.

The dispatch table is a shared data structure in global main memory, which is accessed by the different dispatchers from each AP as well as by the master scheduler-planner. Different dispatchers need to access different columns in the table and so do not interfere with each other. Also, the master and dispatcher on any AP are always working on different parts of the dispatch table. Whenever new tasks arrive at time T_{curr}, the master computes t, an upper bound on the time available for it to compute and return a feasible schedule if such a schedule exists. t is chosen large enough so that there is a high probability of finding a feasible solution within this time, but at the same time, the response of the scheduler to dynamic arrivals is not too slow. MASTER then draws a cutoff line in the dispatch table at $T_{curr} + t$ and attempts to make modifications to the part of the current schedule which are beyond this cutoff line. This avoids race conditions between the master and dispatcher tasks on accesses to the shared dispatch list. The implementation of the dispatch table can be similar to the one used in the Spring real time kernel [?].

The scheduling queues in IRIX are implemented in a distributed fashion to permit a high level of concurrent access [?]. There are local per-processor *squeues* on which only processes which are restricted to that processor and processes which have affinity for that processor are queued. Other processes are maintained on the central global queue. In our case, the APs are explicitly restricted to executing only the dispatcher and the hard real time tasks and multimedia servers as allocated to them by the master process. Also each hard real time task or multimedia server is restricted to be executed only on that AP to which it is queued. So the OS, when it needs to search for the next task to execute on a particular AP has to look only in its local *squeue*. All this together ensures that the performance is predictable and task executions are as laid out by the scheduler-planner. Note that to prevent the unpredictable delays and nondeterminism caused by page faults, all the pages of the supervisor process, the hard real-time tasks and the multimedia sessions, as well as shared pages (for example, the global dispatch table) should be locked into main memory.

4. Degradation of QoS

In our approach, the quality of service (QoS) requirements of the multimedia tasks are mapped into the computation time and period of the multimedia server. As stated in the previous section, one way of alleviating system overload is to degrade the QoS of the multimedia sessions assuming that hard real-time tasks have higher priorities over multimedia sessions. There are a couple of ways to achieve this degradation of multimedia QoS. For example, we can reduce the computation time of the multimedia server, increase the period of the server or even drop some of the server instances. However, deciding how to degrade the QoS of the multimedia sessions so that the scheduling of the hard real-time tasks will likely succeed and still keep the degree of degradation as low as possible is not very easy since the scheduling of the hard real-time tasks itself is a NP-hard problem. Moreover, the cost of the scheduling test is fairly high because the planning-based scheduler takes into account all the resources of every task. Therefore, our goal here is to find the best server adjustment plan, that is, the plan which not only gives a feasible schedule for all the tasks, but also produces a schedule with the highest value, i.e., gives the maximum amount of CPU time to the multimedia server, with a minimal number of scheduling tests.

Here we present a multilevel scheduling approach as a solution for the above problem. If the first scheduling attempt fails, the scheduler passes the information on the multimedia server and hard real-time tasks' requirements to the upper level algorithm. This upper level algorithm is referred to as *a server planner* in the following. The first step that the server planner takes is to lower the server ratio Rs as much as possible so that it satisfies $Rs + Rr + \text{margin} \leq 100\% \times$ (the number of processors). It then iterates as shown in Figure 6 to converge on a suc-

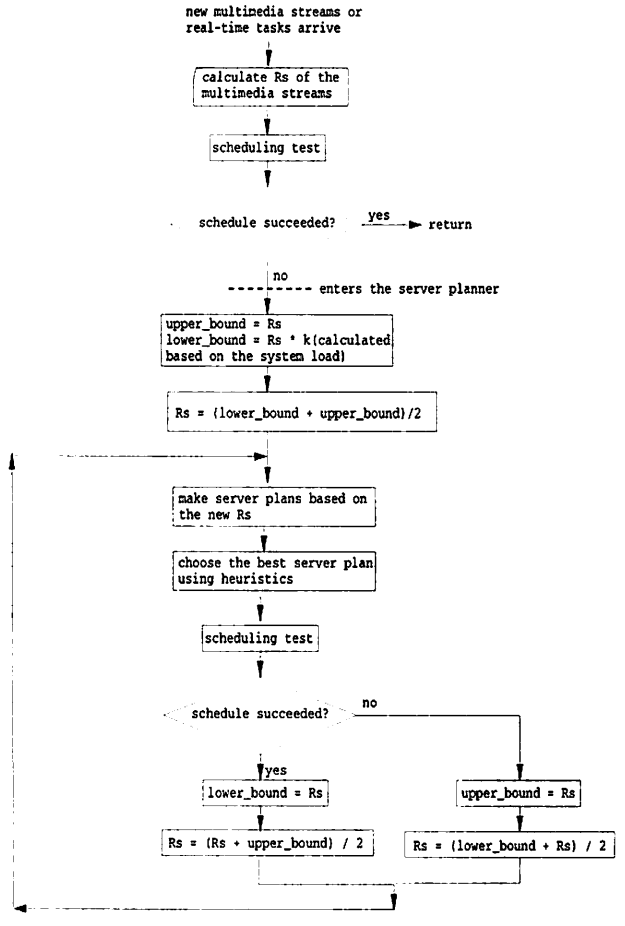

Figure 6. The iterative server rate adjustment algorithm.

cessful server ratio. Each time that a server ratio is chosen, the server planner makes several server arrangement plans. A server arrangement plan is a choice of a server computation time and server period such that the overall server ratio is met. For example, two server arrangements might be a multimedia server with computation time 2 and period 20, and computation time 1 and period 10. The server planner then chooses the best server arrangement using a heuristic that maximizes the laxity for hard real-time tasks and invokes the base planning-based scheduler. The latter tries to schedule all the tasks again with the new multimedia server arrangement. If it is not successful, the server planner chooses the next server arrangement plan with a lower Rs, and if successful, the planner chooses it with a higher Rs. This iterative process continues until the iteration count reaches a pre-defined number, i.e., the scheduler spends its allowed scheduling time, or the rate of change in Rs is less than some pre-defined amount. This scheduling process is summarized in the diagram in Figure 6.

It is important to note that the mechanism presented here degrades multimedia QoS in terms of the server computation time and period and how to quantify the resulting application QoS is still an open issue.

5. Simulation

5.1. Overview

The simulations are divided into roughly three parts. First, the four different strategies for the multimedia server scheduling obtained by combining two types of server allocation policies and two types of approaches for allocating individual multimedia streams to the server instances are compared to find out which combination provides the best performance. Second, different deadline and execution time distributions of hard real-time tasks are input to the scheduler to further evaluate the performance characteristics of the algorithms. Third, the effectiveness of the multilevel scheduler for QoS degradation of multimedia sessions is examined.

5.2. Task Generation

A task set generator generates a hard real-time task set and multimedia stream set for each simulation run. The real-time task set generated by this generator is, by itself, a feasible set. That is, in the absence of the multimedia streams, an optimal scheduler can find a schedule for the task set. The following parameters are used to generate the hard real-time task sets:

1. Probability that a task uses a resource, Use_P.
2. Probability that a task uses a resource in shared mode, Share_P.
3. The minimum processing time of tasks, Min_C.
4. The maximum processing time of tasks, Max_C.
5. The minimum deadline of tasks, Min_D.
6. The maximum deadline of tasks, Max_D.
7. The schedule length, L.

The schedule created by this task set generator is in the form of a matrix M which has r columns and L rows. Each column represents a resource and each row represents a time unit. In order to illustrate the process of task set generation, we assume that there are n processors and m other resources, i.e., the total number of resources is $n + m$. Resource items $1...n$ represent n processors. The task set generator starts with an empty matrix. It then generates a

task by selecting one of these n processors with the earliest available time and then requests the m resources according to the probabilities specified in the generation parameters. The generated task's processing time is randomly chosen using a uniform distribution between the minimum processing time and the maximum processing time. The task set generator then marks on the matrix that the processor and resources required by the task are reserved for a number of time units equal to the task's computation time starting from the aforementioned earliest available time of the processor. The task set generator generates tasks until the remaining unused time for each processor, up to L, is smaller than the minimum processing time of a task, which means that no more tasks can be generated to use the processors. Then the largest finish time of a generated task in the set becomes the task set's *shortest completion time*, SC. As a result, we generate tasks according to a very tight schedule without leaving any usable time units on the n processors between 0 and SC. However, there may be some empty time units in the m resources. The deadline of each task was chosen between (finish time of the task + minimum deadline Min_D) and (finish time of the task + maximum deadline Max_D). The output of this task set generator is a file written in the Spring System Description Language (SDL) [?]. The file describes all the task information such as timing and resource usage specifications needed by the planning scheduler. It is compiled by the Spring compiler and fed into the simulator. The task generator also places multimedia stream information into this file. In these experiments we used 5 multimedia streams whose characteristics are described in the next subsection.

5.3. Simulation Method

In the simulation, the performance of various server assignment policies are evaluated according to how many of the N feasible task sets are found schedulable. Here, we are interested in whether or not all the real-time tasks in a task set and multimedia server instances can finish before their deadlines. Therefore, the most appropriate performance metric is the schedulability of task sets. This metric called the success ratio SR is defined as

$$\frac{\text{total number of task sets found schedulable}}{N, \text{the total number of task sets}}.$$

All the simulation results shown in this section are obtained from the average of six simulation runs. For each run, we generate 500 task sets (i.e., N = 500). The maximum 95% confidence interval of any data point was 3.3% of the success ratio. The system tested consisted of three processors and 12 nonprocessor resources. Use_P is 0.7 and Share_P is 0.5. Although the primary purpose of this simulation is to compare the different server assignment policies and examine the effects of changing the parameters, we normalized the simulation time unit into milliseconds and chose realistic values for the parameters so that we could assess the feasibility of our approach to some extent. The schedule length L is 300 ms, and a task's computation time is randomly chosen between Min_C and Max_C. Thus, for example, when Min_C = 10 and Max_C = 30, each task set has between 40 and 50 tasks. Min_D is 60 and Max_D is 90, thus a deadline of each task is randomly chosen between 60 ms and 90 ms. We put five multimedia streams with different computation times and periods into one processor. We made one of the five streams a baseline stream with a rate of 30 frames/sec and made four other streams with a 20%, 40%, 60% and 80% lower rate than the baseline, respectively. That is, the period of the baseline stream is 33.3 ms and that of the second stream is 40.0 ms (33.3 ms × 1.2). Similarly, the third, fourth and fifth streams have a period of 46.6 ms (33.3 ms × 1.4), 53.3 ms (33.3 ms × 1.6), and 59.9 ms (33.3 ms × 1.8), respectively. These periods correspond to frame rates of 25, 21.4, 18.8, and 16.7 frames/sec. These frame rates were kept constant throughout the simulations and only their computation times were varied. Each of the five streams consumes the same amount of CPU time on average, that is, if the computation time of a task instance in the baseline stream is 5 ms, computation time of a task in other streams are 6 ms (5 ms × 1.2), 7 ms (5 ms × 1.4), 8 ms (5 ms × 1.6), and 9 ms (5 ms × 1.8). If these tasks are allocated to the server proportionally, the computation time of each server instance is 25 ms ($5\text{ ms} + 6\text{ ms} \times \frac{33.3\text{ ms}}{33.3 \times 1.2\text{ ms}} + 7\text{ ms} \times \frac{33.3\text{ ms}}{33.3 \times 1.4\text{ ms}} + 8\text{ ms} \times \frac{33.3\text{ ms}}{33.3 \times 1.6\text{ ms}} + 9\text{ ms} \times \frac{33.3\text{ ms}}{33.3 \times 1.8\text{ ms}} = 5\text{ ms} \times 5$). If they are allocated to the server individually, each server instance has a different execution duration and possibly a different period.

5.4. Simulation Results

Comparison of the Server Allocation Policies. The simulation results with the different multimedia server assignment policies and no degradation policy are shown in Figure 7. The X axis represents computation time of the baseline multimedia stream as described in the previous section. In all the simulation results shown, the success ratio keeps decreasing as the multimedia computation time increases. This is because the increased multimedia computation time leaves less CPU time for hard real-time tasks, thus the tightness of the scheduling increases. The results show that the flexible allocation works much better than the static allocation. For example, when the baseline multimedia computation time is 1.2 ms, the success ratio of the two static approaches goes down to 0%, whereas the flexible approaches achieve 80% and 100%. The proportional allocation also works better than the individual allocation. Especially when the multimedia computation time is relatively short, for ex-

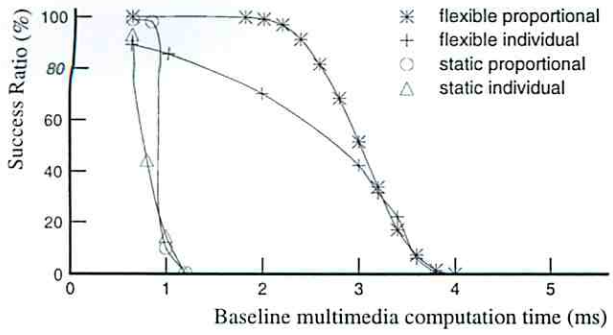

Figure 7. The server types and the success ratio (period = 33ms).

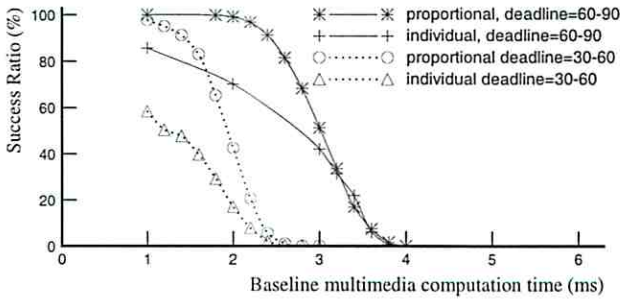

Figure 8. Effect of deadline on success ratio.

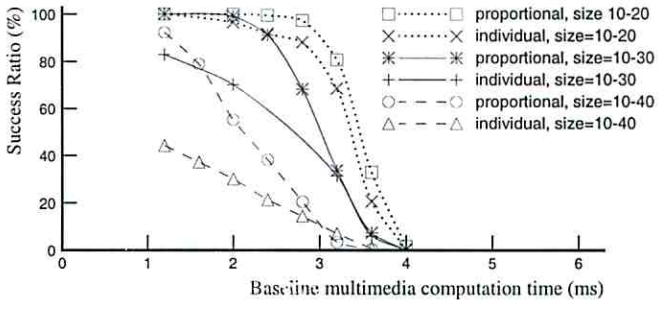

Figure 9. Effect of hard real-time task size on success ratio.

ample 2 ms, the flexible proportional approach has 99% success ratio, but the flexible individual strategy has only 75%. Although these results are as expected, the significant difference between the static allocation and the flexible allocation approaches is noteworthy. Moreover, the difference between the flexible proportional allocation and the flexible individual allocation indicates that the price we have to pay to ensure that every individual instance of any MM session always executes within its deadline is quite expensive. This is mainly because the deadlines of the server instances in the flexible individual allocation are sometimes much shorter than those in the proportional one. From these results, we can conclude that the flexible proportional assignment policy provides the highest performance among the combinations tested in terms of the success ratio.

Effect of Changing Parameters. In the following simulations, only the flexible allocation schemes are examined. Figure 8 shows the effect of changing the deadlines of hard real-time tasks on the success ratio. The plain lines are the success ratio when deadlines are chosen between 60 ms to 90 ms, and the dotted lines are those when deadlines are between 30 ms and 60 ms. Here, the shape of the curves in the different deadline ranges looks almost the same, that is, the curves just shifted horizontally. For example, with the deadline ranges between 60-90 ms the success ratio of both proportional and individual allocations drop to 0% when the baseline computation time is 4 ms, whereas with the deadline ranges between 30-60 ms, they drop to 0% when the baseline computation time is only 2.8 ms. These results indicate that deadlines of hard real-time tasks significantly affect the upper bound of the multimedia server computation time and they are almost proportional.

In Figure 9, results are shown where the execution time of hard real-time tasks is chosen from the different ranges. The dotted line shows the ranges between 10-20 ms, the plain line 10-30 ms, and the dashed line 10-40 ms. Although the CPU loads in those three cases are almost the same because of the task generation procedure, the success ratio varies significantly. In other word, granularity of the hard real-time tasks largely affects the schedulability. In this scheduling approach, the scheduling of the multimedia server instances is fairly tight because the deadline of each instance is relatively short. For example, when the period of the server is 33 ms and its computation time is 10 ms, the laxity of the server is only 23 ms. When the maximum size of the hard real-time tasks is 40 ms (the dashed line), it is difficult for them to fit in between the server instances. In the figure, the success ratio is only 55% with proportional allocation and 30% with the individual allocation when the server's computation time is 10 ms (it corresponds to 2 ms computation time for the baseline multimedia stream). These simulation results have shown the sensitivity of the success ratio to the size of hard real-time tasks.

Degradation of Multimedia QoS. The iterative improvement approach discussed in Section 4 was also evaluated with simulations. Here, the server ratio Rs is adjusted toward the highest value after every scheduling attempt, that is, when the n^{th} scheduling attempt is successful, Rs for the $(n+1)^{th}$ attempt is increased to $(Rs + \text{upper\_bound})/2$ and when it is not successful, Rs is decreased to $(\text{lower\_bound} + Rs)/2$. For each generated task set, this scheduling attempt was iterated 9 times and Figure 10 shows the average success ratio over 500 task sets $\times$ 6 simulation runs achieved within n attempts for each task set. As we can see in Figure 10, all the first scheduling attempts in the simulations failed. (The success ratio = 0%

when the iteration count = 0.) For 78% of the generated task sets, the second scheduling attempt (the first iteration) succeeded. The figure shows that for all the task sets, the scheduling succeeded within 4 iterations. In Figure 11, the Y axis is the degradation ratio r which indicates the degree of degradation from the initial requested QoS. r is defined as

$$\frac{\text{degraded Rs used in the next iteration}}{\text{initial Rs (application requirement)}}$$

(r = 100% means that the server was not degraded at all.) The plotted degradation ratio were obtained by averaging the highest Rs which gave successful schedules in the n iterations. The results show that we can get significant improvement in the server ratio within several iterations.

In Figure 11, we show how close the iteration scheme comes to the minimum loss in multimedia service that is possible due to the presence of hard real-time tasks. The dashed horizontal line in Figure 11 indicates the upper limit of the degradation ratio. From the figure we see that the achieved ratio gets fairly close to this limit. A more elaborate iterative approach may be able to get a higher degradation ratio, but it will require a larger number of iterations.

In summary, the results show that the scheduler can provide a feasible schedule within a few iterations without decreasing the multimedia computation time too much.

5.5. Influence of Context Switch Overhead

So far our results were based on the assumption that context switch overheads are negligible. Here we discuss the effects of context switching. The individual allocation scheme allocates an entire multimedia task instance to a single multimedia server instance, and hence does not add any extra context switching overheads.

The proportional allocation strategy allocates a periodic multimedia stream to a multimedia server with same or smaller period by dividing its periodic time allocation across multiple server periods. This method of servicing a single multimedia task instance in multiple disjoint time intervals requires the system to switch the instance in and out multiple times before it gets its full allocation. The associated additional context switching overhead due to this fragmentation constitutes an additional load on the system. However, this overhead is a function of the period lengths and is constant over different multimedia workloads for a given period length distribution of the multimedia streams.

Assume there are n multimedia tasks in the system. Task i, $1 \leq i \leq n$ has period P_i and estimated worst case execution time per period is L_i. The time to switch a task in or out is a fixed C_s. Let U_{mm} be the offered multimedia load to the system, and U_{input} the associated overhead of switching each task instance in and out once during its period. Let U_{ovhd} be the additional context switching overhead introduced by the proportional allocation method. Let U_{cs} be the total context switching overhead at the end of the server allocation phase. Then, $U_{mm} = \sum_{i=1}^{n} \frac{L_i}{P_i} * 100, U_{input} = \sum_{i=1}^{n} \frac{2*C_s}{P_i} * 100, U_{cs} = \sum_{i=1}^{n} \frac{2*C_s}{P_{server}} * 100$, where $P_{server} = \min_{j=1}^{n}(P_j)$. Therefore, $U_{ovhd} = U_{cs} - U_{input} = 200 * C_s * [\frac{n}{P_{server}} - \sum_{i=1}^{n} \frac{1}{P_i}]$.

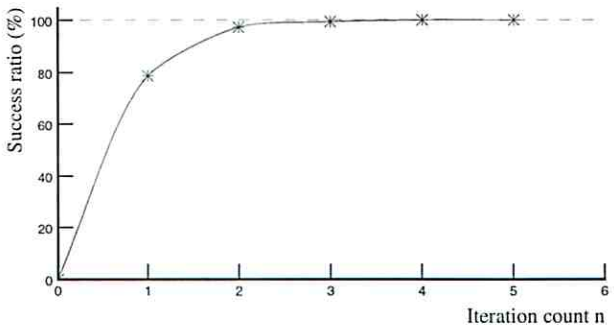

Figure 10. Iteration count and the success ratio.

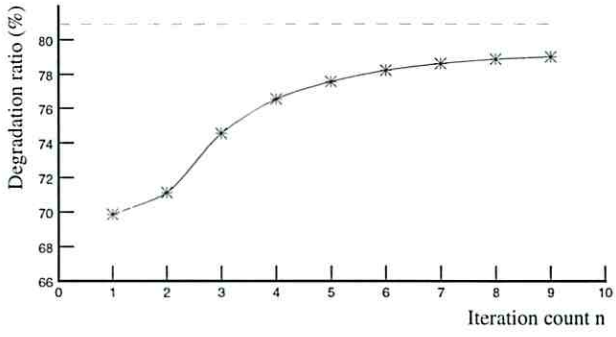

Figure 11. Iteration count and the degradation ratio.

U_{cs}, the total context switching overhead for proportional allocation, is constant for a fixed number of multimedia streams and a given server period. U_{input} is maximum = U_{cs} if all the periods are identical and decreases as the period lengths diverge. U_{ovhd} is minimum (i.e., equal to zero) if all the periods are identical and increases as the period lengths diverge. If the lengths are similar, there is less fragmentation of task instances and the context switching overhead U_{ovhd} is low. If the period of a multimedia stream is much larger than the period of the smallest stream (i.e., the period of the multimedia server), then, this allocation strategy causes more fragmentation and the number of introduced context switches is larger. The percentage of additional context switch overhead caused by proportional allocation depends only on the relative lengths of the periods of the different

C_s (μsec)	Introduced Overhead U_{ovhd} (%)
10	0.076
20	0.153
30	0.229
40	0.306
50	0.382
60	0.458
70	0.535
80	0.611
90	0.688
100	0.764

Figure 12. Overhead introduced by the proportional allocation strategy when the periods are 33.3 ms, 40.0 ms, 46.6 ms, 53.3 ms, and 59.9 ms.

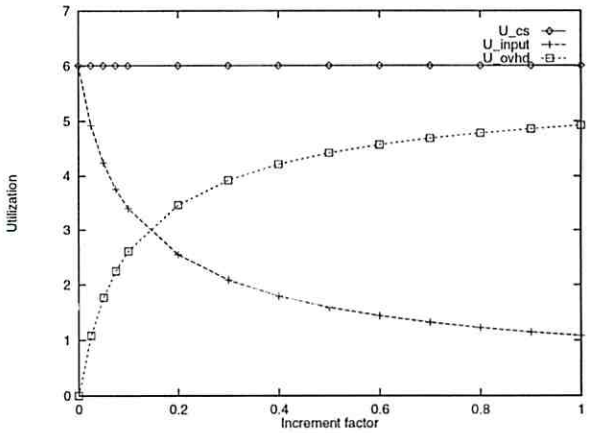

Figure 13. Context Switch overhead as a function of variance in the periods of different multimedia streams.

multimedia streams and is independent of the multimedia workload. For a given number of streams, for a given period distribution, the overhead is fixed across different loads - the overhead as a percentage of the offered load is a decreasing function of the load.

Simulation studies of the previous section assume that the different multimedia streams have very similar periods. There are 5 multimedia streams with periods 33.3 ms, 40.0 ms, 46.6 ms, 53.3 ms, and 59.9 ms - the longest period is 1.8 times the smallest. In Figure 12, we show how the introduced overhead U_{ovhd} for the above scenario varies as the context switch time C_s ranges from 10μsec to 100μsec. Even for very large context switch times (e.g., $C_s = 100\mu$sec), the introduced overhead is a very low 0.764%. For reasonably fast processors, the context switching times are lower and the associated overhead even more insignificant - for eg., for $C_s = 50\mu$sec, the overhead is a mere 0.382%. This overhead is too low to affect the performance of the scheduling algorithm and so we have ignored it in the reported simulations.

But, in situations where the session periods vary over a wide range and the introduced overhead is significant, it can affect task schedulability, and we need to account for this additional load in our simulation model. Consider an example scenario where the context switch overheads become significant. In Figure 13 we see how the different context switch overheads vary as the variance in period lengths changes. We consider 20 multimedia streams, per stream utilization = 3.003%, and smallest period = 33.3 ms. This translates to a multimedia server period P_{server} = 33.3 ms, total multimedia workload U_{mm} = 60.06%. We now vary the period lengths of the different multimedia streams as follows. The periods are generated in increasing order with $P_i = P_{server} * [1 + (i-1) * x]$, $1 \leq i \leq 20$. Here x is a parameter which specifies how far the period lengths vary from each other. We analytically compute and then plot U_{ovhd}, U_{input}, and U_{cs} (corresponding to context switch time $C_s = 50\mu$sec) for different values of x to get Figure 13.

Here, the context switch overheads are considerable. From the figure we see that U_{input} is = 6.006% (U_{ovhd} is = 0) if all the periods are identical and decreases (U_{ovhd} increases) as the period lengths diverge. For similar period lengths, U_{input} dominates and U_{ovhd} is not significant. But as the periods diverge, although U_{input} decreases, due to longer periods of some of the streams, U_{ovhd} rapidly increases to become the dominant contributor to U_{cs}. Note that the total context switching overhead is U_{cs} under the proportional allocation scheme and U_{input} under the individual allocation method. For small variance in periods, the overheads for the two approaches are similar. As the variance in period lengths increases, U_{input} increases, U_{ovhd} decreases and U_{cs} remains constant. So the total overhead for the proportional algorithm remains fixed while that for the individual allocation scheme decreases, increasing the difference in overhead between the two approaches in favor of the individual scheme. The effect of this on the schedulability of the hard real time and multimedia tasks needs to be studied. U_{ovhd} and hence, U_{cs} for the proportional scheme could be reduced by limiting the amount of fragmentation of task instances. For example, if there is a session with period $P > P_{server}$, but its periodic *wcet* can be accommodated in a single instance of the server, we allocate it as such, and no additional switching overheads are introduced. This is one of the ways in which the detrimental effects of widely varying session periods can be reduced. We plan to explore such possibilities.

6. Conclusion

In this paper, we presented a solution for an integrated platform that supports multimedia and hard real-time applications. We described how a scheduling solution would fit within an actual system. Then we presented the multimedia server scheduling algorithm which enables guaranteed execution of both *soft* real-time multimedia processes and traditional hard real-time control processes by using a planning-based scheduling approach. There are four possible policies for assigning multimedia tasks to the servers in this integrated scheduling, and the simulation results indicated that with the flexible proportional approach, we can get reasonable performance even when there are multiple multimedia streams in the system. The results showed that the algorithm can be used in practical application environments although it has to be noted that the performance depends on the computation time of the multimedia tasks and real-time tasks and the tightness of their deadlines.

This scheduling solution also supports an adaptive QoS degradation of multimedia sessions during system overload. We showed through simulations that this degradation approach can provide high CPU utilization without degrading deterministic guarantees for hard real-time tasks.

Acknowledgment

The authors wish to thank Gary Wallace for his help with the implementation of this algorithm in the Spring simulator.

References

[1] D. P. Anderson, Metascheduling for Continuous Media, *ACM Transactions on Computer Systems*, Vol. 11, No. 3, Aug. 1993, pp. 226-252.

[2] J. M. Barton and N. Bitar, A Scalable Multi-Discipline, Multiprocessor Scheduling Framework for IRIX, *IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing*, April 25, 1995.

[3] D. Cortesi et. al., REACT (TM) Real-Time Programmer's Guide, Document Number 007-2499-001, *Silicon Graphics Inc.*, Mountain View, CA 94043-1389.

[4] A. Guha, A. Pavan, J. Liu, A. Rastogi and T. Steeves, Supporting Real-Time and Multimedia Applications on the Mercuri Testbed, *IEEE Journal on Selected Areas In Communications*, Vol. 13, No. 4, May 1995, pp. 749-763.

[5] K. Jeffay and D. Bennett, A Rate-Based Execution Abstraction For Multimedia Computing, *Proc. 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, Durham, New Hampshire, April 18-21, 1995, pp. 67-78.

[6] D. I. Katcher, K. A. Kettler and J. K. Stronsnider, Real-Time Operating Systems for Multimedia Processing, *Proceedings of Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, May 4-5, 1995.

[7] D. Niehaus, J. A. Stankovic and K. Ramamritham, The Spring System Description Language, UMASS CS TR 93-01, January 1991.

[8] D. Niehaus, J. A. Stankovic and Krithi Ramamritham, A Real-Time Systems Description Language, *IEEE Real-Time Technology and Applications Symposium*, May 1995.

[9] K. Ramamritham, J. A. Stankovic and P. Shiah, Efficient Scheduling Algorithms for Real-time Multiprocessor Systems, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, April 1990, pp. 184-194.

[10] L. Sha, Ragunathan Rajkumar and John P. Lehoczsky, Priority Inheritance Protocols: An approach to Real-Time Synchronization, *IEEE Transactions on Computers*, Vol. 39, No. 9, Sept. 1990.

[11] J. A. Stankovic and K. Ramamritham, The Spring Kernel: A New Paradigm For Real-Time Systems, *IEEE Software*, May 1991, pp. 62-72.

[12] J. A. Stankovic, Continuous and Multimedia OS Support In Real-Time Control Applications, *Proceedings of Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, May 1995, pp. 8-11.

[13] R. Steinmetz, Analyzing the Multimedia Operating System, *IEEE Multimedia*, Spring 1995, pp. 68-84.

[14] Y. Taniguchi, A. Akutsu, Y. Tonomura and H. Hamada, An Intuitive and Efficient Access Interface to Real-Time Incoming Video Based on Automatic Indexing, *Proceedings of ACM Multimedia*, 1995, pp. 25-33.

[15] D. L. Tennenhouse, J. Adam, D. Carver, H. Houh, M. Ismert, C. Lindblad, B. Stasior, D. Weatherall, D. Bacher and T. Chang, A Software-Oriented Approach to the Design of Media Processing Environments, *Proceedings of the International Conference on Multimedia Computing and Systems*, Boston, MA, May 1994, pp. 435-444.

[16] C. A. Waldspurger and W. E. Weihl, Lottery Scheduling: Flexible Proportional-Share Resource Mangement, *Proceedings of the First Symposium on Operating System Design and Implementation*, November 1994.

[17] R. Yavatkar and K. Lakshman, A CPU Scheduling Algorithm for Continuous Media Applications, *Proc. 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, Durham, New Hampshire, April 18-21, 1995, pp. 223-226.

[18] H. J. Zhang, C. Y. Low, S. W. Smoliar and J. H. Wu, Video Parsing, Retrieval and Browsing: An Integrated and Content-Based Solution, *Proceedings of ACM Multimedia*, 1995, pp. 15-24.