YARTOS
Kernel support for efficient, predictable real-time systems[*]

Kevin Jeffay, Donald L. Stone, Daniel E. Poirier

University of North Carolina at Chapel Hill,
Department of Computer Science, Chapel Hill, NC, USA

Abstract. YARTOS is an experimental real-time operating system kernel that provides guaranteed response times to tasks. It is currently used as a vehicle for research in the design, analysis, and implementation of real-time applications. It is a micro-kernel with an underlying formal model based on sporadic tasks with response time requirements and shared software resources. It is distinguished by the programming model it supports and by its use of a novel processor scheduling and resource allocation policy. The implementation of YARTOS is outlined and two real-time applications that run under YARTOS are described.

Introduction

To be considered correct, a real-time computer system must perform its computations and I/O operations in a time frame defined by processes in the environment external to the computer. In the real-time domain the environment is viewed as imposing a set of constraints — commonly referred to as "timing constraints" — on the correct operation of the system. For example, when manipulating live digital video, a real-time system must process video data at the (precise) rate of 30 video frames a second in order to preserve the fidelity of the image.

The existence of timing constraints distinguish real-time systems from more traditional multi-programmed systems. A real-time operating system therefore differs from a more conventional operating system in that it should provide mechanisms for ensuring that timing constraints are not violated. Our view is that a real-time operating system must support the notion of a *predictable* computation. In much the same manner as (sequential) programming language constructs have well defined semantics that enable a programmer to prove logical properties of a program, a real-time operating system should have a temporal semantics for its services that will enable a programmer to reason about temporal properties of a program.

In this paper we present an overview of an operating system kernel called YARTOS (Yet Another Real-Time Operating System) that supports the construction of efficient, predictable real-time systems. Initially we are focusing on the problem of designing and constructing *hard-real-time* systems. Hard-real-time systems are real-time systems that require guaranteed adherence to timing constraints. These are systems in which the cost of failing to interact with the external environment in real-time is high. This high cost can be measured in monetary terms (*e.g.*, an inefficient use of raw materials in a process control system), aesthetic terms (*e.g.*, unrealistic output from a computer music or computer animation system), or possibly in human or environmental terms (*e.g.*, an accident due to untimely control in a nuclear power plant or fly-by-wire avionics system).

In recent years, numerous real-time operating systems have been developed (see [Zhao 89] for a good survey). Our work is distinguished by the programming model that YARTOS supports and by the aggressive use in YARTOS of recent results in the theory of deterministic processor and resource allocation. The programming model supported by YARTOS is an extension of Wirth's discipline of real-time programming [Wirth 77]. In essence it is a message passing system with a semantics of inter-process communication that specifies the real-time response that an operating system must provide to a message

[*] This work was supported in parts by grants from the National Science Foundation (numbers CCR-9110938 and ICI-9015443), and by a Digital Faculty Program grant from Digital Equipment Corporation.

In Proc. Joint IEEE Wrkshp on Real-Time Operating Systems and Software and IFAC/IFIP Wrkshp on Real-Time Programming, Atlanta, GA, May 1991, Real-Time Systems Newsletter, Vol. 7, No. 4, (Fall 1991), pp. 8-13. Republished in "Real-Time Programming," W. Halang and K. Ramamritham, eds., Pergamon Press, Oxford, UK, 1992., pp. 7-12.

receiver. This allows a programmer to assert an upper bound on the time to receipt and processing of each message. The exact response time requirement is a function of such factors as the rate with which a process receives messages on a given input channel. Ultimately, these rates are functions of the rates at which data arrives from external sources. These semantics provide a framework both for expressing processor-time-dependent computations and for reasoning about the real-time behavior of programs. The programming model is described in greater detail elsewhere [Jeffay 89a].

Programs that execute under YARTOS are compiled into a set of *sporadic* tasks that share a set of serially reusable, single-unit resources. A sporadic task is a sequential program that is invoked in response to the occurrence of an event. An event is a stimulus that may be generated by processes external to the system (*e.g.*, an interrupt from a device) or by processes internal to the system (*e.g.*, the arrival of a message). We assume events are generated repeatedly with a (non-zero) lower bound on the duration between consecutive occurrences of the same event. A resource in YARTOS is a software object (*e.g.*, an abstract data type) that is shared (read/write) by multiple tasks. For a given workload, the goal of YARTOS is to guarantee that (1) all requests of all tasks will complete execution before their deadlines and (2) no shared resource is accessed simultaneously by more than one task. We have developed an optimal (preemptive) algorithm for sequencing such tasks on a single processor [Jeffay 89b, 90]. The algorithm is optimal in the sense that it can provide the two guarantees whenever it is possible to do so. Moreover, an efficient algorithm has been developed for determining if a workload can be guaranteed a correct execution [Jeffay 90]. Our development and analysis of a formal scheduling model has resulted in a surprisingly efficient implementation of YARTOS tasking. Specifically, applications consisting of multiple tasks can be executed on a single run-time stack and no explicit locking mechanism is required for accessing shared resources. This improves the memory utilization of the system and yields efficient context switches between tasks. This encourages liberal use of tasks and data sharing in YARTOS applications.

In this note we concentrate on the YARTOS's scheduling model and its implementation. The following section describes the scheduling model and the algorithms used for processor and resource allocation. Section three briefly describes a prototype implementation of the YARTOS kernel. We conclude with some brief comments on our experiences with YARTOS.

Scheduling Model

The YARTOS scheduling model is composed of two basic abstractions: *tasks* and *resources*. A *task* is an independent thread of control (a sequential program) that is invoked at sporadic intervals. Each invocation of a task must complete execution before a well-defined deadline. The invocation interval and deadline of a task is derived from constructs in the higher-level programming model. During the course of execution, a task may require access to some number of *resources*. A resource is a software object (an abstract data type) that encapsulates shared data and exports a set of procedures for accessing and manipulating the data. Like a monitor, objects require mutually exclusive access to the data they encapsulate.

Formally, a YARTOS workload consists of a set of n sporadic tasks $\{T_1, T_2, ..., T_n\}$ and a set of m serially reusable, single unit resources $\{M_1, M_2, ..., M_m\}$. A task is described by a pair $T = (C, R)$ where C is the computational cost: the maximum amount of processor time required to execute the program of task T to completion on a dedicated uniprocessor, and R is the response time requirement of task T: the maximum time allowed between an invocation of task T and the completion of the execution of T's sequential program. Presently response time parameters are derived from the *rate* at which tasks are invoked where the rate is measured in terms of minimum inter-invocation time. For a task T we assume that the response time requirement is equal to the minimum time interval between successive invocations of T. This characterization of real-time task behavior is motivated by the class of real-time applications wherein timing constraints arise from the need to ensure that input data is acquired and processed in real-time.

The computational cost of a task is expressed as the sum of the costs of executing code to perform operations on shared resource and private task code. Let n_i be the number of operations on shared resources performed by an invocation of task T_i, and let c_{i1}, c_{i2}, ..., c_{in_i} be the maximum execution time required for each operation. Let c_{i0} be the maximum execution time required to execute the remaining code (sequential code in task T_i that does not require access to a shared resource). Hence $C_i = c_{i0} + c_{i1} + c_{i2} + ... + c_{in_i}$.

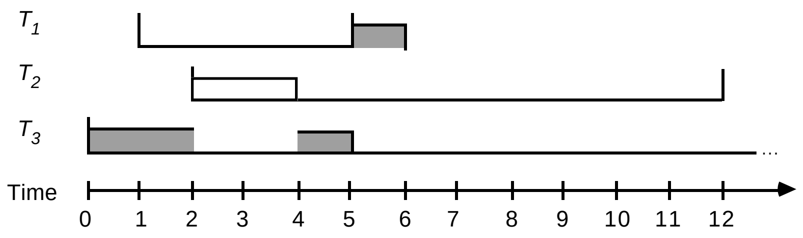

Figure 1

Scheduling Tasks

Given a set of tasks and a set of resources, the YARTOS kernel must sequence the execution of task invocations on a uniprocessor such that in all cases — and in particular in the worst case — it is guaranteed that:

- each invocation of each task completes execution at or before its deadline, and
- a resource is never accessed by more than one task simultaneously.

A set of tasks is said to be *feasible* if it is possible to achieve these two criteria. We have developed a scheduling algorithm that will schedule any feasible task set [Jeffay 89b, 90]. The algorithm results from the integration of a synchronization scheme for access to shared resources with the *earliest deadline first* (*EDF*) processor scheduling algorithm of Liu and Layland; a preemptive, priority-driven scheduling algorithm with dynamic priority assignment [Liu & Layland 73].

Under the YARTOS scheduling discipline, a task has *two* notions of a deadline: one for the initial acquisition of the processor, and one for execution of operations on resources. Tasks have separate deadlines for performing operations on shared resources to avoid a variant of the priority inversion problem [Sha *et al.* 90]. This problem can occurs when tasks with large response time requirements ("low priority") perform operations on resources that are shared with tasks with small response time requirements ("high priority").

For example, consider the set of three tasks T_1 = (1, 4) (*i.e.*, task T_1 has a computational cost of 1 time unit and a response time requirement of 4 time units), T_2 = (2, 10), T_3 = (3, 20). Assume that tasks T_1 and T_3 consist of a single operation on a shared resource and that task T_2 does not use this resource. If an *EDF* scheduling policy is used and the tasks are invoked as shown in Figure 1 then task T_1 misses a deadline at time 5. (Striped rectangles in Figure 1 denote execution with a shared resource.) Initially task T_3 will be scheduled at time 0. At time 1 task T_1 is invoked and has the nearest deadline (at time 5). However, since T_1 requires the resource that is in use by task T_3, task T_1 is blocked by task T_3 and task T_3 continues execution. At time 2, task T_2 has a nearer deadline than the executing task T_3 (time 12 versus time 20). Since task T_2 has the nearer deadline, one might be tempted to allow task T_2 to preempt task T_3. However, as illustrated in Figure 1, such a decision will cause task T_1 to fail at time 5.

The problem is that at time 1, it is no longer sufficient for the invocation of task T_3 occurring at time 0 to be completed by its nominal deadline at time 20. Since tasks T_1 and T_3 share a resource, when task T_1 is invoked at time 1, the invocation of task T_3 occurring at time 0 must now be completed no later than time 5: the initial deadline of task T_1.

Under the scheduling discipline we have developed, when a task is invoked at time t, the invocation will have an initial deadline at time $t + R$ as in traditional EDF scheduling. This deadline is a deadline for the task to complete execution. Once a task has been

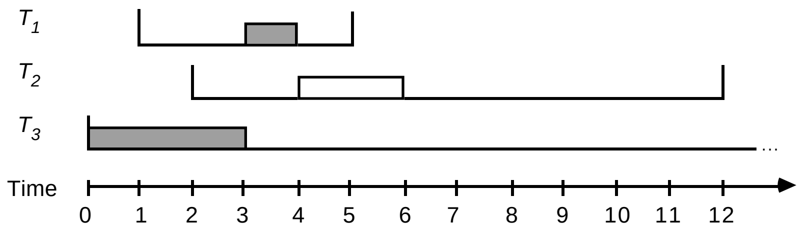

Figure 2

scheduled and is executing, it may perform an operation on a shared resource. If a task commences an operation on a resource at time t' then at time t' the task will have a deadline equal to time MIN($t + R$, $t' + 1 + R_{min}$) where R_{min} is the smallest response time requirement of all the tasks that can access the resource. This manipulation of deadlines ensures that there will exist no contention for shared resources and hence ensures mutual exclusion on resource operations. If at or after time t', another task is invoked that wishes to perform an operation on this same resource, then that task will initially have a deadline that is necessarily greater than $t' + R_{min}$. Since we use deadline scheduling this task will therefore not preempt the task currently using the shared resource. For example, applying the YARTOS scheduling discipline to the three tasks from the previous example yields the execution shown in Figure 2. This time task T_3 initially commences execution with a deadline at time 5 (*i.e.*, MIN(0 + 20, 0 + 1 + R_{min}) = 5). Now, at times 1 and 2 task T_3 has the nearest deadline and will not be preempted by either task T_1 (ensuring mutual exclusion on the resource task T_3 shares with task T_1) or task T_2 (ensuring task T_3 will complete execution in time for task T_1 to make its deadline).

For the problem of scheduling a set of sporadic tasks that share a set of single unit serially reusable resources, the YARTOS scheduling algorithm is optimal with respect to the class of scheduling policies that do not use inserted idle time.[1] The algorithm is optimal in the sense that it can schedule a set of tasks, without inserted idle time, whenever it will be possible to do so. This dynamic manipulation of a task's deadline is similar to the concept of a priority ceiling in priority inheritance protocols [Sha *et al.* 90].

Deciding Feasibility

To ensure deadlines are not missed at run-time, it is necessary to have a test, or decision procedure, that can be applied to a set of tasks and will determine whether of not the tasks are feasible. We have developed an efficient decision procedure for deciding if a set of tasks is feasible. This allows us to determine *a priori* whether or not a set of tasks can be executed in real-time. In practice, this test is performed before applications are executed under YARTOS.

A set of tasks will be feasible on a single processor if:[2]

1) $\sum_{i=1}^{n} \frac{C_i}{R_i} \le 1$,

2) $\forall i,\ 1 \le i \le n;\ \forall k,\ 1 \le k \le n_i;\ \forall L,\ R_{min,k} < L < R_i$:

$$L \ \ge\ c_{ik} - 1 + \sum_{j=1}^{i-1} \left\lfloor \frac{L}{R_j} \right\rfloor C_j \ ,$$

where

- n is the number of tasks in the system,
- n_i is the number of operations on shared resources performed by an invocation of task T_i, and
- $R_{min,k}$ is the smallest response time requirement of the tasks that access resource M_k.

The term C_i/R_i is the fraction of the processor that must be allocated to processing invocations of task i. The first condition stipulates that the processor not be overloaded. Condition (2) applies only to tasks that require access to resources (tasks for which $n_i > 0$). It quantifies the processor demand that occurs when tasks simultaneously access a shared resource. The right hand side of the inequality in condition (2) is a least upper bound on the processor demand that can be realized in an interval of length L starting at the time an invocation of a resource requesting task i is scheduled, and ending sometime before the deadline for completion of the invocation. Under all circumstances this bound must be less than or equal to the minimum inter-invocation time (or a fraction thereof) of task i. A set of tasks can be tested against these conditions in time $O(m\,R_{max})$ where m is the number of resources in the system and R_{max} is the largest response time requirement.

Implementation

YARTOS is currently implemented on IBM PS/2 computers (Intel 80386 processor). When the machine is booted it initially executes the DOS operating system. When YARTOS applications are executed,

[1] If tasks are scheduled by a discipline that allows itself to idle the processor when there exists a task with an outstanding request for execution, then that discipline is said to use *inserted idle time*.

[2] Necessary and sufficient conditions for feasibility are presented in [Jeffay 90]. For brevity, we present a simpler (sufficient) formulation of these conditions.

YARTOS replaces the interrupt vectors to point to an entry in the kernel and inserts code to handle all software interrupts (*i.e.*, TRAPS). At this point YARTOS is in control of the machine. YARTOS creates user tasks and resources and then waits for an interrupt to commence execution of the application program. When the application terminates YARTOS restores the original DOS environment and returns control to the DOS command interpreter.

YARTOS is a “micro-kernel” that supports three basic abstractions: tasks, resources, and messages. Tasks and resources were introduced in the previous section. Messages are typed collections of data. More generic operating system facilities such as a file system can be implemented on top of YARTOS by users. For example, as described shortly, the DOS file system can be easily made available to YARTOS applications by encapsulating DOS functions as resource operations.

Tasks

A YARTOS application is a *C* program that is linked with a YARTOS specific library.[3] This library provides the YARTOS system call interface. There are two classes of systems calls: start-up, to initialize the kernel and application(s), and run-time, for task synchronization and communication. Start-up system calls include entries to create tasks and resources. Run-time calls include entries to send messages or perform an operation on a resource. For example, a task is created by the system call:

```
port = create_task( task_descriptor);
```

The task descriptor is a template that the programmer instantiates and then customizes. Template entries include: a name (a character string), the task body (a pointer to a *C* function), a list of resources used by the task, and the required response time or deadline (the maximum allowed time between task invocation and completion). Optional parameters include routines to be called on each context switch involving this task to save and restore task specific state information (*e.g.*, device registers). (These last parameters are in keeping with the micro-kernel philosophy. YARTOS maintains a minimal machine state for preempted tasks: the program counter and general purpose registers. User tasks may extend the machine state when, for example, performing low-level device control. In such cases users are responsible for writing their own functions for saving and restoring their extended state.)

[3] For concreteness we refer to the *C* programming language as the application implementation language. YARTOS makes no assumptions about the implementation language of tasks and resources.

Inter-task communication in YARTOS is via message passing. Messages are sent to *ports*. There is a one-to-one correspondence between ports and tasks. The level of indirection provided by ports allow tasks to be separately compiled and linked. After a task is created, it will commence execution when it receives a message. A task is a sequential program (*e.g.*, a *C* function). Upon receipt of a message a task will execute to completion (possibly being preempted by other tasks and later resumed). If the tasks currently in the system are feasible then there will be at most one instance of each task either ready for execution or executing at any time.

Resources

A resource is a *C* program with persistent data. Resources are created with a system call. The call registers a set of functions for operating on a resource. These functions are then accessed indirectly through the kernel by tasks. When a task attempts to perform an operation on a resource the scheduler is invoked and the current deadline of the requesting task is possibly advanced to an earlier point in time as described in the previous section.

The concept of a resource is useful for incorporating utilities written for other operating systems into the YARTOS environment. For example, DOS is treated as a single large resource. DOS calls (TRAPS) are intercepted by YARTOS and mapped into resource calls. This ensures that at most one YARTOS task is executing inside DOS at a time. This works particularly well with DOS since it is (roughly speaking) a sequential program and does not directly support concurrent threads of control.

By treating DOS as a resource YARTOS programmers have access to an expanded list of operating system services.

Device Management

To ensure a faithful implementation of the scheduling model, YARTOS must ensure that *all* computational activities are dispatched by the scheduler. This means that traditional non-dispatched activities, such as all interrupt handlers, are implemented as tasks and are scheduled in the same manner as user tasks. Interrupt handlers must be scheduled to ensure that tasks with near deadlines do not fail. In YARTOS, traditional interrupt handlers are tasks that are created by the user and invoked by a hardware signal rather than by a message arrival. The deadlines of these tasks are assigned based the expected inter-arrival time of the interrupt. In general this information may not be known or reliable, however, this has not been a problem for the real-time applications we have constructed using YARTOS. (These applications are briefly described in the following section.)

Scheduling-Based Optimizations

A final note on the YARTOS implementation concerns the low-level implementation of tasks. Although the YARTOS programming model is one of communicating sequential processes, a collection of tasks is implemented more like a single sequential program than a set of independent threads of control. Specifically, all YARTOS tasks execute on a single run-time stack. Such an implementation is made possible by the scheduling policy we have developed. If a task T is preempted, it is the case that any task that executes while T is preempted, is guaranteed to complete execution and terminate before T is resumed. Since tasks execute to completion, we may execute all tasks on a single run-time stack. This greatly improves memory utilization and reduces context switching overhead by reducing the machine state. (This optimization is an implementation of a simple form of Baker's stack allocation policy [Baker 90].)

A second optimization concerns mutual exclusion for shared resources. Since the YARTOS scheduling policy eliminates contention for resources at run-time by manipulating the deadline of a task that is operating on a resource, YARTOS need not provide any special locking facilities for accessing shared resources.

These two properties of our scheduling algorithm afford us an extremely efficient implementation of tasks.

Status and Experiences

A YARTOS prototype has been constructed that implements both the tasking model and scheduling discipline described in the previous section. It is implemented in *C* on an IBM PS/2 computer and consists of approximately 2500 lines of code. Its small size is due largely to the fact that we may implement tasks and resources in a simple and straightforward manner.

YARTOS has been used to support two real-time applications: a 3-dimensional graphics display system used for research in *virtual realities* [Chung *et al.* 89] and a workstation-based conferencing system using digital audio and video [Jeffay & Smith 91]. The graphics system consists of a head-mounted display system (a helmet with miniature television monitors embedded in it), and tracking hardware for the helmet and for a hand-held pointing device. A computer generated image of a 3-dimensional "virtual world" is displayed in the helmet. The goal of the system is to track the user's head and pointing device in real-time and to update the image displayed in the helmet so as to maintain the illusion that the user is immersed in an artificial world.

There are two separate real-time concerns in this application. First, the system must update the display at a rate sufficient for ensuring that animate objects displayed in the helmet move in a smooth and realistic manner. Second, as the user moves her head or pointing device, the displayed image must appear to move with the user's movements. Such constraints were simple to model and realize with YARTOS. A task graph for this application is shown in Figure 3. Dark circles represent hardware devices, circles represent tasks and double circles represent resources.

YARTOS and its programming system provide a framework for both expressing processor-time-dependent computations and for reasoning about the real-time behavior of programs. For example, in the head-mounted display application, we can demonstrate that the maximum time between the arrival of a complete head and hand position report and the display of an image based on the new position information is approximately 100ms. This determination is made based on the response time guarantees provided by YARTOS, the structure of the application, and the semantics of the programming discipline. In particular, this analysis is independent of other applications that may be executing simultaneously. By performing some simple restructuring of the program, we were able to reduce this performance guarantee to approximately 33ms (see [Jeffay 89a]).

Currently, YARTOS is being used to experiment with digital audio and video on a local area network of workstations [Jeffay & Smith 91]. The hardware consists of a (small) number of workstations interconnected with a 16Mbit token ring network. Each workstation is connected to a video camera, microphone, speaker, and video monitor, and contains real-time video encoding/decoding hardware with

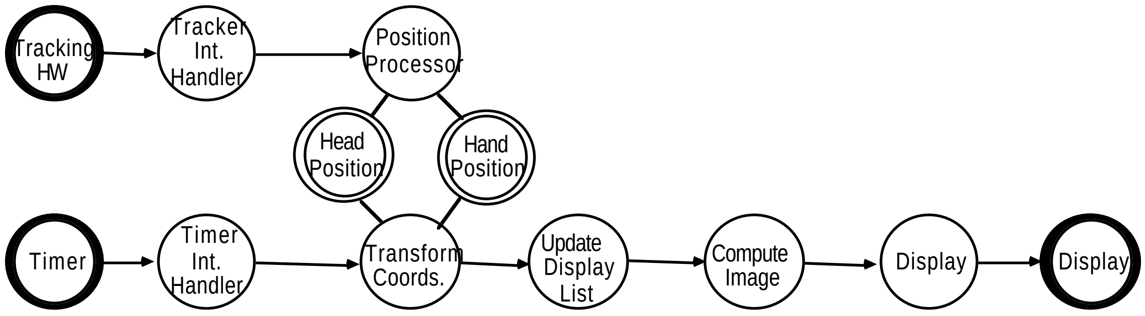

Figure 3

compression capabilities (Intel's ActionMedia Digital Video Interactive product [Ripley 89]). The goal is to use the workstation and network to emulate a cable television network. This requires real-time data acquisition, processing and transmission. The motivation for this project is to support the remote, real-time collaboration of scientific and technical professionals [Smith *et al.* 90].

In addition to reasoning about timing properties of the application, we are experimenting with kernel support for reasoning about logical correctness issues. For example, in the digital video system the tasks that control the digitization, compression, and transmission of video data on a workstation operate in a pipeline. A video frame is first digitized, then compressed and then packetized and sent over the network. These three tasks share a buffer pool of video frames. When a task acquires a buffer we would like to assert that a boolean expression describing the state of the contents of the buffer is true (*e.g.*, that a buffer contains compressed video data). Ideally, we would like to demonstrate that the application can never enter a state in which the assertion is false given the text of the program and the scheduling guarantees provided by YARTOS.

Our approach is to use a simplified version of Jahanian and Mok's real-time logic *RTL* [Jahanian & Mok 86] for reasoning about safety properties of a YARTOS application [Jeffay & Stone 91]. Safety assertions are formulated in terms of relations on observable and measurable events such as the number of messages sent to a task. We manually prove that the truth of these assertions implies that the desired safety property holds. During the execution of a YARTOS application, the kernel maintains a count of all messages sent to and received by a task. A user can perform a system call to obtain the current value of an event count and thereby dynamically verify the correct operation of peer tasks.

Summary

YARTOS is an experimental system for research in the design and implementation of real-time applications. It is a micro-kernel with an underlying formal model of sporadic tasks with response time requirements and shared software resources. It is distinguished by the programming model it supports and by its use of a novel processor scheduling and resource allocation policy.

YARTOS is not a panacea for real-time computing problems. It is designed to support a particular paradigm of timing constraints and resource usage. It has, however, proved to be a useful vehicle for real-time applications that are primarily concerned with processing long-lived, uniform data streams (*e.g.*, the class of so-called *continuous media* applications). It has allowed us to design, implement, and analyze the performance of such real-time applications in a convenient manner. As such it represents a useful step towards the goal of supporting predictable real-time computing.

References

Baker, T.P. (1990). *A Stack-Based Resource Allocation Policy for Real-Time Processes*, Proc. Eleventh IEEE Real-Time Systems Symp, Lake Buena Vista, FL, December 1990, pp. 191-200.

Chung, J.C., Haris, M.R., Brooks, F.P., Fuchs, H., Kelley, M.T., Hughes, J., Ouh-young, M., Cheung, C., Holloway, R.L., Pique, M. (1989). *Exploring Virtual Worlds with Head-Mounted Displays*, Non-Holographic True 3-Dimensional Display Technologies, SPIE Proceedings, Vol. 1083, Los Angeles, CA, January 1989.

Jahanian, F., Mok, A. K.L. (1986). *Safety Analysis of Timing Properties in Real-Time Systems*, **IEEE Trans. on Soft. Eng.**, Vol SE-12, No. 9, pp. 890-904.

Jeffay, K. (1989). *The Real-Time Producer/Consumer Paradigm: Towards Verifiable Real-Time Computations*, Ph.D. Thesis, University of Washington, Department of Computer Science, Technical Report #89-09-15, September 1989.

Jeffay, K. (1989). *Analysis of a Synchronization and Scheduling Discipline for Real-Time Tasks with Preemption Constraints*, Proc. Tenth IEEE Real-Time Systems Symp., Santa Monica, CA, December 1989, pp. 295-305.

Jeffay, K. (1990). *Scheduling Sporadic Tasks With Shared Resources in Hard-Real-Time Systems,* University of North Carolina at Chapel Hill, Department of Computer Science, Technical Report TR90-038, August 1990. (Submitted for publication.)

Jeffay, K., Smith, F.D. (1991). *System Design for Workstation-Based Conferencing With Digital Audio and Video*, Proc. IEEE Conference on Communication Software: Communications for Distributed Applications and Systems, Chapel Hill, NC, April 1991, pp.169-180.

Jeffay, K., Stone D. (1991). Operating system and programming language support for predictable real-time computations. (In preparation.)

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

Ripley, G.D. (1989). *DVI - A Digital Multimedia Technology*, **CACM**, Vol. 32, No. 7, pp. 811-822.

Sha, L., Rajkumar, R., Lehoczky, J.P. (1990). *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*, **IEEE Trans. on Computers**, Vol. 39, No. 9, pp. 1175-1185.

Smith, J.B., Smith, F.D., Calingaert, P., Hayes, J.R., Holland, D., Jeffay, K., Lansman, L. (1990). *UNC Collaboratory Project: Overview*, University of North Carolina at Chapel Hill, Department of Computer Science, Technical Report TR90-042, November 1990.

Wirth, N. (1977). *Toward a discipline of real-time programming*, **CACM**, Vol. 20, No. 8, 577-583.

Zhao, W. (Ed.) (1989). **Operating Systems Review**, Vol. 23, No. 3.