On Latency Management in Time-Shared Operating Systems[*]

Kevin Jeffay

University of North Carolina at Chapel Hill
Department of Computer Science
Chapel Hill, NC 27599-3175
jeffay@cs.unc.edu

Abstract: The design of general purpose operating systems impose constraints on the way one can structure real-time applications. This paper addresses the problem of minimizing the end-to-end latency of applications that are structured as a set of cooperating (real-time) tasks. When applications are structured as a set of cooperating tasks the time required for data to progress from an input task to an output task is a function of the number of the tasks that handle the data and the deadlines of individual tasks. We present an integrated inter-process communication and scheduling scheme that can be used to minimize the end-to-end latency of multi-threaded applications. Our approach is to provide the scheduler with information on the inter-process communication interconnections between tasks and to use this information to guarantee an end-to-latency to applications that is simply a function of the timing properties of the application and not its task structure. This scheme has been implemented within the YARTOS kernel and is presently being ported to the Real-Time Mach kernel.

1. Introduction

Multimedia applications that process streams of live and stored audio and video are stimulating research on the integration of real-time computation and communication services into general purpose, time-shared operating systems. While much is known about the scheduling and resource allocation problems that comprise the formal underpinnings of such services, techniques for implementing and using existing algorithms, in particular within the context of general purpose operating systems, have received relatively little attention.

In this note, we describe a problem that arose during the implementation of an experimental desktop video-conferencing system [4, 5]. Abstractly, the problem is that of minimizing end-to-end latency in real-time applications that consist of a set of cooperating tasks or threads. Here latency is defined as the difference between the times at which input data is first made available to an application thread and the time at which an application thread performs an output operation based on the input data. The thesis of this work is that by providing the kernel with information on the task structure of real-time applications, one can both dramatically reduce the worst case end-to-end application latency and employ relatively simple scheduling algorithms to provide real-time response to individual tasks.

The following section motivates the end-to-end latency problem using an idealized version of our video-conferencing system as an example. Section 3 outlines a real-time message passing service that we constructed within the YARTOS (Yet Another Real-Time Operating System) kernel [7]. We show how this service reduces worst case end-to-end latency and how it can be efficiently implemented. The YARTOS message passing service is currently being ported to the Real-Time Mach kernel [11] and will form the basis for a comparative study of the real-time performance of the YARTOS and RT-Mach thread models.

2. The End-to-End Latency Problem

Real-time computations require bounded response times. In general, by employing results from the real-time scheduling literature (*e.g.*, [10]), for relatively simple

[*] Supported in part by grants from the IBM Corporation, the Intel Corporation, and the National Science Foundation (numbers CCR-9110938 and ICI-9015443).

In: Proc. 11th IEEE Workshop on Real-Time Operating Systems and Software, May 1994, Seattle, WA, to pp. 86-90.

models of computation, it is possible to (1) determine conditions under which it is theoretically possible to guarantee that an invocation of a task will complete execution by a certain point in time, and (2) allocate resources within an operating system to ensure that an invocation of a task actually achieves its response time bound.

Often, it is desirable to guarantee a response time to a collection of cooperating tasks that execute in concert to realize some application. For example, consider the video processing portion of a desktop videoconferencing application. The goal of this application is to acquire, compress, and transmit a logically infinite sequence of digitized video frames across a network. The application is composed of the following (idealized) tasks:

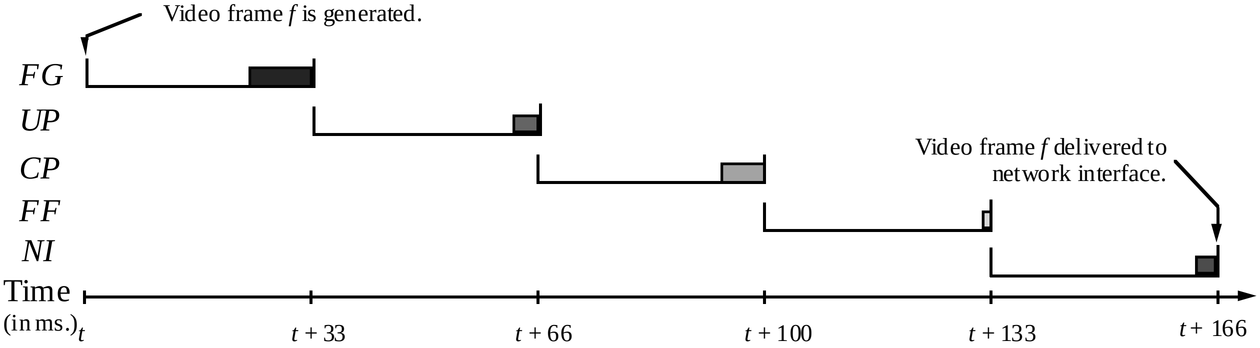
- *FG* — a task to control a frame-grabber that digitizes video frames generated by a camera,
- *UP* — a task to invoke user programs on the digitized frames for any user-level image processing that is desired (*e.g.*, for feature extraction and notification),
- *CP* — a task to compress the digitized frame,
- *FF* — a task to format and fragment the compressed frame(s) into network packets for delivery across a network, and
- *NI* — a task to control the network interface hardware.

These tasks cooperate to form a simple pipeline. Every video frame generated by the camera is digitized, processed by the user, compressed, formatted, and delivered to the network interface for transmission across the network.

In order for this conferencing application to be effective, two real-time constraints must be met. First, every video frame that is generated by the camera must make it through all stages of the pipeline and be delivered to the network interface. Second, the end-to-end latency of each frame — defined as the difference between the time the frame arrives at the network interface and the time the frame was generated — must be kept to a minimum. Since current video cameras and frame-grabbers generate data at regular, periodic intervals, the first constraint is easily satisfied by implementing each stage of the pipeline as a periodic task and using any number of real-time scheduling algorithms from the literature to schedule the tasks. The second constraint is not so easily satisfied.

An (NTSC) video frame is generated, and enters the pipeline, every 33.3 ms. In our implementation of the above video pipeline, every task has a period of 33.3 ms. Since there are 5 stages in the pipeline, the worst case end-to-latency of a video frame is 166.6 ms. The worst case occurs when each invocation of each task completes as late as possible within its period and stage $i + 1$ of the pipeline is not invoked until stage i has completed as shown in Figure 1. Whether or not the worst case actually occurs will depend on factors that are beyond the application writer's control such as the magnitude of the total system workload (*e.g.*, the number of other real-time and non-real-time tasks sharing the processor).

Note that the latency bound of 166.6 ms is really an artifact of the pipeline implementation of the video application and is not fundamental to the conferencing


Video frame f is generated.
FG
UP
CP
FF
NI
Video frame f delivered to network interface.
Time (in ms.)
t
t + 33
t + 66
t + 100
t + 133
t + 166

Worst case end-to-end latency for a single frame of video.

Figure 1

problem itself. For example, consider an implementation of the conferencing application that combines functions *FG*, *UP*, *CP*, *FF*, and *NI* into a single task. If the task had a period of 33.3 ms, then all video frames that are generated will be delivered to the network interface (assuming the system is still schedulable) and the worst case end-to-end latency of each frame will be no more than 33.3 ms.

The problem is that it is not always possible (or desirable) to collapse all application and system functions into a single task. In particular, when executed on top of a general purpose operating system, many of the real-time application's functions (such as input and output), are implemented by operating system system calls or servers (and associated device drivers) and are shared with other applications.

The challenge therefore is to support the pipeline model of application design and execution while not incurring the penalty inherent in the straightforward realization of the pipeline. Specifically, we would like to structure the conferencing application as a series of cooperating tasks and maintain a worst case end-to-end latency bound of 33.3 ms — the period of a single stage of the pipeline.

Note that in principle this should be possible since the total amount of computation (ignoring operating system overhead) performed by the single and multi-task implementations of the application are the same. The only difference is that in the multi-task implementation of the application, several video frames may be processed simultaneously (*i.e.*, several video frames may be in the pipeline at any one time).

3. A Real-Time Message Passing Service

Our solution to the problem of minimizing end-to-end latency is to make the pipeline structure of the conferencing application known to the kernel and to use this information to schedule the stages of the pipeline as if they were part of a larger sequential program. This technique has been implemented as part of the message passing system in the YARTOS kernel [7]. We begin with an overview of the YARTOS programming model.

The YARTOS kernel supports a simple data-flow model of real-time computation. Briefly, applications are composed of tasks, resources, and ports. Tasks are threads of control, resources are shared abstract data types, and ports are queues for messages. Tasks communicate with other tasks by sending messages to ports. Each port is bound to a unique task. When a message is sent to a port, the kernel schedules the task bound to the port so that the message will be consumed before a deadline defined by the rate at which the message sender emits messages. The deadline is chosen so as to ensure that all messages from this sender can be processed in real-time (*i.e.*, without any buffering). (The YARTOS programming model is explained in greater detail in [9]. The scheduling algorithm used in the kernel is described in [6].)

When tasks and ports are created, the kernel constructs a directed graph of all possible communication paths. When a message is sent from task T_i to task T_j, the deadline for task T_j is computed using the time of T_i's most recent invocation as invocation time for T_j. That is, tasks T_i and T_j are scheduled as if they were invoked simultaneously — as if they were a single task.

For example, assume task T_i is invoked at time t and has a deadline at time $t + p$. T_i executes sometime during the interval $[t, t+p]$ and sends a message to task T_j. No matter when the message is actually sent to T_j, task T_j is considered to have been invoked at time t. It is a property of the YARTOS programming model that the invocation of task T_j "occurring" at time t cannot have a deadline before time $t + p$. Therefore, during the interval $[t, t+p]$, T_j will not preempt T_i and when T_j is dispatched, there will be a message from task T_i for it to process.

The one exception to these invocation rules is when messages arrive from the outside world (*e.g.*, from interrupt handlers). When a task receives a message from an external process, the task's deadline is computed from the arrival time of the message (using application specified parameters that are sufficient for providing the desired real-time response to the external process).

With this message passing scheme, the time required in the worst case for a message to pass through tasks T_i and T_j in YARTOS is the same as the time required in the worst case for a message to be processed by a single task that combined the functions of T_i and T_j. Thus the worst case end-to-end latency of a multi-threaded YARTOS application is not a function of the task structure. Rather, it is a function of the deadlines associated with application messages.

For example, in our videoconferencing application, all messages have a deadline for processing of 33 ms. If the *FG* task receives a message (an interrupt in this case) at time *t*, then if this message results in messages being sent to tasks *UP*, *CP*, *FF*, and *NI*, all messages will be processed at or before time $t + 33$. Therefore, each video frame is delivered to the network interface no more than 33 ms after it was generated.

The alternate approach to minimizing end-to-end latency is to combine all video processing tasks into a single task. However, in our system tasks *FF* and *NI* are actually general purpose operating system services that are shared with other user applications and hence can not be embedded directly into the conferencing application. (In fact, it is largely for this reason that a common approach to achieving real-time performance in general purpose operating systems has been to move application code into the operating system where finer-grain control over resource allocation is also usually possible.)

4. Related Work

Our message passing system is related to the paradigm of communication and scheduling integration reported by Draves *et al*. [2]. In this work a scheduling and context-switching mechanism based on the programming language concept of continuations is introduced to allow an applications that consists of multiple threads to execute more like a single threaded application. The emphasis is [2], however, was on reducing system overhead. Our work seeks to minimize worst case end-to-end latency.

Other related work includes the general priority model of Harbour *et al.* [3], wherein periodic tasks can be decomposed into subtasks that may have varying execution priority. In such a model it is possible to more directly express and reason about what we have called end-to-end latency constraints. In our work we have argued that a simple scheduling algorithm (described in [6]) is sufficient for managing latency.

Lastly, the flow shop scheduling results of Bettati and Liu [1] are relevant. They consider the problem of minimizing end-to-end latency in a system of multiple processing elements (*e.g.*, a distributed system). We have only considered the latency problem on a single shared processor.

5. Conclusions and Future Work

The design of general purpose operating systems impose constraints on the way one can structure real-time applications. Common operating system services such as network transport protocols, and device management need to be used by real-time applications. Because such services are shared with other applications they cannot by tightly bound to the real-time applications. We have shown that making application inter-task communication paths known to the kernel, one can provide a worst case end-to-end application latency bound that is the equivalent to the bound for an implementation of the application as a single task.

While we described the real-time message passing service within the context of an application whose tasks form a pipeline, the service can be applied to any graph structure to minimize latency of message communication along any path in the graph.

Currently we are porting the YARTOS message passing service to RT Mach (MK83) kernel and hope to compare the end-to-end latency of applications using the RT Mach and YARTOS communication primitives.

6. References

[1] Bettati, R., Liu, J.W.-S., *End-to-End Scheduling to Meet Deadlines in Distributed Systems*, Proc. IJDCS '92, Yokohama, Japan, pp. 452-459.

[2] Draves, R.P., Bershad, B.N., Rashid, R.F., Dean, R.W., *Using Continuations to Implement Thread Management and Communication in Operating Systems*, Proc. 13th ACM Symp. on Operating System Principles, Pacific Grove, CA, October 1991, pp. 122-136.

[3] Harbour, M.G., Klein, M.H., Lehoczky, J., *Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority*, Proc. 12th IEEE Real-Time

Systems Symp., San Antonio, TX, December 1991, pp. 116-128.

[4] Jeffay, K., Stone, D.L., and Smith, F.D., *Transport and Display Mechanisms for Multimedia Conferencing Across Packet-Switched Networks*, Computer Networks and ISDN Systems, to appear.

[5] Jeffay, K., Stone, D.L., and Smith, F.D., 1992. *Kernel Support for Live Digital Audio and Video*. Computer Communications, Vol. 16, No. 6 (July), pp. 388-395.

[6] Jeffay, K., 1992. *Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems*, Proc. 13th IEEE Real-Time Systems Symp., Phoenix, AZ, December 1992, pp. 89-99.

[7] Jeffay, K., Stone, D.L., Poirier, D., *YARTOS: Kernel support for efficient, predictable real-time systems*, in “Real-Time Programming,” W. Halang and K. Ramamritham, eds., Pergamon Press, Oxford, UK, 1992, pp. 7-12.

[8] Jeffay, K., *Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems*, Proc. 13th IEEE Real-Time Systems Symp., Phoenix, AZ, December 1992, pp. 89-99.

[9] Jeffay, K., *The Real-Time Producer/ Consumer Paradigm: A paradigm for the construction of efficient, predictable real-time systems*, Proc. 1993 ACM/SIGAPP Symposium on Applied Computing, Indianapolis, IN, ACM Press, February 1993, pp. 796-804.

[10] Liu, C.L., Layland, J.W., *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, Journal of the ACM, Vol. 20, No. 1, (January 1973), pp. 46-61.

[11] Tokuda, H., Nakajima, T., Rao, P., *Real-Time Mach: Towards a Predictable Real-Time System*, Proc.USENIX Mach Workshop, Burlington, VT, October 1990, pp. 73-82.