, , 1–22 ()

Rapid Prototyping with Constraints-based Scheduling for Multimedia Applications

WYNNE HSU AND TEIK GUAN TAN whsu@iscs.nus.sg and tanteikg@iscs.nus.sg
Department of Information Systems and Computer Science
National University of Singapore
Lower Kent Ridge Road
Singapore 119260

Abstract. With the increased demand in multimedia applications, the need to provide better system support is greater than ever. Multimedia applications have an added dimension of time in their execution which results in stringent timing requirements. Existing systems incorporate such stringent timing requirements either at the system-level or the application-level. System-level supports are typically operating-system-dependent whereas application-level supports are achieved by building timing controls into the application itself. This lengthens the application development time and fails to take full advantage of operating system's capabilities. In this paper, we propose a framework that resides between the system-level and application-level support. The framework consists of two layers: an interface layer that incorporates high-level end-to-end timing constraints, and a system layer that implements a host-end scheduling mechanism to support high-level end-to-end timing specifications. Two applications have been developed using this framework. The results indicate that the framework is able to support rapid-prototyping of multimedia applications with stringent timing requirements.

Keywords: soft real-time scheduling, multimedia timing constraints, rapid prototyping

1. INTRODUCTION

Multimedia applications place different demands on computer systems with the added dimension of time in their execution. For instance, the playback of a video sequence has to follow a pre-established frame rate of display to achieve an optimal presentation. Adhering to the timing constraint in the frame rate is essential so that the viewers do not experience too much jitters or unnatural 'jerkiness' in the display. There are a number of ways to incorporate timing constraints into multimedia applications. These can be partitioned into system-level support or application-level support. Under system-level support, the operating system (and perhaps the hardware itself) may consist of a micro-kernel, supporting only the most essential system functions, and timely execution is achieved through the use of threads and intelligent scheduling mechanisms. An example of such an operating system is Real-Time Mach [6] by CMU. Applications written in this manner are specific to a particular operating system and are often not portable across different platforms. On the other hand, application-level support means that the application does not rely on the operating system to achieve time-regulated execution. Whatever timing controls needed by the application are built into the application itself, like Adobe Premiere. While such approach is inherently more portable, they are unable to give a guarantee on the real-time requirements demanded by the applications. For hard real-time application, this is not an acceptable option. However, for most multimedia applications, where timing requirement is a soft constraint, such a scenario is a good compromise between portability and a satisfactory level of timing control.

Following this line of reasoning, many commercial operating system developers begin to incorporate some form of real-time support into existing UNIX systems. These include popular operating systems like Solaris, IRIX and LINUX. The support provided is usually in the form of the ability to assign real-time (non-degrading) priorities to processes which will take precedence over all other processes. It also allows the user to specify a guaranteed amount of CPU time to be allocated to the process over a fixed period. SRV4 is an operating system that supports this concept of real-time threads. However, in [13], multimedia experiments performed on the SVR4 did not yield the desired results as directly applying hard real-time algorithm does not cater fully to the execution needs of multimedia applications. Instead of applying hard real-time algorithm directly, we introduce a flexible constraints-based scheduling framework that supports the soft real-time requirements of multimedia applications, while making use of the modern UNIX operating systems. We propose a framework that resides between the operating systems and the applications. It aims to: (1) provide a set of generic timing elements which can be reused across most, if not all, applications, (2) implement a scheduling system that takes as inputs the multimedia timing constraints and performs a best effort scheduling, and (3) maintain the portability associated with the application-level support, and yet at the same time, fully exploit the optimization and real-time features of specific operating systems.

The framework consists of two layers (see Figure 1). The interface layer defines

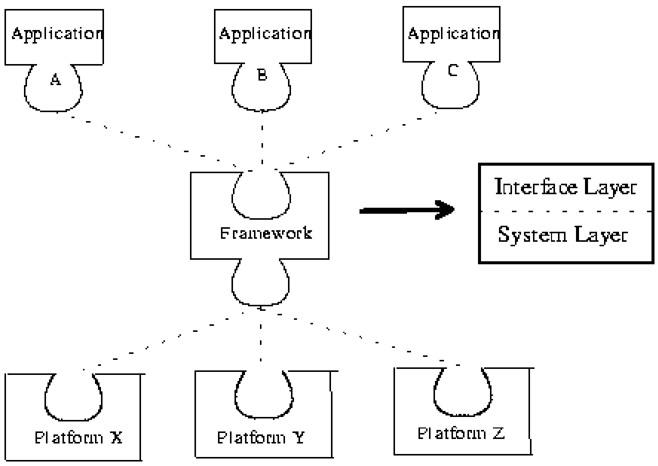

Figure 1. The Proposed Framework

a model for users to indicate the high-level end-to-end timing constraints of the various modules within the model. The system layer implements the best effort scheduling system. We have implemented the framework on a SGI Indy workstation running on IRIX 5.3. The rest of the paper is organized as follows. Section 2 highlights some related work. Section 3 describes the interface layer. Section 4 introduces the overall architecture of the system layer and a brief overview of the implementation details in the system. Two test applications have been built and the results of execution are shown in Section 5. Section 6 concludes this paper.

2. RELATED WORK

Multimedia support environments have been traditionally built on real-time kernels. Such environments include the QoS architecture by [3] which extends the Chorus operating system by including QoS configurability, connection-oriented communication and real-time threads. [6] describes a scheduling mechanism for Real-Time Mach through processor reservation. The CPU allocation is achieved through a QOS manager which can interact with the user. The use of the real-time micro-kernel in such systems, albeit workable, generally makes the applications not portable. We feel that absolute guarantee in execution is not a necessity for multimedia applications. Instead, we have adopted to use the basic time-regulated support in the modern UNIX operating systems, thus achieving greater portability in the process.

We also wish to provide features in the framework to enhance the development process of the applications. These include concepts of media scheduling, data-passing, media synchronization, etc. An example of a fully-integrated support system is the VuSystem [8]. The VuSystem is a multimedia programming environment developed in MIT. It combines the programming techniques of visualization systems and the temporal sensitivity of multimedia systems to process/manipulate digital media. It adopts a media-flow architecture which splits the application into two partitions, namely in-band code and out-of-band code (see Figure 2). In-

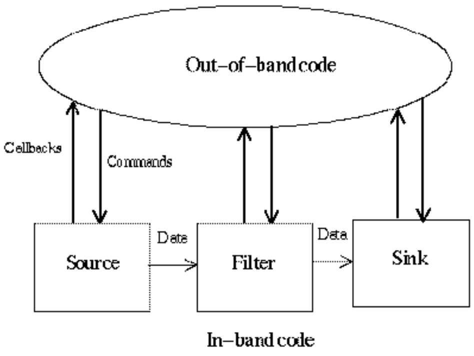

Figure 2. Design Approach of the VuSystem

band code is a small fragment of the application that is performed for every frame, processing continuously on a video or audio stream. They are differentiated into Source, for modules which produce data, Sink, for modules which consume data, and Filter, for modules which perform intermediate processing on the data. The execution follows a re-configurable directed graph of modules. The out-of-band code is event-driven and contains the GUI (Graphical User Interface) and the bulk of functionality that can be applied to the payloads (media data). Each payload is dynamically-typed and time-stamped for scheduling and synchronization purposes. Module dispatching by the scheduler depends largely on the user-mode application dispatcher provided by Xt. This, however, limits the possibility of using different scheduling algorithms to schedule each of the tasks.

Another platform which combines architectural support with OS scheduling mechanisms is the multimedia server on the Spring real-time system [5]. The Spring system incorporates a planning-based scheduler for distributed computing. Citing some differences in requirements between hard real-time versus multimedia tasks, CPU time is divided into two parts, one for multimedia tasks and the other for hard real-time tasks. A multimedia server is responsible for managing a number of multimedia streams through allocation schemes which attempt to multiplex all the streams within the given CPU time. Each multimedia stream is viewed simply as some periodic demand for CPU time and not much structure and control is provided. Under overload conditions, [5] describes an adjustment algorithm that iteratively degrades the QOS of the multimedia server to accommodate the hard real-time tasks, although there is no mention of how each multimedia stream under the server will react to the change. These and some of the other issues will be investigated in our proposed framework.

3. The Interface Layer

The proposed framework is partitioned into 2 layers: the interface layer and the system layer (see Figure 1). The objective of the interface layer is to provide an environment that allows the user to specify complex timing relationships as naturally as possible. To this end, we have identified specific timing elements that are lacking in the current operating systems:

- Temporal intervals specification. This refers to the specification of relative times of execution for the tasks. Example of temporal interval is "Task A must begin execution before Task B." It has been shown that temporal intervals can be represented by 13 binary relationships [1]. These relationships fall into the categories of "overlaps", "before", "during", "meets", "starts", "finishes", and "equals".

- Synchronization. This refers to the ability to specify both inter-media and intra-media synchronization.

- End-to-End Time Duration. This is an area that is most lacking in terms of research and system supports. In many situations, the user are not concerned with the exact time duration for each task. Instead, it is the end-to-end duration of a sequence of tasks (for example, manipulating the data from retrieval to presentation) that is important.

Thus our proposed model for the timing requirements must take all these factors into account. Among the existing specification models for multimedia applications, we find that Petri net models are good for the event-driven and synchronization nature of multimedia applications. Ordinary Petri Nets [2] can be defined as a 3-tuple (P, T, F) where: P is a non-empty set of places, each denoted by a circle. T is a non-empty set of transitions, each denoted by a vertical line. F is a non-empty set of arcs, each denoted by an arrow from a place to transition or vice versa. Many different extensions to Petri Nets were proposed to include additional timing features so as to fulfill the presentation requirements. These extensions include Object Component Petri Nets [10] (OCPN), and Time Stream Petri Nets [4] (TSPN). Figure 3 shows some examples of how the TSPN can model temporal instants and temporal intervals timing elements. However, they are unable to handle dynamic inter-media synchronization, as well as to allow for the specification of end-to-end timing constraints.

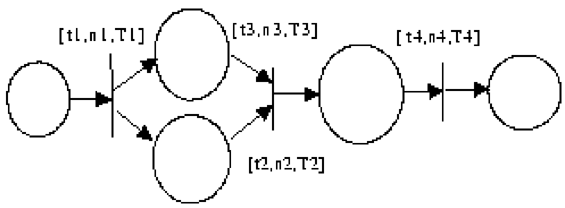

Figure 3. Example of Time Stream Petri Nets

The use of Petri Nets inherently supports the notion of inter-media synchronization. The use of the transition will force all incoming arcs to be synchronized before firing off the outgoing arcs. However, the characteristics of multimedia streams call for a stronger property in inter-media synchronization. For example, there are two media streams, one audio and one video stream. The video stream is to be synchronized with the audio stream (the primary stream) during a real-time play-out. If the video stream is inadvertently delayed during transmission, the audio stream should not wait for the arrival of the video and instead should be played immediately (within a tolerance level of jitters, delays, etc.). This will ensure that the audio playback is maintained at a smooth, uninterrupted pace, although the video may skip a frame. Therefore, the arc representing data coming in from the audio task should be differentiated as it will force the transition to fire without waiting for the other late media streams to arrive. The use of this property can thus allow all other media streams to be synchronized with respect to a primary stream. We label this arc as a *Non-strict Firing* arc. This non-strict firing arc (depicted by a square preceding the arc) is shown in Figure 4. As for intra-media synchronization, each media stream has to be synchronized with respect to time. This is achieved by

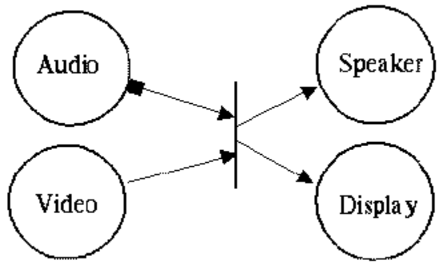

Figure 4. An Example of Non-Strict Firing Arc

providing timing routines for the programmer to incorporate in each media stream. These timing routines will essentially "wake up" the media stream to spawn a new thread at regular time intervals.

To support the timing constraints within each media stream, we introduce the concept of *Constraint Ties.* The constraint tie is a construct that binds a timing constraint between two outgoing arcs (see Figure 5). The constraint tie differs

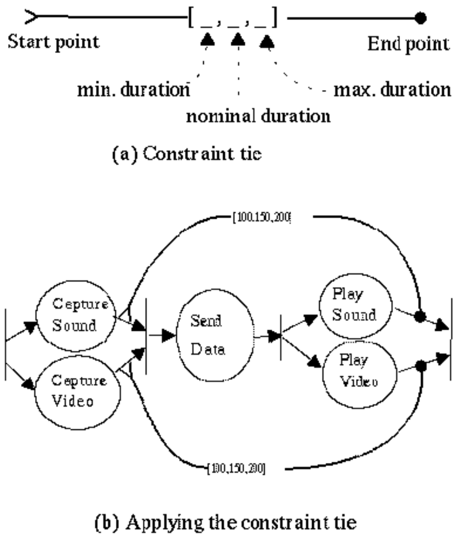

Figure 5. An Example of Using Constraint Tie

from OCPN and TSPN in that the user does not need to meticulously draw up the deadlines for every task (which is not very intuitive) but instead can simply specify the desired end-to-end duration across multiple tasks. Multiple deadlines across a task is also supported. The implementation of constraint ties will be described further in Section 4.

Formally, the newly extended petri nets model is defined as a 5-tuple (P, T, F, N, C) where P, T and F are as defined in the original petri nets model, N is the set of non-strict firing arcs, and C is the set of 2-tuple defining the constraints from one arc to another.

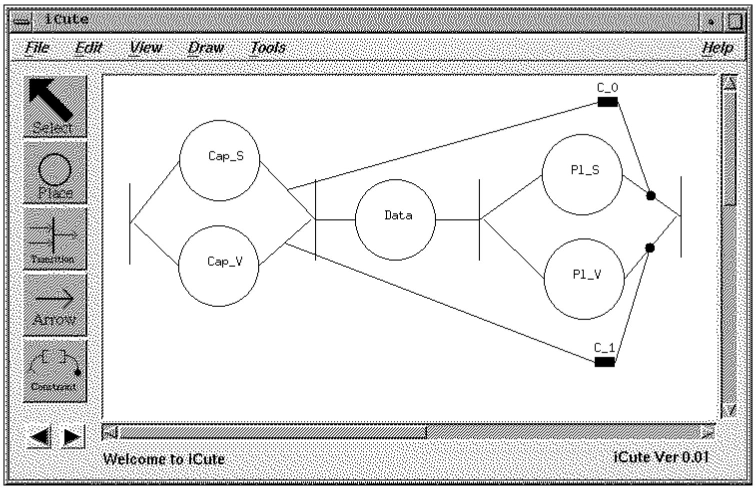

Figure 6. The Visual Programming Interface Screen Design

To support the proposed extended Petri net model, we implement a visual programming interface. Figure 6 shows the graphical editor that the user interacts with. The user designs his/her application using the editor. Properties associated with the places, transitions, arcs and constraint ties can also be specified easily through the interface (see Figure 7). Once the user is satisfied with the design, the model is passed through a verification process to check for discrepancies.

A translation mechanism has been implemented that takes the Petri net model as input and outputs an encapsulated program that incorporates suitable calls to the API library provided by the framework. This encapsulated program is then presented to the scheduling system for run-time execution. Figure 8 shows an example of the encapsulated program corresponding to the model of Figure 5. There are three essential API calls: "Open" is to start execution and to accept input data, "Return" is to output data, and "Close" is to end the execution. The need for "Return" and "Close" is to decouple the output of data and end of execution so that *Operations* can return more than one set of data. The encapsulated program is compiled and linked with the API library to produce the executable for the application.

Underlying the visual programming interface is an object-oriented model that matches closely with the Petri Net specification model. The user program is an *Application*. Each *Application* object is composed of a number of *Operations*

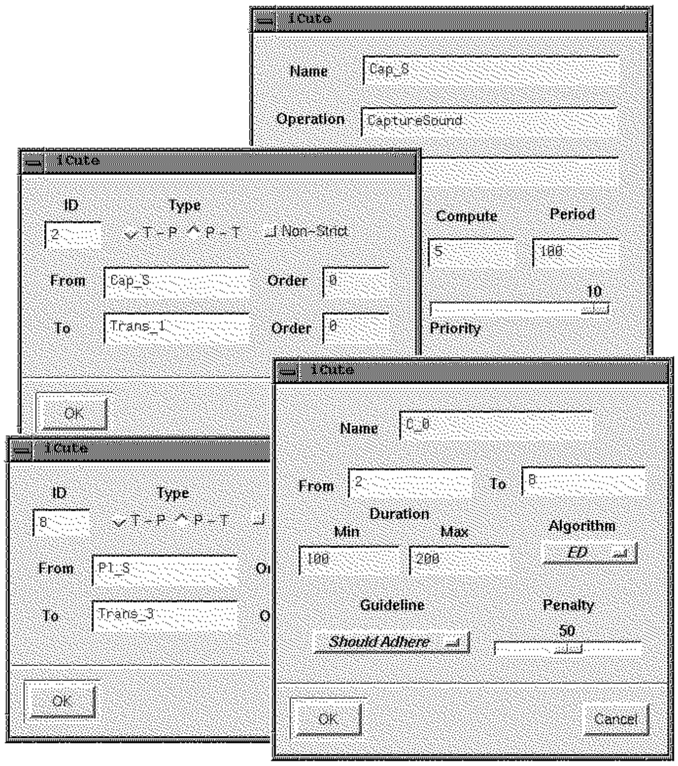

Figure 7. Specifying the properties of the nodes

```
#include <cute.h>
#include "UserRoutines.h"

void CaptureSound(void * InData) {
  char * SoundBuffer;
  Open(0,InData);
  SoundBuffer = UserCaptureSound();
  Return(0,InData,SoundBuffer);
  Close(InData);    }
void CaptureVideo(void * InData) {
  char * VideoBuffer;
  Open(0,InData);
  VideoBuffer = UserCaptureVideo();
  Return(0,InData,VideoBuffer);
  Close(InData);    }
void SendData(void * InData) {
  char * SoundBuffer,*VideoBuffer;
  char * ReceivedData;
  Open(2,InData,&SoundBuffer,&VideoBuffer);
  UserSendData(SoundBuffer,VideoBuffer);
  ReceivedData = UserReceiveData();
  Return(0,InData,ReceivedData);
  Close(InData);    }
void PlaySound(void * InData) {
  char * Buffer;
  Open(1,InData,&Buffer);
  UserPlaySound(Buffer);
  Close(InData);    }
void PlayVideo(void * InData) {
  char * Buffer;
  Open(1,InData,&Buffer);
  UserPlayVideo(Buffer);
  Close(InData);    }

int main(int argc, char *argv[]) {
  char *PetriModel = "Video.Cute";
  if (argc > 1)
    PetriModel argv[1];
  return CTreadFile(PetriModel);
 }
```

Figure 8. The Encapsulated Program Translated from Figure 5

and *Mailboxes* which correspond to the places and transitions in the Petri net model (Figure 9). Each *Operation* is an executing entity running on a thread. Data returned from an *Operation* is sent to the *Mailbox*. The *Mailbox* can be regarded as a logical address that provides the parameter-passing mechanism between *Operations*. Once the *Mailbox* is filled, the respective *Operations* are triggered. Timing requirements are specified by binding some *Operations* to the user-specified timing constraints (see Figure 9). The mapping of the model onto the underlying

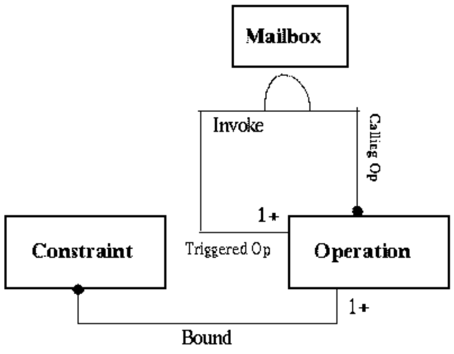

Figure 9. Application object structure

operating system is as follows: every *Operation* is a thread which takes in data, processes and outputs new data. Data is collected at a storage location in the *Mailbox*. Once filled, the *Mailbox* will cause the subsequent threads to be spawned. The assignment of priority to the threads is determined by the constraint ties associated with the *Operations*.

4. The System Layer

Having defined the interface layer of the framework, we next describe the system layer. There are three components in the system layer: (1) a host-level scheduler to ensure fairness between independent user programs, (2) a constraint-based scheduler to handle the processing of user-defined timing constraints within each user program, and (3) some additional implementation supports (such as optimization techniques) to make the framework practical and useful.

4.1. Overview of the System Layer

An overview of the system layer's architecture is shown in Figure 10. The input is the *Application* which is an independent user program that may require the use of physical devices such as camera or microphone etc. Since the devices are shared among different *Applications*, there is a need to ensure efficient utilization of the devices and the prevention of deadlock. This is the function of a *ResourceManager*. Furthermore, to coordinate among the various *ResourceManager* and *Applications*, we need an *Arbitrator*. This *Arbitrator* is implemented as a system-level process

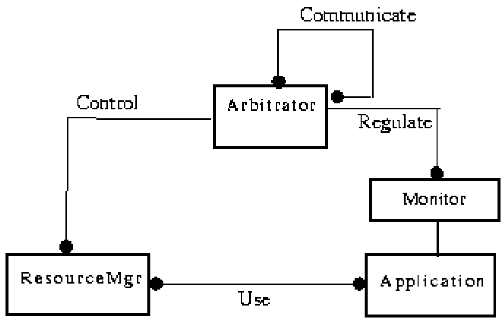

Figure 10. Overview of the System Layer

daemon. Its responsibilities include:

- To regulate the CPU usage of each *Application*, ensuring a maximal throughput for the computer system.
- To control the *ResourceManagers* in the system, resolving contention by *Applications* (both local and remote) to the devices.
- To handle all communications with other *Arbitrators* on remote systems across the network. This is to make the network transparent to the *Applications*.

The *Arbitrator* interacts with the *Application* through the *Monitor* object. The *Monitor* is a separate thread that executes independently from the *Operations* and collects status information from the execution of the *Application*. If the *Monitor* detects that the *Application* is not receiving enough service due to missed deadlines, it will initiate a quality of service negotiation with the *Arbitrator*.

4.2. Inter-application Scheduling

Inter-application scheduling deals with the management of CPU between different users whereas the management of CPU among tasks of a single user is termed intra-application. We differentiate between inter-application scheduling and intra-application scheduling because, typically, there is little or no dependency among tasks of different users, and thus a simpler scheduling algorithm can be applied. However, when it comes to scheduling of tasks within an application, a more complex scheduling algorithm is needed to take into considerations the complex timing requirements among inter-related tasks. Hence, in order to achieve greate flexibility and scheduling efficiency, we have decoupled the two. Inter-application scheduling is done through the system-wide *Arbitrator*. Every application has its own *Monitor* which will communicate with the *Arbitrator*, and negotiate for a CPU allocation quota. This quota is quantified by a period, and an amount of CPU time within that period. The *Arbitrator* ensures that the total sum of all the allocations is within what the system can support by sourcing the system load regularly. Once the allocation quota is negotiated, the *Arbitrator* will, in turn, make a request to the operating system for this quantum. (On IRIX 5.3, this request is through the system call "schedctl()").

During execution, the *Monitor* constantly update the execution history of the application and feedback to the *Arbitrator* on the execution status. There are three possible execution status:

- Lack. The application is facing some missed deadlines and will perform better if more time slice is given.
- OK. The *Arbitrator* should not change the time-slice if possible.
- Excess. The *Arbitrator* can decrease the time-slice of the the application if the need arises.

The *Arbitrator* will then, based on the negotiated execution level and feedback status, adjust the appropriate allocation for that application.

The *Arbitrator* may also refuse to admit an application should it find that the requested service level cannot be fulfilled due to a high system load. The objective is to prevent the system from becoming overloaded by limiting the number of entering processes. In this way, execution cycles are not wasted on those processes whose deadlines cannot be met, and processes already in the system are not deprived of the needed execution cycles. The loading of the system can be established by sourcing the process execution image of the system (for UNIX) [14]. The *Monitor* is then responsible for terminating the application, should admission to the system is disallowed by the *Arbitrator*. The *Arbitrator*, however, does not perform eviction for processes already in the system. The management of memory, disk or network among different users is beyond the scope of this work.

4.3. Intra-Application Scheduling

At the intra-application scheduling level, the objective is to derive a schedule such that the more crucial threads are scheduled first. This feature is particular useful if the host system is temporarily overloaded and may not be able to process all the threads within the specified deadlines. In such situations, it makes sense to sacrifice the less important threads so that the more important threads can be completed within deadlines. To achieve this objective, we propose a constraint-based scheduling algorithm. The algorithm proceeds as follows: (1) the high-level end-to-end timing constraints that span over a number of tasks are first broken down into simpler constraints that span over at most one task, (2) if a task has more than one constraints associated with it, a conflict resolution process is initiated. (3) for tasks with no constraints associated with them, they are scheduled on a first-come-first-serve basis.

To break down the end-to-end constraint into simpler constraints, we need the execution information of each task. This information is obtained with the help of the *Monitor*. The *Monitor* keeps track of the execution of the task. As the task is executed, the *Monitor* updates a global table D where D_i represents the execution time of $task_i$. This table is then used as the estimate for the next execution of $task_i$. The breakdown of the constraints can be divided into three cases: the serial

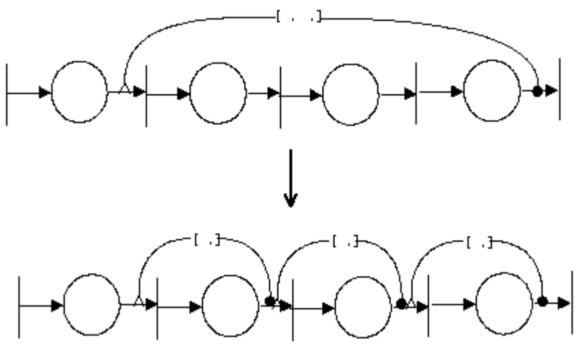

Figure 11. Decomposition of Serial Constraints

case in Figure 11, the parallel case in Figure 12 and the multiple constraint case in Figure 13. Once the end-to-end constraint is broken down into singular constraints, we can apply standard scheduling algorithms to compute the actual priority and deadline of the thread when it is spawned.

Currently, our framework supports the Earliest Arrival (EA), Earliest Deadline (ED), and Least Slack (LS) scheduling algorithms [11, 12]. Assuming a constraint tie with an end-to-end duration, T, the computation of the deadlines and priorities[1] are as follows:

- Earliest Arrival:

$$\begin{aligned} Deadline, d_i &= \text{Current Time} + T - \sum_{k=1}^{i-1} D_k \\ Priority, p_i &= \text{Current Time} \end{aligned}$$

- Earliest Deadline:

$$\begin{aligned} Deadline, d_i &= \text{Current Time} + T - \sum_{k=1}^{i-1} D_k \\ Priority, p_i &= T - \sum_{k=1}^{i-1} D_k \end{aligned}$$

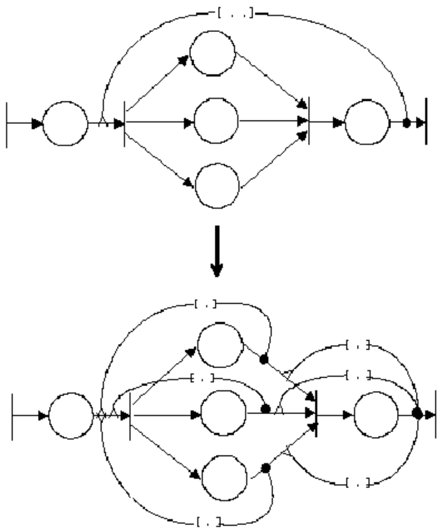

Figure 12. Decomposition of Parallel Constraints

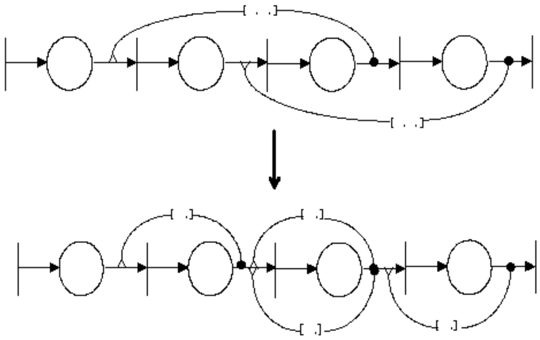

Figure 13. Decomposition of Multiple Constraints

- Least Slack:

$$
\begin{aligned}
Deadline, d_i &= \text{Current Time} + T - \sum_{k=1}^{i-1} D_k - \sum_{k=i+1}^{n} D_k \\
Priority, p_i &= T - \sum_{k=1}^{n} D_k
\end{aligned}
$$

Other scheduling algorithms can be incorporated into the framework easily.

Three questions need to be answered at this point: (1) "What is the overhead involved in breaking down an end-to-end constraint?" To answer this question, we conduct an experiment to measure the overhead involved per timing constraint specified. Details can be found in Section 5. (2) "Will the break-down of end-to-end constraints lead to an exponential growth in the number of constraints thus eliminating any advantages to be gained by constraints specification?" We want to point out that the break down of the constraints is performed once during the initial invocation of the application. For subsequent execution, each process maintains a queue of its relevant constraints (which usually is less than 5 per process). In other words, the break down of constraints, in practice, will not affect the execution time of individual threads drastically. (3) "What happens when one thread/process has two or more constraints associated with it?" When a task has two or more constraints associated with it, it is scheduled according to the most critical constraint. The determination of the most critical constraint proceeds as follows:

1. whichever constraint gives the earliest deadline, then that constraint is the most critical to the task,
2. if there exists more than one constraints with the same deadline, then the constraints that incur the largest penalty is the most critical constraint. The penalty value is a user-specified quantity.

4.4. Additional Implementation Supports

Recall that an *Application* is made up of a number of *Operations*. Each *Operation* is a thread which takes in data, processes the data, and outputs new data. In our framework, this threading is based on light-weight, non-preemptive user threads. Each *Operation* or user task, represented by a place in the Petri-Nets model, is allowed to execute to completion (unless a sleep or semaphore wait request is done by the user) before a new task is spawned. Hence, context-switching between tasks can only happen at the transitions. This implies that the amount of context-switching is controlled, and many concurrency control issues can be avoided. The disadvantage of using a light-weight, non-prememptive user threads is that some tasks may be inadvertently delayed during execution. However, we feel this is tolerable since multimedia applications do not have hard real-time requirements and occasional late jitters are acceptable. With this in mind, we next discuss the issues of deadline management, parameters binding supports, shadow parameter passing, and imprecise computation technique.

4.4.1. Deadline Management To improve usability, the framework must allow a user to specify what happens when a task misses its deadline. In a soft real-time application such as multimedia application, missing a deadline is not a fatal event. Instead, when a deadline is missed, the system will perform a number of actions: first, the *Monitor* is informed. This allows the *Monitor* to detect changes in the load of the system by checking the level of accumulated penalty across all tasks. This change in load affects the overall priority of the application as the *Monitor* feedbacks this change to the *Arbitrator*.

At the same time, we have associated three additional attributes with each constraint. They are: the penalty attribute, the guideline attribute, and the violation call-back attribute.

The penalty attribute dictates the penalty value incurred when the deadline of the associated constraint is missed.

The guideline attribute is used to specify what to do with the current task if the task has missed the deadline. The three possible guidelines are:

- Must Adhere. This represents a hard deadline, meaning that if it is missed, the whole application will fail.
- Should Adhere. This guideline is equivalent to the firm real-time system. Processing of that particular task is abandoned but the application carries on executing. This is relevant, for instance, when a video stream is constantly supplying data to be presented. If the current frame does not arrive in time, it is dropped, and the next frame immediately takes its place.
- Try Adhere. When the system detects the missed deadline, the task incurs the penalty but does not stop. This can be used to define a soft deadline for the task.

Finally, the violation call-back attribute allows the user to specify the actions to be taken in the event that a task has missed its deadline. Generally, when a task has missed its deadline, the data frame is dropped by the system if the guideline specified is ShouldAdhere or MustAdhere. Although the system executes the internal mechanisms to handle such exceptions, provision is also given to the user, in the form of call-backs, to maintain the integrity of the application.

4.4.2. Parameters Binding Supports Besides the issue of deadline management, there is also the question of how to perform the binding of parameters. Multimedia data are usually periodic in nature. In other words, for the same task, we can have multiple instances of the data originating from it. It is important to be able to differentiate between the different instances of data. For example, in a video-on-demand application (a simple producer-consumer model), the user may want to pre-fetch some video data before commencing the display. In other words, the retrieve(producer) task would have to output several data frames before any of the data are consumed by the display task. Thus, there are multiple instances of data frames from the display task perspective.

To allow for multiple instances of data from the same task, we introduce the use of *PigeonHole*. Incoming data from tasks will be placed into *PigeonHoles* associated with the *Mailbox*. A *PigeonHole* is simply a storage location, pre-allocated in anticipation of incoming data. The illustration of how the tasks associate with the *Mailbox* and *PigeonHole* is found in Figure 14.

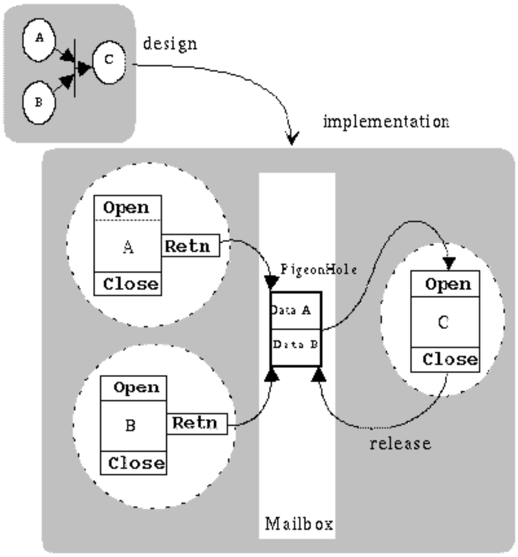

Figure 14. Roles of *Mailbox* and *PigeonHole*

Once the *PigeonHole* is filled by the *Operations*, the *Mailbox* fires the next task. On completion of the task, the *PigeonHole* is then released to be reused. Note that since the threads are non-preemtive, there is no need for the "locking" of *PigeonHole* for concurrency control.

4.4.3. Shadow Parameter passing Parameter passing for multimedia tasks often cause bottlenecks in execution. This is because the amount of data that is copied each time is very large. For example, a NTSC video frame with dimensions 640 x 480 (using 32-bit RGB representation) , will take up over 1 MB of data. If the application is running at 15 frames per second, it will certainly cause too much data movement in-between tasks.

In order to prevent excessive data copying, out-parameters are returned from a task in the form of a pointer to the data. This pointer is subsequently passed to all the ensuing tasks expecting the data. This eliminates the need for data copying in between tasks, but as only the shadow of the actual data is passed, each task should not modify the incoming data.

4.4.4. Imprecise Computation Technique For systems which may face overload conditions, sometimes, a partial execution of several tasks may be more appealing than forcing all tasks to run to completion, missing their deadlines in the process. Situations where partial computations [7] are accepted are:

- Breadth-first search in Artificial Intelligence, where some answers can be obtained quickly although a longer search may yield a more optimal result.
- Display of video images where fuzzy images can give a general picture and production of high resolution images can be done when the system is less loaded.
- Browsing multimedia documents where textual data is first transferred to convey most of the information, skipping the graphical data if additional resources cannot be obtained.

Provided in the toolkit are run-time routines for users to find out the overall execution status, the outcome of the previous task and if the task had already missed the deadline. These inform the users on the current execution status of the system and form the basis for users to decide to use alternative paths of execution.

5. PERFORMANCE AND RAPID PROTOTYPING TESTS

In this section, we test the practical usefulness of the framework in terms of the amount of overhead time introduced by the framework and its ability to support the design and development of real-life multimedia applications. The framework was initially developed on a SGI Personal Indy workstation running on IRIX 5.3. Various demonstration applications were written using the framework. Here, we describe two applications. The first is a live video capture and display application. The objective of this application is to (1) demonstrate that the overhead time involves in maintaining the timing constraints is within the tolerance limits of a soft real-time application, and (2) demonstrate that by using the framework, we are able to achieve better control over the quality of service of our application. In the second application, we attempt to demonstrate how the framework can aid in the design and development of a simple multimedia application.

5.1. The First Application

The first application is a live video manipulation application. It essentially captures video frame from the IndyCam and outputs the capture frame to two display windows. One display window is set with a tighter timing constraint than the other. At the same time, a third display window is opened to perform a texture-mapped of an image onto a non-uniform rational b-spline(NURBS). The NURBS is further manipulated by rotating it along the x and y axis. This NURBS module is adapted from source code provided SGI. With the NURBS module, we simulate the condition of system overload by gradually increasing the size of the NURBS window

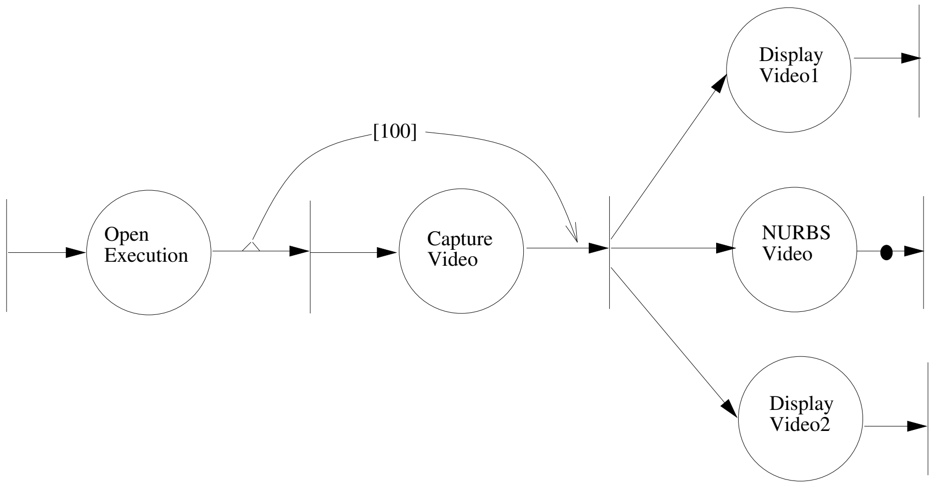

Figure 15. Specification model of Live video manipulation

(implying a larger texture surface to work on). The specification model is shown in Figure 15.

The OpenExecution operation initializes the application to ready it for the video capture. The CaptureVideo task is written to grab and return a video frame. It is set as a periodic thread that will execute at 10 frames per second. The first test we perform on the application is to find out the total amount of overheads the framework adds to the application, as compared to one that does not use the framework. The results indicate that the total overhead to create a thread, perform a context switch to the new thread, and destroy the thread is 150μs. With a constraint declared over the thread, it is an additional 30μs. Such an overhead is quite acceptable for soft real-time tasks, considering that the cost for a fork and exit is about 3ms.

Next, we test the ability of the framework to maintain the required quality of service of a specific thread under various system load. The size of the NURBS is slowly increased while the video display rate of the two windows are being measured. Once again, the result of the execution is compared to one without the framework (see Figure 16). In the graph, the application without the framework faces a degradation of frame rate to approximately 5 fps (from 10 fps) under high loads. The application with the framework is still able to maintain at 10 fps although the display of the NURBS window drops to 3-4 fps. This tradeoff is necessary to ensure that the CaptureVideo maintains at 10 fps.

An area of usage is in computer-participative multimedia [8] systems. A security system can be built to collect video images from different areas and display them at a central console. An added feature in this system is that the computer performs intruder-detection by analyzing the images during video capture and alerts the guards if necessary. The important issue in this system is to maintain the com-

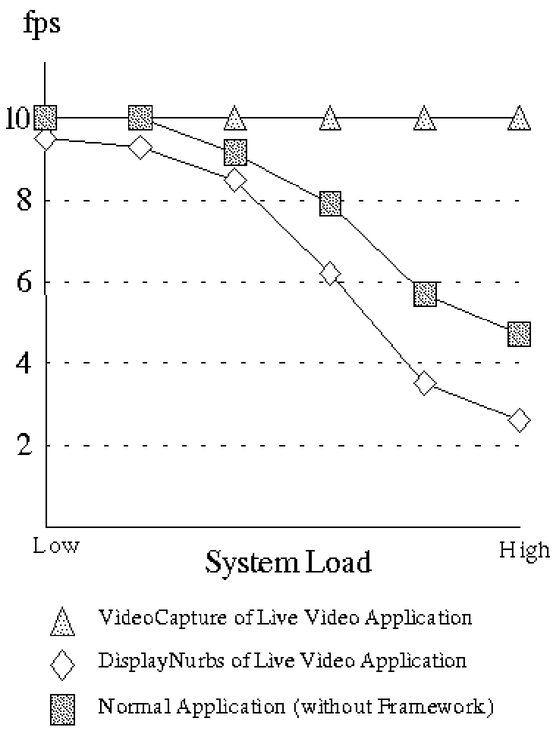

Figure 16. Graph of execution of Live Video under varying loads

puter's response time to capture and detect the images while the display at the console can be compromised.

5.2. The Second Application

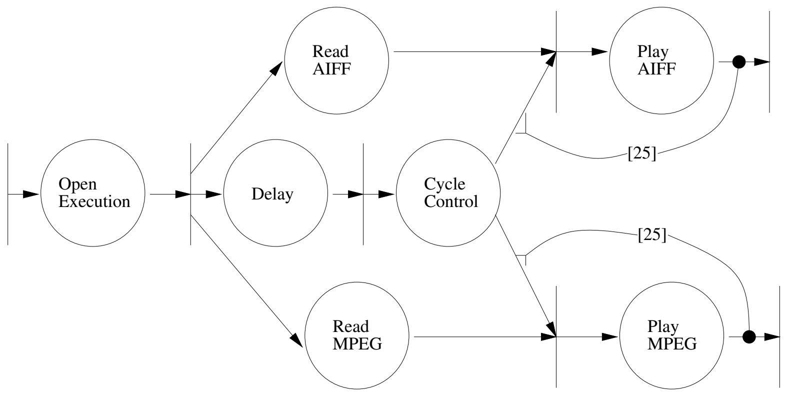

Figure 17. Specification model of audio-video player

An audio-video player is built using the framework as shown in Figure 17. A MPEG-1 (Motion Pictures Expert Group) video stream is synchronized with a AIFF (Audio Interchange File Format) audio stream to playback a movie clip in a rate-controlled manner. Code for the ReadAIFF and PlayAIFF are adapted from source code made available by SGI. Code for ReadMPEG and PlayMPEG are adapted from the Berkeley MPEG decoder,mpeg_decode. Both ReadAIFF and ReadMPEG read and decode data from separate data files. The CycleControl operation provides the timing element to the application by returning at pre-defined intervals so that the presentation will be synchronized. To simulate transmission of real-time video and audio, we modify the application by removing the delay in the CycleControl and by implementing imprecise computation techniques in ReadMPEG to detect and discarding late frames, hence not wasting CPU time to decode them. The development time for this player, using the framework, is approximately 2 days (with debugging). However, the same program requires 4 weeks to complete on a normal UNIX workstation without the benefits of the supports provided by the framework.

6. CONCLUSION

In this paper, we have described a middle-level framework for the design and implementation of multimedia applications. The framework incorporates the ability to allow the specification of end-to-end timing constraints, and to use these constraints to perform scheduling so as to realize the stringent timing requirements of multimedia applications. Two applications have been written. Our experience indicates that the framework can aid in the rapid prototyping of multimedia applications. More work needs to be done in the area of resource managers and QoS negotiations.

Acknowledgments

We would like to thank Dr H. K. Pung and A/P T. S. Chua for their valuable comments throughout the design and implementation of the framework.

Notes

1. lower values represent higher priorities

References

1. J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26:832–843, 1983.
2. G. Bruno. *Model-based Software Engineering.* ISBN 0-412-48670-9, Chapman & Hall, 1995.
3. G. Coulson, G. S. Blair, P. Robin, and D. Shepherd. Extending the Chorus micro-kernel to support continuous media applications. In *Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio And Video*, pages 49–60, November 1993.
4. M. Diaz and P Senac. Time-stream petri nets, a model for multimedia stream synchronization. In *Proceedings of the 1st International Conference on Multimedia Modelling*, pages 257–273, 1993.
5. H. Kaneko and J. Stankovic. A multimedia server on the spring real-time system. Technical Report UMass CS 96-11, Department of Computer Science, University of Massachusetts, Amherst, 1996.
6. C. Lee, R. Rajkumar, and C. Mercer. Experiences with processor reservation and dynamic QOS in real-time mach. *In the proceedings of Multimedia Japan 96*, April 1996.
7. K. J. Lin, S. Natarajan, and J. W. S. Liu. Scheduling real-time, periodic jobs using imprecise results. In *Proc. IEEE Real-Time System Symp.*, 1987.
8. C. J. Lindblad, D. J. Wetherall, and D. L. Tennenhouse. The VuSystem: A programming system for visual processing of digital video. In *Proceedings of ACM Multimedia 94*, October 1994.
9. T. D. C. Little. A framework for synchronous delivery of time-dependent multimedia data. *Multimedia Systems*, 1(2):87–94, 1993.
10. T. D. C. Little and A. Ghafor. Synchronization and storage models for multimedia objects. *IEEE Journal on Selected Areas in Communications*, 8(3):413–427, 1990.
11. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. In *Journal of the ACM*, volume 20, pages 46–61. 1973.
12. A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment.* PhD thesis, MIT Laboratory for Computer Science, 1983.
13. Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerard A. Wall. SVR4UNIX scheduler unacceptable for multimedia applications. In *Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, November 1993.
14. Uresh Vahalia. *UNIX internals : the new frontiers.* ISBN 0-131-01908-2, Prentice Hall, 1996.