A MULTIMEDIA SERVER ON THE REAL-TIME SYSTEM

H. KANEKO and J. A. STANKOVIC

CMPSCI Technical Report 96-11

January 1996

A Multimedia Server on the Spring Real-Time System *

Hiroyuki Kaneko, John A. Stankovic
Computer Science Department
University of Massachusetts
Amherst, MA 01003

January 16, 1996

Abstract

An integrated platform which is capable of meeting requirements of both traditional real-time control processing and multimedia processing has enormous potential for accommodating various kinds of new applications. However, few, if any, research or commercial systems successfully provide architectural and OS mechanisms which can efficiently support both deterministic hard real-time computation and less deterministic multimedia soft real-time computation. In this paper, we propose a multimedia server executing on the Spring real-time system to provide different classes of guarantee to support both types of processing. Using a planning based scheduler, the Spring system is a multiprocessor real-time system which was developed to satisfy the requirements of deterministic guarantees for hard real-time tasks. Adding a multimedia server enables Spring to also support multiple periodic multimedia streams with a capability for QOS graceful degradation of the multimedia streams during system overload. In this paper we address realistic system implementation issues and develop multimedia server scheduling algorithms. Our performance evaluation demonstrates both that a multimedia server algorithm based on a *flexible, proportional* allocation scheme provides the best performance and that simple iterative scheduling is adequate to handle graceful degradation of the multimedia streams.

1 Introduction

Multimedia applications have real-time requirements such as delay and jitter tolerance and require suitable operating system supports which are different from the widely used and time-sharing based operating systems. However, real-time requirements of multimedia applications are also different from those of traditional real-time applications such as plant control, aircraft piloting or factory automation. In the traditional real-time applications, not only the correctness of computation, but also timeliness of execution are required. Tasks have to be executed within their deadlines otherwise the failure causes some destruction of equipment or threat to human life. On the other hand, real-time requirements for multimedia are usually less stringent in such a way that missing a deadline of a task does not necessarily mean a significant failure. Multimedia applications usually involve human interactions and humans are not as sensitive to timing performance of the system as traditional real-time control applications. Dropping certain frames from audio and video data streams may not be perceptible in some applications. Applying the traditional real-time system

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

approach to multimedia applications is ineffective and expensive since real-time systems often take pessimistic approach in that resource allocations are deterministic and usually based on the worst case execution time of tasks.

On the other hand, systems designed for multimedia applications are not suitable for the traditional real-time applications neither, because they lack mechanisms to support the stringent timing requirements of those applications including resource and precedence constraints. For example, the rate monotonic (RM) and the earliest deadline first (EDF) scheduling algorithms are most widely used in the on-going multimedia research systems and testbeds [9]. However, the RM algorithm can handle only periodic tasks and does not work well in a dynamic environment. EDF is optimal only for a single processor and does not scale to multiprocessor systems. EDF also exhibits non-deterministic behavior in an overloaded situation. Accommodating both multimedia and traditional real-time applications is therefore a challenging research issue. However, little attention has been paid to the coexistence of these applications. For example, the Mercuri system [1] 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 present any provision in terms of real-time scheduling and end up with providing best effort services.

This paper presents a mechanism to provide the coexistence of multimedia applications and traditional hard real-time applications using the Spring system [8]. The Spring system is a distributed real-time system in which tasks are scheduled in a heuristic manner with their resource and precedence constraints being taken into account. The system is predictable in a way that it guarantees execution of a task within its deadline as soon as it arrives at the system. The system aims at both deterministic and dynamic support for traditional hard real-time applications. We show that we can also support the *soft* real-time requirements and QOS graceful degradation of the multimedia applications with the Spring system by incorporating a multimedia server into the system. That is, we partition cpu time into two parts and allocate multimedia tasks and hard real-time tasks separately to each partition. In the partitions for multimedia tasks, the multimedia server controls those tasks with its own scheduling discipline. While the partitioning scheme is straightforward in principle, it does not work accurately in most operating systems because of the insidious effects of the two classes of tasks (real-time control and multimedia) on each other via resource interactions, unbounded waits, priority inversions, virtual memory uncertainties, etc. These problems have been carefully resolved in the Spring kernel.

The rest of this paper is organized as follows. Section 2 introduces several applications which can benefit by integrating hard real-time control and multimedia. In section 2 how scheduling solutions can be integrated into an actual system architecture is also discussed. Section 3 presents the integrated scheduling algorithms along with additional system implementation details. 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 and the System Architecture

2.1 Applications of the Integrated Platform

As technology such as high performance CPUs, memory, disks and high speed networks become 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 derive reasons for the need of the 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 require new solutions. For example, suppose in a power plant or industrial manufacturing plant, plant operators monitor situations in the different locations of the plant via cameras and control actuators based on this monitoring. Currently these analog video monitoring systems and digitized controlling systems are implemented in completely separate platforms. Replacing these redundant systems with the 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 the cost, the integrated digitized system can provide more functionality. For example, 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. It is also possible to capture one of the scenes out of the video stream and store it on the disk for later use. Many companies are pursuing this application including Honeywell and Mitsubishi.

Second, *computer-participative* multimedia applications are another emerging trend in multimedia research [11]. 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 upon 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 [11]. In this type of application, input data have to be manipulated or filtered by software rather than hardware because large extent of flexibility in design is required. Similar applications can be seen in [14] and [10]. 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. Another example is a manufacturing system where the system detects the kind of objects on a belt conveyer which pass by the video camera in a factory. We can decide the next action to take according to the type of object the system has detected. Similar techniques can be exploited in conjunction with computer vision and robotics.

Third, many other examples can be found in military applications such as controlling the fly by wire Comanche helicopter through trees, telephone wires, etc. and "looking for" enemy soldiers or vehicles based on processing video and audio data.

2.2 System Architecture

We consider that single processor systems do not scale well for these applications since multimedia processing is sometimes very computationally intensive (e.g., the Comanche helicopter uses a multiprocessor as the main processing engine). In some of the ongoing research which uses multiprocessor approaches, some of the processors are dedicated to multimedia processing and others to traditional real-time processing e.g., [4]. Although this approach can provide good isolation of one type of processing from another, it can not achieve high utilization of system resources in a dynamic environment. As an example, 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. Another disadvantage of the separated processors is the inefficiency in processing of the integrated applications. That is, in the integrated applications described above, both multimedia processes and real-time processes need to refer to the multimedia data streams. If we have separated processors, the data streams have to be copied through main memory of the processors, or even if the data stream is put on a shared memory, frequent references to the data via the system bus may cause saturation of the bus. Having both types of processes residing in one processor may solve these problems. 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. The Spring system with some modifications is suitable for this approach.

In the Spring system, three nodes are connected with a fiber optic ring and each node consists of several processors connected with a local bus. Application processes are scheduled by the Spring scheduler in a system processor (SP) and executed on application processors (APs). The executions of tasks are non-preemptive[1] and external interrupts are handled only by the SP or I/O processors (IP) which are attached to a node and dedicated only to I/O processing. When we consider the distributed multimedia system, video and audio streams are transferred via networks. In most cases, data from networks are received with interrupt driven mechanisms, especially when the network has high bandwidth, otherwise the receiving buffer overflows. In the Spring system, this network protocol processing has to be done by the SP or IP since application tasks are executed non-preemptively in order to maintain predictability and APs do not handle interrupts. Even if we mask interrupts during the execution of hard real-time tasks and allow the multimedia server to get interrupts, the amount of data the system receives via networks during the mask will be beyond the capacity of any practical buffer size. Therefore, APs are responsible for processing hard real-time tasks and multimedia processing other than network processing, for example, decompression, expansion/contraction or filtering. The kernel on the SP or IP gets data from the network interface, performs some protocol handling and copies them to the shared memory in the node. APs take the data from the shared memory and performs some additional processing. Instead, if it is possible to implement all the network processing on the network interface hardware, then all that the application software has to do is to read the incoming data stored in a buffer on the interface card and transfer it to the output devices. The interface card needs to be intelligent enough to have a per session buffer space. That is, each application registers its buffer space at the time of initialization and the interface card demultiplexes the incoming stream and distributes packets into the buffers. Although this structure provides simplicity in architecture and good performance, its large buffer requirement and supposedly high cost because of the specialized design and little flexibility of the interface card make this configuration less feasible. The rapid change in network protocols in the real world also prevents us from taking this static approach.

As stated above, data from the network needs to be further processed with decompression, expansion/contraction, transformation from YUV space to RGB space, etc. Although all of this processing also can be implemented in hardware, decompression and expansion/contraction are more likely done by software since there are still too many video compression schemes such as Cinpack, MPEG-1 and MPEG-2. When the application requires video filtering as described in the above examples, implementing those filtering functions as software is preferable rather than as hardware since we may want to have flexibility in the filtering. We can create and choose any types of filters depending on the application if we have filtering libraries as presented in [11]. It will

[1]Task sizes are small because execution units (processes) produced by users are broken into tasks through compilations by the Spring-C compiler.

not take long until available processing power will become powerful enough to do the filtering as a software process. Computational entities which are executing on APs will be these data processing processes.

3 Multimedia Server in the Spring System

3.1 Background

Before presenting the multimedia server based scheduler, we briefly describe the original Spring scheduling approach. The Spring scheduler is a planning-based scheduler that dynamically generates schedules 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 arrive at the system, it attempts to assign a starting time 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. In the course of the scheduling, the scheduler allocates the resources to a task one by one based on heuristics. If the feasible schedule cannot be found, the node rejects the new set of tasks or sends them to the other nodes and reverts to its previous schedule, in which all tasks are guaranteed. This is essentially an admission control paradigm and a reservation based system.

On the basis of this Spring system, 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 executions of multimedia tasks. Task executions of more than one 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 needed to schedule these task instances is too high and as mentioned above, the deterministic guarantee for the multimedia tasks are often not necessary.

3.2 Multimedia Task Allocation Policies

In this paper, we investigated both static and flexible allocation schemes (Section 3.2.1) as well as proportional and individual allocation schemes (Section 3.2.2). This gives rise to four different combinations. Namely,

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

Section 3.2 discusses these combinations and Section 3.3 presents how these various combinations are integrated with the Spring scheduler.

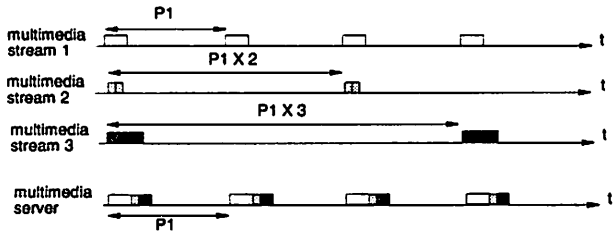

Figure 1 : Proportional allocation of multimedia streams

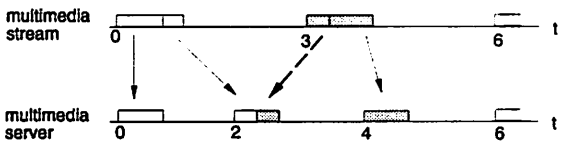

Figure 2 : Start time of each task in the proportional allocation

3.2.1 Static and 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 the static allocation, 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 Spring 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 Spring 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 the deadline are calculated based on the period of the multimedia server as described below. The scheduling overhead of the flexible approach is higher than that of the static approach because the Spring 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.

3.2.2 Proportional and Individual Allocation

For both static and flexible allocation schemes, there are also two ways to assign each multimedia task instance to the multimedia server instance. We call one of them *proportional allocation* where each task instance is split equally into the multimedia server. Suppose we have 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 $\frac{P_S}{P_i}$ 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 $P1$, the server has the same period as stream 1, namely $P1$. In this example, the length of each server instance is the sum of the task of stream 1, half of stream 2 and one third of stream 3.

Note that in this allocation scheme, some task instances have to be executed before their period if the period of the multimedia stream is not an integer multiple of that of the server. This can be

shown with a simple example. (Figure 2)

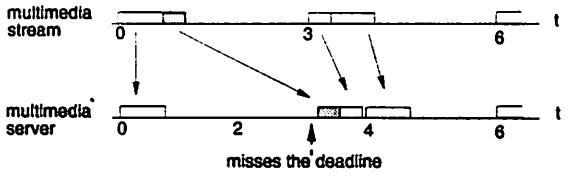

Figure 3 : Missing deadline in the flexible proportional allocation

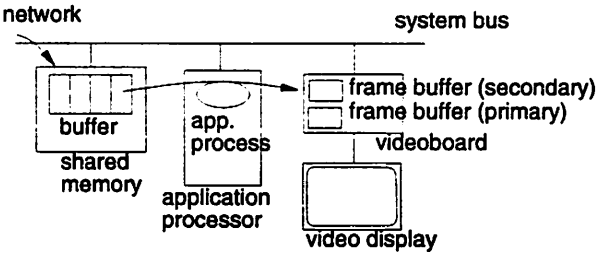

Figure 4 : An example application system

In Figure 2, the second task instance in the multimedia stream has to be started in the second server instance. Note also that since proportional allocation is a statistical scheme, it is possible that a task's deadline can be violated. This is shown in Figure 3. If a designer wanted to avoid this possibility, then the individual allocation scheme, described below, should be used.

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, 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, we can tolerate the jitter caused by the execution delay.

For example, in the architecture we are considering here, a typical type of processing of the multimedia task is to take frame data out of the buffer, process it and put it into the secondary frame buffer on the videoboard (Figure 4). 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 contents [13]. 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 approach *an individual allocation*. 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 5, 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 which the task is assigned has to be located between the task's release time and deadline. If such a server instance can not be found, a new server instance has to be created. The order with which tasks

are executed inside the server can be decided by the earliest deadline first manner. Each server instance has to keep the information on which tasks it is responsible for and in what order it has to execute them.

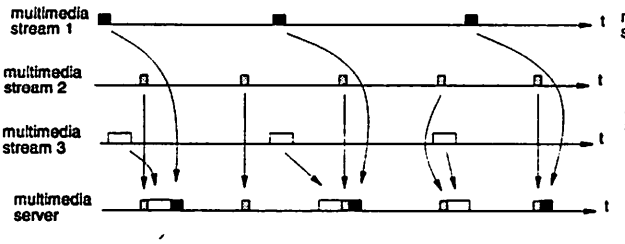

Figure 5 : Individual allocation of multimedia streams

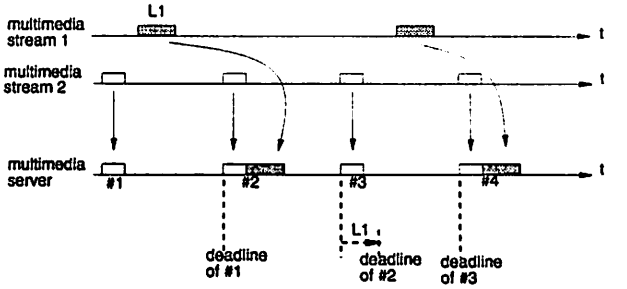

Figure 6 : Deadlines of the flexible individual allocation

As opposed to the proportional allocation scheme, the individual allocation 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 of 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 6), 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 a 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 #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. In this way, the multimedia server with the flexible individual allocation can provide the deterministic guarantee by keeping identification information of task instances.

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 is the same as those of the incoming stream. When the scheduler creates a server, it has to create it up until the latest deadline of the existing hard real-time tasks. 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). 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 cpu time outside the multimedia server using the Spring scheduling algorithm. 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 with the Spring algorithm. 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 just attempts to schedule the task with the current task set plus the new task. Otherwise, the scheduler needs to create more multimedia server instances to the point of the deadline of the incoming task. After the creation of the server, 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 granted multimedia streams or real-time tasks are always guaranteed to be executed no matter how many tasks will arrive later. Of course, a different approach is possible here. If hard real-time tasks have higher priority over multimedia streams, we can make the existing multimedia streams back off so that the retrial of the schedule is more likely to succeed and the incoming hard real-time task is guaranteed. We will discuss this issue in the next section.

Before actually running the Spring scheduling algorithm, making an 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 for the scheduler to create a feasible schedule. With this testing, 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 a percentage of the cpu time multimedia tasks use in a scheduling time period l and that hard real-time tasks use. Now let's call the percentages *a multimedia server ratio* and *a hard real-time task ratio* and denote them Rs and Rr. 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 20msec of server computation time and 100msec 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 time of server instances divided by the scheduling period. The hard real-time task ratio Rr is also sum of the execution time of hard real-time tasks divided by the scheduling period. In order for the schedule to be successful, $Rw + Rr$ have to be at least less than 100% $\times$ (the number of processors). If $Rw + Rr$ is greater than 100% $\times$ (the number of processors), the incoming request is immediately rejected or the following degradation approach will be taken, depending on the policy in effect at that time.

4 Degradation of QOS

In the Spring scheduler with a multimedia server, 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 the degree of degradation is kept 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 Spring 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.

The mechanism we present here which provides a solution for this problem is a multilevel scheduler and works as follows. If the first scheduling attempt fails, the scheduler passes the information on the multimedia server and hard real-time task's 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 7 to converge on a successful 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 recalls the Spring scheduler. Then the Spring scheduler 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 lower Rs, and if successful, the planner chooses it with 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. An example of this scheduling process is summarized in the diagram in Figure 7.

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 task distribution approaches to the server instances are compared to find out which allocation policy is most feasible. Second, different deadline and execution time distributions of hard real-time tasks are given to the scheduler to further evaluate the performance characteristics of the algorithms. Finally, the effectiveness of the multilevel scheduler for degradation of multimedia QOS 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 feasible without multimedia streams, that is, 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.

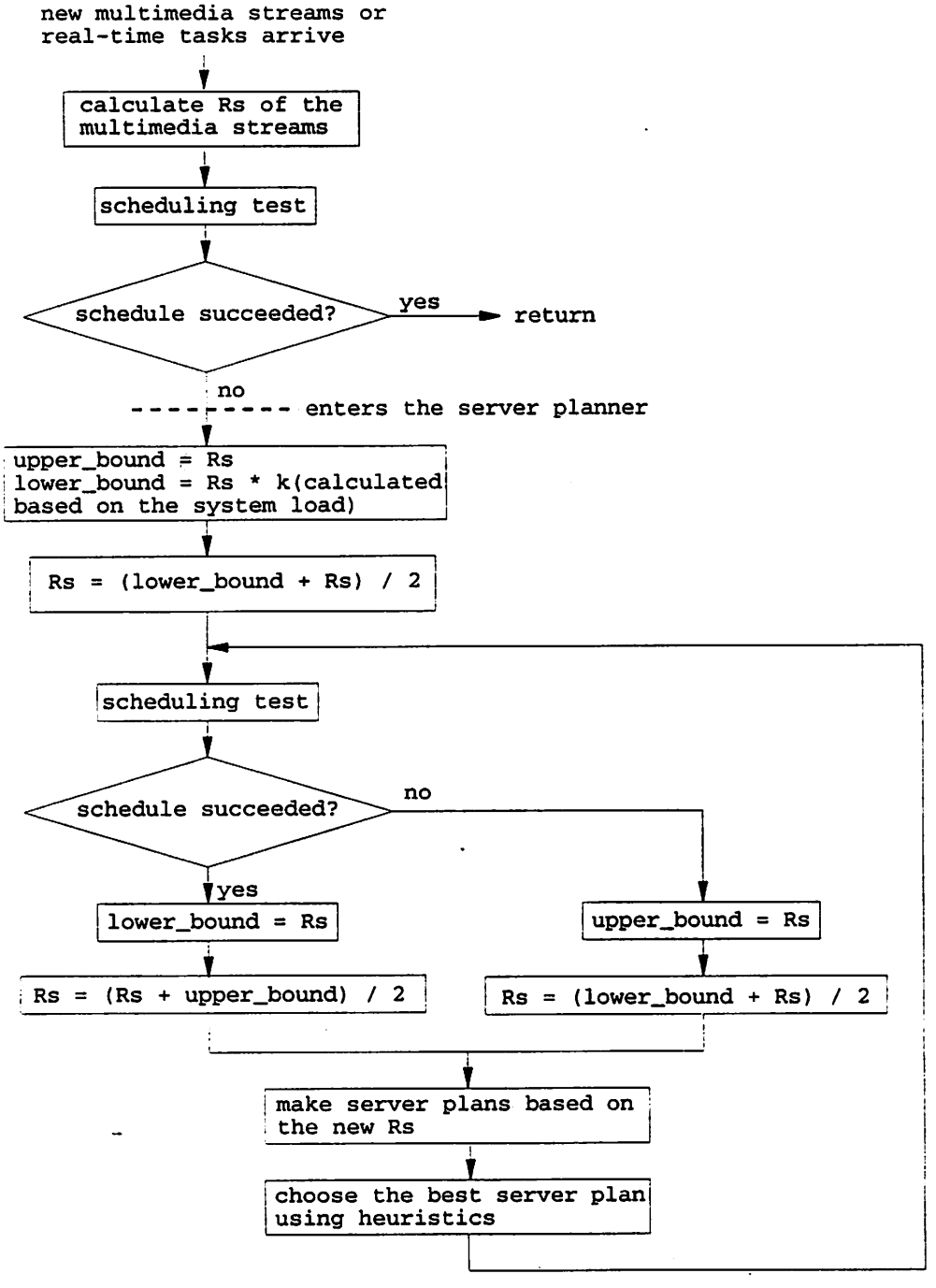

Figure 7 : The iterative server rate adjustment algorithm

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 generated by this task set generator is in the form of 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 used up 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 units of 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. In the original simulation introduced in [7], the deadline of each task in the task set was chosen randomly between the task set's SC and $(1+R)*SC$, where R is a simulation parameter indicating the tightness of the deadlines. However, in the course of the simulation, we found this deadline assignment method was not suitable for the study of the server based algorithm because if there is a large multimedia server instance just before $(1+R)*SC$, it means that the deadlines of all tasks were shortened by that server instance. Therefore, whether the scheduling succeeded or not largely depended on whether a server instance resided near $(1+R)*SC$ rather than the computation time and the period of the server. In order to avoid this effect, a 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) [5]. The file describes all the task information such as timing and resource usage specifications needed by the Spring 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 with characteristics as described in the next subsection.

5.3 Simulation Method

In the simulation, 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

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

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 primary purpose of this simulation is to compare the different server assignment policies and examine effects of changing the parameters, we normalized the simulation time unit into milliseconds and chose realistic values for the parameters so that we can assess the feasibility of our approach to some extent. The schedule length L is 300msec, 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 60msec and 90msec. We put five multimedia streams with different computation time and period 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 33.3msec and that of the second stream is 40.0msec (33.3msec × 1.2). Similarly, the third, fourth and fifth streams have a period of 46.6msec (33.3msec × 1.4), 53.3msec (33.3msec × 1.6), and 59.9msec (33.3msec × 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 time were varied. Each of the five streams consumes the same amount of cpu time on average, that is, if computation time of a task instance in the baseline stream is 5msec, computation time of a task in other streams are 6msec (5msec × 1.2), 7msec (5msec × 1.4), 8msec (5msec × 1.6), and 9msec (5msec × 1.8). If these tasks are allocated to the server proportionally, the computation time of each server instance is 25msec ($5\text{msec} + 6\text{msec}\times\frac{33.3\text{msec}}{33.3\times1.2\text{msec}} + 7\text{msec}\times\frac{33.3\text{msec}}{33.3\times1.4\text{msec}} + 8\text{msec}\times\frac{33.3\text{msec}}{33.3\times1.6\text{msec}} + 9\text{msec}\times\frac{33.3\text{msec}}{33.3\times1.8\text{msec}}$ = 5msec×5). If they are allocated to the server individually, each server instance has different duration.

5.4 Simulation Results

5.4.1 Comparison of the Server Allocation Policies

The simulation results with the different multimedia server assignment policies and no degradation policy are shown in Figure 8. The X axis represents computation time of the baseline multimedia stream as described in the previous section. In all simulation results shown, the success ratio of the scheduling 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 1.2msec, 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 example 2msec, the flexible proportional has 99% of success ratio, but the flexible individual has only 75%. Although these results are as expected, the significant difference between the static allocation and the flexible allocation is noteworthy. Moreover, the difference between the flexible proportional allocation and the flexible individual allocation indicates that the price we have to pay for the deterministic guarantee is quite expensive. This is mainly because the deadlines of the server instances in the flexible individual allocation is sometimes much shorter than those in the proportional one. From

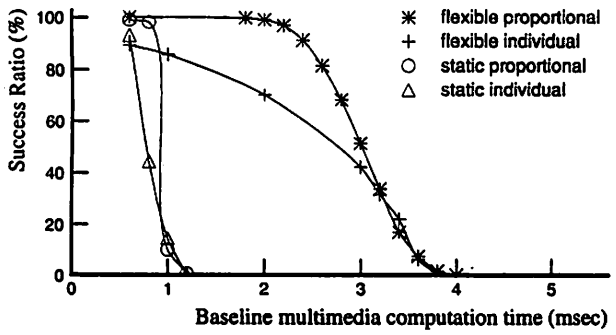

Figure 8 : The server types and the success ratio (period = 33msec)

these results, we can conclude that the flexible proportional assignment policy provides the highest performance in terms of success ratio in the Spring system with a multimedia server.

5.4.2 Effect of Changing Parameters

In the following simulations, only the flexible allocation schemes are examined. Figure 9 shows an effect of changing deadlines of hard real-time tasks on the success ratio. The plain lines are the success ratio when deadlines are chosen between 60msec to 90msec, and the dotted line are those when deadlines are between 30msec to 60msec. 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 60-90msec the success ratio of both proportional and individual allocations drop to 0% when the baseline computation time is 4msec, whereas with the deadline ranges between 30-60msec, they drop to 0% when the baseline computation time is only 2.8msec. These results indicate that deadlines of hard real-time tasks largely affect the upper bound of the multimedia server computation time and they are almost proportional.

In Figure 10, results are shown where the execution time of hard real-time tasks is chosen from the different ranges. The dotted lines shows the ranges between 10-20msec, the plain line 10-30msec, and the dashed line 10-40msec. 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 33msec and its computation time is 10msec, the laxity of the server is only 23msec. When the maximum size of the hard real-time tasks is 40msec (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 10msec (it corresponds to the baseline computation time of 2msec). These simulation results have shown the sensitivity of the success ratio to the size of hard real-time tasks.

5.4.3 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 schedule attempt, that is, when the $n-th$ scheduling attempt is successful, Rs for the $(n+1)-th$ attempt is increased

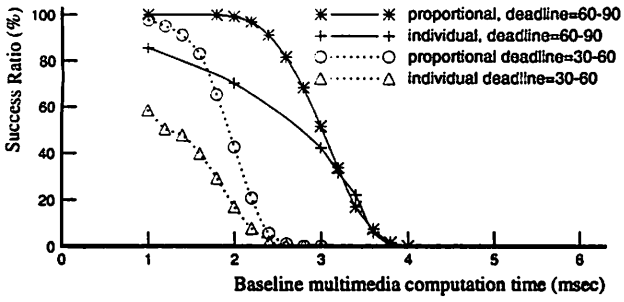

Figure 9 : Effect of deadline on success ratio

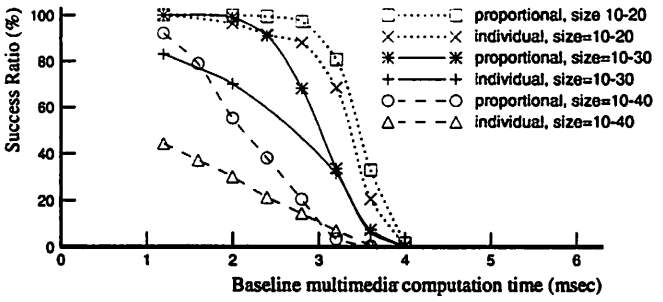

Figure 10 : Effect of hard real-time task size on success ratio

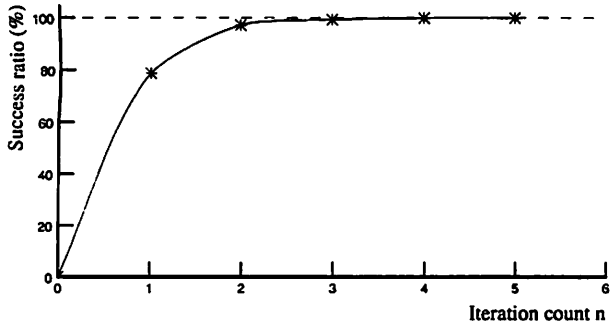

Figure 11 : Iteration count and the success ratio

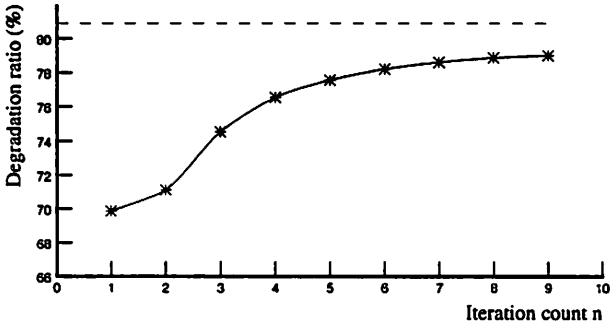

Figure 12 : Iteration count and the degradation ratio

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 11 shows the average success ratio over 500 task sets × 6 simulation runs achieved within n attempts for each task set. As we can see in Figure 11, 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 12, the Y axis is the degradation ratio r which is defined as below and which indicates the degree of degradation from the initial requirement.

$$r = \frac{\text{degraded Rs with which the scheduler tries the next schedule}}{\text{initial Rs (application requirement)}}$$

(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 12 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 12 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 higher degradation ratio, but it will require the larger number of iterations.

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

6 Conclusion

In this paper, we presented a rationale for requiring an integrated platform for 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 based on the server based Spring scheduler. There are four possible multimedia task assignment policies to the server 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 Prof. Krithi Ramamritham for his helpful comments. This paper has also benefited from many discussions with Subhabrata Sen. Thanks also go to Gary Wallace for his help with the implementation of this algorithm into the Spring simulator. We also acknowledge the support of Mitsubishi Electric Corporation and NSF under grant IRI-9208920.

References

[1] Aloke Guha, Allalaghatta Pavan, Jonathan Liu, Ajay Rastogi, Todd 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.

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

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

[4] Daniel I. Katcher, Kevin A. Kettler, and Jay 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.

[5] Douglas Niehaus, John A. Stankovic, Krithi Ramamritham, The Spring System Description Language, UMASS CS TR 93-01, January 1991.

[6] Douglas Niehaus, John A. Stankovic, Krithi Ramamritham, A Real-Time Systems Description Language, *IEEE Real-Time Technology and Applications Symposium*, May 1995.

[7] Krithi Ramamritham, John A. Stankovic, Perng-Fei 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.

[8] John A. Stankovic, Krithi Ramamritham, The Spring Kernel: A New Paradigm For Real-Time Systems, *IEEE Software*, May 1991, pp. 62-72.

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

[10] Yukinobu Taniguchi, Akihito Akutsu, Yoshinobu Tonomura and Hiroshi Hamada, An Intuitive and Efficient Access Interface to Real-Time Incoming Video Based on Automatic Indexing, *Proceedings of ACM Multimedia*, 1995, pp. 25-33.

[11] David L. Tennenhouse, Joel Adam, David Carver, Henry Houh, Michael Ismert, Chiristopher Lindblad, Bill Stasior, David Weatherall, David Bacher and Theresa Chang, A Software-Oriented Approach to the Design of Media Processing Environments, *Proceedings of the International Conference on Multimedia Computing and Systems*, pp. 435-444, Boston, MA, May 1994.

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

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

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