Interactive Scheduling Supports for Real-time Multimedia Execution *

Teik Guan Tan and Wynne Hsu
Department of Information Systems and Computer Science
National University of Singapore
Singapore 119260
{tanteikg, whsu}@iscs.nus.sg

1 INTRODUCTION

With the widespread use of multimedia in the industry, there is a need to re-study the underlying computer system structure to better support the execution of multimedia applications. Traditional schedulers, found in the general-purpose operating systems (GPOS) like UNIX, are insufficient as they cannot give sufficient and timely service to the multimedia applications. Though there are many hard real-time systems available [3, 1, 2], they are tailored more for hard real-time tasks where predictability and absolute guanrantee of schedule is the primary concerns. This is not so for multimedia applications where missing a deadline is not fatal. To address the special needs of multimedia scheduling, we have designed and implemented a flexible scheduling testbed, called *ICute* (Interactive exeCUTE). The objectives of *ICute* are: (1) to implement a flexible scheduling mechanism that is able to regulate the execution of multimedia applications to achieve optimal output under various load conditions, (2) to present the programmer with a suitable environment for developing and executing multimedia applications by abstracting all the internal process management and scheduling from the programmer, and (3) to improve portability since *ICute* is implemented on a UNIX platform, the codes are portable across many systems.

2 DEVELOPMENT ENVIRONMENT

An application (specified by a Petri Nets graph) is made up of a series of tasks to be performed. Formally, each Petri Nets graph is a 4-tuple (P,T,A,C) where P represents the set of places. Each place is associated with a particular task to be performed. The types of tasks supported are *periodic* tasks which are automatically invoked at every period interval; *sporadic* tasks which are executed at a regulated rate whenever there is incoming data; and *aperiodic* tasks. T represents the set of transitions. A represents the set of arcs that connect transitions to and from places. A new feature in our Petri Nets model is that the programmer can specify the order of firing. With this explicit control, *ICute* is able to support the 13 binary temporal relationships between two different media intervals. C represents the set of constraint ties. This is a new extension to the existing Petri Nets graph. A constraint tie essentially spans across a number of places and binds two arcs, a start-point and end-point, such that execution of the tasks under the span is constrained within a certain time duration.

Figure 1 shows the development process that a multimedia application undergoes. It begins with the specification of a Petri Nets graph. At the front-end, the programmer designs the Petri Nets graph of the application using an interactive graphical editor. When the programmer is satisfied with the design, it is passed through a verification process to check for discrepancies. A code generation process is then invoked whereby a script file (describing the Petri Net graph) and a C program file (containing the necessary headers and stub callbacks) are generated. This C program file can then be compiled and linked with the API library to produce the executable. The script file is sourced during run-time by the scheduling system to translate the front-end Petri Nets graph into its internal repre-

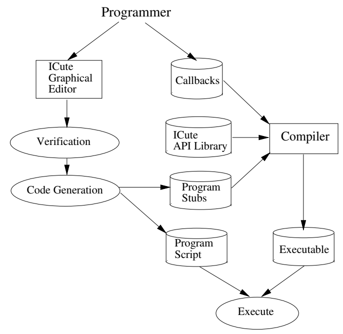

Figure 1: Development process of the Application

*This work was supported in part by the NSTB-MOE of Singapore under Research Project Grant PS964770

sentation.

The internal representation of the application consists of four separate modules, namely the user tasks, monitor, threads and scheduling support, working in unison. User tasks refer to the procedures/functions that are associated with each place in the Petri Net model. The monitor is a special task added to the application by *ICute* to perform all the necessary jobs of interprocess communication, system profiling, QoS negotiation, etc, on behalf of the application. It is implemented as a periodic task and currently set to execute at once per second to minimize overheads. A user-level real-time threads package has been developed in-house to execute all user tasks. Scheduling support refers to the scheduling control structures and algorithms which are responsible for scheduling the individual threads within the application.

3 SCHEDULING MECHANISM

In *ICute*, the overall scheduling mechanism in the system relies on a split-level decision making policy (see Figure 2). At the inter-application level, an incoming

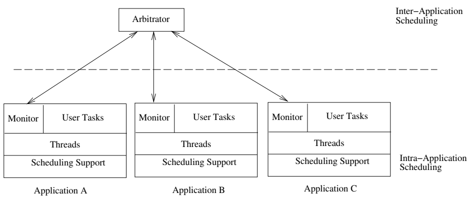

Figure 2: Split-level Scheduling system

application will negotiate with the *Arbitrator* about to the quality of service (QoS) it requires through its monitor. During the execution, the monitor feedbacks the status of execution to the *Arbitrator* who can then adjust the QoS dynamically. Within an application, *ICute* provides a frequency-based scheduling support.

The basic idea is to gradually decrease the execution requirement of the application during overloaded conditions such that it can still perform acceptably. To achieve this, we associate a frequency parameter, $f \in [0, 1]$, with every task. This frequency controls the desired rate at which the programmer wants the tasks to be executed. All the ready tasks must first go through a simple filtering agent which decides if the task should be executed or discarded. The filtering objective is to limit the demand on the CPU by preventing unimportant tasks from flooding the CPU.

A video capture-playback application is created to test the performance of the scheduling support. The application captures a 240x180 video image at 10 fps and produces three display windows. Two windows show the normal image (one is called the primary window and the other called the secondary window) while the third displays the image, texture-mapped onto a rotating b-spline. The experiment begins with the application running under normal system load. At time=3, we overload the system such that some form of degradation from the application is necessary. At time=24, we restore the system load to normal.

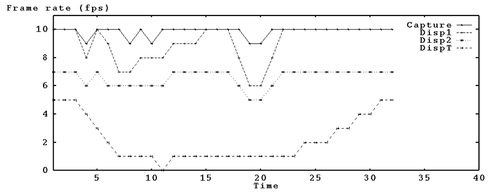

Figure 3: Experiment Results.

In our experiment, we assign the primary window display to have a frequency of $f = 1.0$, the secondary window display is assigned a frequency of $f = 0.7$ while the textured window display has a frequency of $f = 0.5$. In the graph (Figure 3), execution starts with all tasks executing at their desired frame rates. When the application is overloaded, the frame rates of all the tasks drop rapidly until they eventually stabilizes at some sub-optimal rate. Once stable, the scheduling mechanism then tries to get the tasks back to their original desired frequencies by raising the frame rate slowly. We see that at time=15, the application has regained the desired executions for both the primary and secondary window display. However, as it tries to increase the frame rate of the textured window display at time=17, this once again overloads the system and measures must be taken to restore stability. Once the overload condition is removed (at time=24), the textured window display slowly climbs back to its desired frequency again. From this experiment, we see that the use of the frequencies do give the programmer more control over the timing requirement of the application under overload conditions.

4 CONCLUSION

In this paper, we have presented a testbed, *ICute*, aimed at providing real-time scheduling support for multimedia applications under various load conditions.

References

[1] Geoff Coulson, Andrew Campbell, David Hutchison, and Francisco Garcia. Bt uri project; wp2/d2: Session acceptance and qos management in end-systems. Technical Report MPG-95-21, Distributed Multimedia Research Group, Department of Computing, Lancaster University, 1995.

[2] K. Jeffay and D. Bennett. A rate-based execution abstraction for multimedia computing. *Lecture Notes in Computer Science*, 1018:64–78, 1995.

[3] 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.