The Design, Implementation, and Use of a Sporadic Tasking Model

Kevin Jeffay *David Becker* *David Bennett*
Shaun Bharrat *Timothy Gramling* *Mark Housel*
University of North Carolina at Chapel Hill
Department of Computer Science
Chapel Hill, NC 27599-3175 USA
April 1994

1. Introduction

This paper describes the design, implementation, and use of a model of sporadic tasks in the Real-Time Mach kernel (MK83) [15]. The purpose of this exercise was two-fold. First, was our desire to provide a programming model in real-Time Mach (RT-Mach) that was better suited to the demands and requirements of applications that must execute in response to non-periodic, but repetitive events. Examples of such applications include those that manipulate live multimedia data, such as videoconferencing systems, and those that need to control asynchronous devices such as local-area network adaptors, to implement real-time communications protocols. Second, we wanted to study the issues involved in applying recent scheduling theory results for sporadic tasks in the context of a general purpose kernel such as the Mach kernel.

With respect to these goals this work makes two contributions. The first is the development of programming model for real-time systems that allows for the modular construction of real-time applications as a set of cooperating processes where the real-time performance of an application is invariant of its structure. The second is a new result in the theory of sporadic tasks that is used to greatly simplify the implementation of sporadic tasks.

The starting point for this work is the model of sporadic tasks developed by Mok [12], and later extended by Jeffay [8], and Jeffay and Stone [6]. A sporadic task is a simple variant of a periodic task. Whereas periodic tasks recur at constant intervals, sporadic tasks simply have an upper bound on the their rate of occurrence (*i.e.*, a lower bound on their inter-arrival time). The fact that sporadic tasks may execute at a variable (but bounded) rate makes them well-suited for supporting event-driven applications. At present, the theory of sporadic tasks is general enough to accommodate a model of computation wherein tasks may communicate via shared memory (*i.e.*, tasks may have

critical sections) [8], and tasks may preempted by interrupt handlers (*i.e.*, realistic device interactions can be modeled) [6]. A set of relations on model parameters that are necessary and sufficient for tasks to execute in real-time are known and an optimal algorithm for scheduling tasks (based on earliest-deadline first scheduling) has been developed. A subset of this theory was applied to a design and implementation of a videoconferencing system [5]. A special purpose real-time kernel and programming system was developed for that effort [7]. The goal now is to provide a more complete realization of the theory and to refine the programming model to deal with issues arising from the implementation of sporadic tasks in a more general purpose computing environment.

Related work includes the extensions to the (non-real-time) Mach kernel to provide time-constrained event handlers and virtual memory support for multimedia applications [14]; the work on integrating scheduling and IPC mechanisms to avoid unbounded priority inversions when real-time tasks (clients) invoke non-real-time services (services) [13]; and the work on providing real-time performance predictability within the context of a general purpose time-sharing system [11]. Each work seeks to better design operating system services to meet the needs of real-time (and in particular multimedia) applications. Our work shares these goals, however, we have chosen to experiment with a different set of resource allocation policies, particularly those based on deadline scheduling. Moreover, we are somewhat more academic in our approach as we also seek to understand how the theory of real-time scheduling needs to evolve to meet the needs of these applications.

The remainder of the paper is divided into three sections. In the following section we outline a programming discipline we have developed for building predictable real-time applications. Section 3 discuses the implementation of the discipline and discusses a simple extension to the theory that was required to overcome a basic limitation in the formal model. Section 4 presents the realization of the implementation in the RT-Mach kernel and gives some performance results for some alternative task implementation strategies. We conclude in Section 5 with a discussion of future work.

2. A Real-Time Programming Discipline

Our extensions to the RT-Mach kernel support a programming model with a (arguably) higher-level of abstraction over the simple periodic thread model present in RT-Mach. Our programming system is primarily design for generic real-time data-flow applications

— applications that read data from a set of input devices, manipulate the data, and eventually write data to a set of output devices, all in a relatively continuous manner.

In our system an application is structured as a directed graph (called a *process graph*) where vertices represent processes and edges represent unidirectional communication channels. Processes exchange messages along communication channels. Processes may be either sequential programs that execute on a single processor or physical processes in the environment external to the processor that communicate with internal processes via interrupts. In the latter case, processes are simply stubs (*i.e.*, non-executable code) that specify the real-time characteristics of the external processes they represent. A message is typed collection of data. A simplified *conceptual* schema for an internal process is

```
process P
  loop
    Accept( in_mesg)
    <compute>
    Emit(out_mesg, channel1)
    <compute>
    Emit(out_mesg, channel2)
      :
      :
  end loop
end P1
```

Communication and synchronization in our system are based on the client/server paradigm of message passing. A process has a single input port and a set of output ports. A process repeatedly accepts a message on its input port, processes the message — possibly emitting messages to other processes — and then attempts to accept another input message. Message passing is asynchronous. The Emit statement is always non-blocking while the Accept statement is potentially blocking. If a message is not available for a process that executes an Accept statement then the process is blocked until a message is available.

Each channel in a process graph defines a client/server (or producer/consumer) relationship between two processes. The novel abstraction in our programming system is that all messages are guaranteed to be processed in real-time (*i.e.*, without buffering). When a message is sent on a channel it will be received *and processed* before the next message is sent on that channel. (This paradigm of programming is elaborated on more completely in [9].)

For example, if a process *S* acts as a server for a client process *C*, then *S* will consume each message from *C* before *C* sends its next message. More precisely, for each output

port of a process we define a *transmission rate function*. This function relates the maximum rate at which a client produces messages on a channel to the rate at which messages arrive at the client. This rate is defined in terms of the worst case minimum inter-arrival time of messages. For this message passing discipline it can be shown that the transmission rate functions have the form $f(r) = (1/x)r$, where r is the rate at which messages arrive at the process and x is a positive, non-zero integer. For a given transmission rate function the value of x will depend on the logic within a process. (Conceptually, a process is modeled as a finite state machine in which state transitions occur upon the receipt of a message. The parameter x in the transmission rate function is simply the minimum number of state transitions that separate two states in which a message is emitted on the output channel in question. Currently the value of x is specified manually by a programmer as they develop the code for each process.)

Source nodes in the process graph (those with no input edges) represent processes in the external environment (*i.e.*, devices). The output channels of these nodes are labeled with constants (symbolic or numeric) that indicate the maximum message transmission rate on the channel. For example, a rate might indicate the maximum rate with which a particular type of interrupt is expected to arrive. The remaining edges in a process graph can be labeled with a transmission rate in the obvious manner. Starting from the source nodes, the transmission rate functions can be evaluated in topological order and each edge in the process graph can be labeled with an actual transmission rate. This rate will be either a numeric value or an expression containing symbolic constants. The precise semantics of message passing are defined in terms of this rate. If a process accepts messages from a channel with a transmission rate r, then the message must be consumed within $1/r$ time units of its arrival. Since in general a server has no knowledge as to when its clients will send their next message, these semantics are required to ensure that messages are consumed in real-time under all circumstances.

Processes that receive messages from multiple clients are treated as multiple instances of processes that receive messages from a single client. For example, if a process S is a server for multiple processes, then S will consume a message from each client C_i before C_i sends its next message. If S receives messages from n clients then S is logically treated as a server with of n identical copies of process S's code. Each copy of S receives messages from a single client and consumes them in real-time. (The one process per client is a standard architecture for client/server systems. Our scheme differs in that our processes will be long lived and process multiple service requests.)

The key point is that the semantics of message passing apply separately to each communication channel in a process graph. The rate at which a server *S* consumes messages from a client *C* depends only on the rate at which *C* produces messages; independent of whether or not server *S* receives messages from clients other than *C*. From the point of view of a client, the rate at which a client's requests get serviced depends only on the rate at which *it* emits messages. Servers that receive messages from multiple clients do, however, introduce additional constraints on the implementation of message passing. If a server *S* receives messages from multiple clients, then to ensure the consistency of the function that *S* performs, the consumption of any client message by *S* may not be interrupted by processing of any other message at *S*.

For many applications, the temporal coupling of all message senders and receivers is unnecessary and often undesirable. Our programming system provides a facility for *non-time constrained communication* based on shared memory. Shared memory is referred to syntactically as a *data repository*. A data repository encapsulates shared data and exports a set of entry routines for accessing and manipulating the data. Processes invoke these routines via a procedure call. Like a monitor, data repositories provide mutually exclusive access to the data they encapsulate. Data repositories are also useful for sending large messages efficiently. Rather than sending a large structure in a message, a programmer may create a data repository to store the contents of such messages. Sending and receiving processes then may (safely) operate on the same copy of the data. In this case the physical message that is sent from a sender to a receiver is simply a synchronization signal. This style of interaction can eliminate the need for copying large amounts of data from a sender to a receiver.

2.1 Example: A videoconferencing system

We have used the discipline outlined above to design and implement a videoconferencing system [4]. The video conferencing application runs as an application in user space. Figure 1 shows a simplified version of the design of the "capture" portion of our video conferencing system. This is the portion of the system that acquires audio and video data from the respective hardware subsystems, compresses the data in software, and packages the data for transmission across a network. The application consists of 5 processes (represented by circles) and 1 data repository (represented as a double circle). Arrows indicate message channels, bold circles represent hardware devices (and their interrupt handlers). (For simplicity, the processes that control the network interface are not shown.)

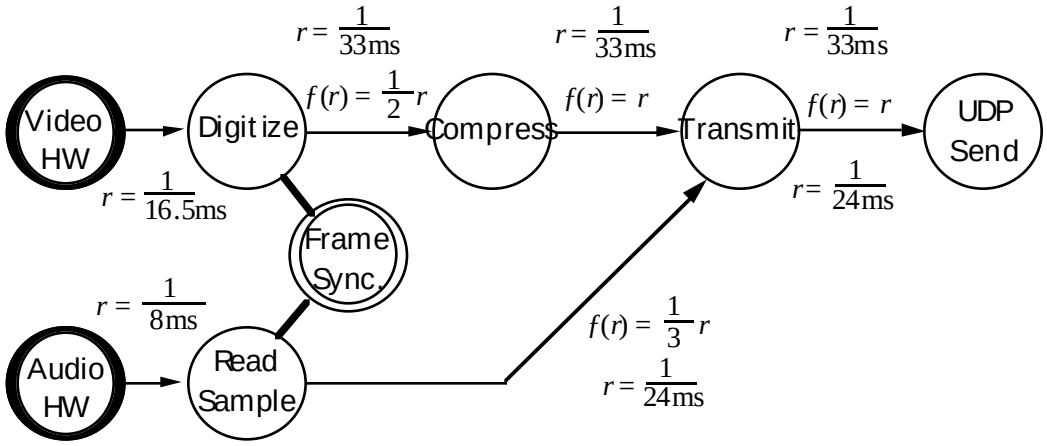

Figure 1: Process graph for the capture application, showing transmission rate functions and actual worst case transmission rates.

Each video frame propagates through a three stage pipeline: digitization of the video frame, compression of the digitized image, and transmission of the compressed data over the network. The results of a stage in the pipeline are communicated to later stages through messages. The capture portion of the application is controlled by interrupts from the audio and video subsystems. The video system generates a *vertical blanking interrupts* (VBI) after each half of a frame of video has been scanned (assuming interlaced video). This occurs once every 16.7 ms. Since one frame of video is generated for every two VBIs, the digitize process's transmission rate function is $f(r) = 2^{-1}r$. For the rest of the video pipeline, each process (in the worst case) outputs a message every time it receives a message. The audio system continually digitizes the audio source and interrupts the system at specified intervals (every 8 ms. in our case). The audio process ("Read Sample") collects sequences of audio samples before transmitting them. In the worst case (in terms of maximal transmission rate), it always delivers at least three samples at a time to the network interface. The audio and video processes are not synchronized, however, they must know about each other's progress so that their samples may be played in sync at the receiver. This coordination is achieved via a data repository that issues pairs of tickets (sequence numbers) to the audio and video processes. This function could be accomplished with another server, however, for efficiency, we use shared memory.

We have implemented a simple real-time UDP/IP server as a pair of processes. The server receives messages from both the audio and video processes and processes them within their respective time frames.

2.2 Discussion

This programming discipline allows us to succinctly express the obvious timing constraints of the videoconferencing system: that every audio and video sample generated by the hardware will be delivered to the network interface. This follows by a transitivity property and the fact that all communication occurs in real-time. Moreover, as described below, it is possible to implement this discipline such that it is further possible to bound the end-to-end latency of a message flow, *i.e.*, the latency for a (series of) messages to propagate from a source node in the graph to a sink node. If r is the actual worst case transmission rate for a data stream (a path through the process graph) at the sink process, then assuming all processes simultaneously transmit messages at their worst case rate (an occurrence that in general is impossible to guarantee), then data will flow from source to sink along that path with a worst case end-to-end latency of no more than $1/r$ time units. For example, for the video path, it is the case that frames of video are delivered to the network interface within 33 ms. of being generated. For audio samples the bound is 24 ms.

Note that these bounds are solely a function of the worst case transmission rate for a stream at the sink node. In particular, if, for example, the compression process were broken up into several processes, so long as they all processed data at the same rate, the worst case end-to-end latency of the stream would be unaffected (assuming the extra overhead induced by splitting apart the original process does not make the entire system infeasible). Because of this property, we claim that this methodology makes many issues of structure and real-time performance orthogonal concerns. Real-time performance is solely a function of the transmission rate functions (which is term a function of algorithms used in processes) and not a function of how systems are decomposed into processes.

A second point to discuss concerns the transmission rate formalism itself. While the formalism can be somewhat cumbersome, it is a useful tool for making application structure explicit and can aid in the feasibility analysis of a system. For example, the decrease in transmission rate for the video stream beyond the digitize process occurs because of branching logic in the digitize process. Part of what the process does is count VBI interrupts. Only when the count is an even number does the process emit a message. Capturing this fact in the transmission rate makes it explicit that the compression function in fact only executes once every 33 ms. Without this information (*e.g.*, if the compression function were combined with the digitize process) one would have to

assume that in the worse case the cost of compression was incurred every time the digitize process executed. This would lead to an overly pessimistic analysis.

3. Implementation

In reality, the programming discipline we have developed is a thin (but useful) veneer of programming abstractions on top of the scheduling model presented in [8]. There, a real-time system was modeled as a set of sporadic tasks that share a set of serially reusable, single-unit, software resources. A sporadic task is a sequential program that is partitioned into a sequence of disjoint phases. A phase is a contiguous sequence of executable statements that together require exclusive access to a resource. A resource is a collection of data that are shared between two or more tasks (a special null resource is used to model code that does not access any shared resource). Sporadic tasks are further characterized by a worst case execution cost c, and a period p that represents that minimum inter-arrival time of invocations of the task.

To correctly implement this model of sporadic tasks, one must schedule the tasks such that (1) when a task is invoked, it must complete execution within p time units (*i.e.*, before the next invocation of the task can possible occur), and (2) whenever a task commences a phase that requires exclusive access to a resource, no other task may access the same resource until the first task completes its current phase. In [8], an optimal scheduling algorithm based on earliest deadline first scheduling was developed. This is the algorithm we employ.

This model of sporadic tasks is used to implement the programming discipline as follows. Message channels are implemented as sporadic tasks. The executable code for the task is taken from the process that receives messages on the channel. The periods of each task correspond to the inverse of the rate at which messages are transmitted on the channel. Note that if every process receives messages from at most one other process then implementing a channel as a sporadic task is equivalent to implementing a process as a sporadic task. However, when a process receives messages from multiple other processes, it must process each message before a unique deadline specified by the message's sender. This constraint is achieved by associating tasks with channels rather than processes.

Data repositories are implemented as resources. Resources are also used to implement the mutual exclusion constraints on the processing of messages at processes that receive messages from multiple sources. If two tasks are created from the same process (*e.g.*, a

process receives messages from two senders) then an artificial resource (*e.g.*, a mutual exclusion lock) is created to ensure that the two tasks never execute simultaneously.

One practical complexity that arises in applying the formal model to the implementation of the programming discipline is that the real world does not always meet the assumptions of the model. Consider a task's period parameter. The formal model assumes that consecutive invocations of a sporadic task with period p are separated by at least p time units. Tasks that are invoked in response to events generated by devices may not satisfy this property. For example, when video frames are transmitted across a network they may be delayed for arbitrary intervals at nodes in the network. Therefore, even though video frames are, in theory, generated periodically, their arrival pattern at a conference receiver may be highly irregular. One solution to this problem is to simply buffer video frames at the receiver and release then at regular intervals to the conference application (although this begs the question of how one implements and models the real-time tasks that perform this buffering process). This approach is undesirable because it is difficult and tedious to implement correctly and because it will increase the acquisition-to-display latency of each video frame (and latency is a primary measure of conference quality).

Our approach is to alter the formal model to account for the fact that there may be significant "jitter" (deviation) in the inter-invocation time. We extend the definition of a sporadic task as follows. We have the same characterization of a sporadic task as before, however, we make no assumptions about the spacing in time of invocations of a sporadic task. If a task with "period" p is invoked at time t the task is scheduled (placed into the run queue) with a deadline of time $\mathrm{MAX}(t, d_{last}) + p$, where d_{last} is the deadline of the previous (and possibly still outstanding) invocation of the task. That is, we no longer require that invocations of a task be separated in time — only that its deadlines be separated by at least p time units.

It turns out that the results developed for the formal model of sporadic tasks we are using are insensitive to this change. That is, allowing "early release" of tasks does not change the basic feasibility conditions or optimality of previously developed algorithms. A proof of this result for a simpler model of sporadic tasks is presented in the appendix. (Although the result holds for the more general resource model, the proof is far more tedious and no more enlightening than the result we present.)

With early release, task invocations that arrive less than *p* time units apart are now "buffered" in the system run queue. That is, multiple requests for the same task may now be present in the run queue. This is not a problem, however, since all requests have monotonically increasing deadlines (and since we are using deadline scheduling), there will never be two invocations of the same task executing at any time. (Tasks with earlier deadlines *must* complete before tasks with later deadlines may commence.) Moreover, this scheme is trivial to implement.

4. Realization in Real-Time Mach and Performance

The basic Mach programming primitives are *tasks*, *threads*, *ports*, *messages*, and *memory objects*. Tasks are collections of systems resources including a paged virtual address space and a port name space (and port rights). A thread is the basic unit of program execution. They are points of control flow within tasks. Each thread may belong to at most one task. A port is a one-way communications channel between a client (message sender) and server (message receiver). Ports are implemented as message queues that are protected by the kernel. Message are typed collections of data that are passed between threads. A memory object is a storage object that is backed by secondary storage.

RT-Mach is a variant of the Mach designed to simultaneously support both real-time and non-real time applications [15]. There are three major enhancements over standard Mach. The RT-Mach thread model has been extended to included both non-real-time threads, real-time periodic threads, and real-time aperiodic threads, and, to support these new thread models, the scheduling policy is selectable by user applications. Most of the common policies, including round-robin, rate-monotonic, and earliest-deadline-first, are available. Second, RT-Mach provides a new IPC mechanism that eliminates priority inversions by executing server tasks at their client's priority. Finally, RT-Mach introduces a new resource allocation mechanism. The implementation uses a consistent and integrated approach to avoid direct and cascaded priority inversion.

The scheduling model used in RT-Mach is an extension of the Integrated Time-Driven Scheduler for the ARTS kernel. The goal is predictable, flexible, and modifiable scheduling for both the hard-real-time and soft-real-time threads as well as the non-real-time threads. To achieve this goal, the scheduler is cleanly divided into scheduling mechanism and scheduling policy. The scheduling mechanism is fixed priority scheduling so any scheduling policy that can map its ordering to a priority is possible. The novelty is that the scheduling mechanism can be dynamically changed by any user

application with the necessary permissions. This allows considerable flexibility in scheduling of both real-time and non-real time threads.

We implemented sporadic tasks, resources (as described in the previous section), and the optimal earliest deadline first algorithm starting from the implementation of periodic tasks in RT-Mach. This was relatively a straightforward exercise.

Measurements of the conferencing system indicate that the system is feasible (we have no tools for analyzing execution times hence this part of the analysis was done in an *ad hoc* manner) with a processor utilization of just under 60%. This means that all data generated by the application is delivered to the network interface in real-time. Moreover, for the capture application, all video frames are delivered with an end-to-end latency as advertised in the previous section.

The timing for the capture half of the conferencing system is relatively deterministic. This is not the case for the display application (essentially the inverse of Figure 1). Here all data arrived was delivered to the display, however, the end-to-end latency (network arrival to display) was more variable. However, because of our use of the early release mechanism to deal with network jitter, latency always stayed within 33 ms.

To illustrate the effects of the early release modification to the sporadic tasking model we compared the performance of the display application with and without the early release. Table 1 summarizes these results. In the experiments, the system was loaded at various levels with other real-time (periodic) processes. The level of load reported in the table represents only the artificially generated load as measured in terms of CPU utilization. (The total CPU utilization is the value in the table plus approximately 57%.) Data values in the table are the largest observed end-to-end latency.

Three experiments were run. In the first, data was delivered to the application with essentially no jitter — data arrived once every 33 ms. The second and third experiments had variable (but reproducible) jitter wherein arrivals occurred nearly simultaneously and spaced out as far as 60 ms. Table 1 indicates that the early release strategy can significantly reduce end-to-end latency in the face of bursty invocations of tasks.

CPU utilization due to artificially generated load

	0%	10%	20%	30%	40%
No jitter	18.48	20.98	22.98	24.98	26.97
Early Release	18.98	21.14	24.02	28.46	31.46
Buffered	19.08	27.92	74.42	121.01	206.32

Table 1: End-to-end latency for various levels of background load and jitter.

5. Conclusions

In this work we implemented a programming discipline based on a sporadic tasking model. The model allows one to express simple timing constraints that ensure high-level properties such all inputs are processed in real-time and end-to-end latencies are minimized, are realized. We have also demonstrated that a formal model of sporadic tasks was implementable (with small modifications). Along the way we learned that the theory of sporadic tasks is insensitive to generous assumptions about the repetitive nature of the tasks, only the deadlines at which those tasks must complete execution.

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., *Kernel Support for Live Digital Audio and Video*. Computer Communications, Vol. 16, No. 6 (July), pp. 388-395.

[6] Jeffay, K., Stone, D.L., *Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems*, K. Jeffay, D.L. Stone, Proc. 14th IEEE Real-Time Systems Symp., Durham, NC, December 1993, pp. 212-221.

[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] Mercer, C.W., Savage, S., Tokuda, H., *Processor Capacity Reserves: Operating System Support for Multimedia Applications*, IEEE Intl. Conf. on Multimedia Computing and Systems, Boston, MA, May 1994, to appear.

[12] Mok, A.K.-L., *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*, Ph.D. Thesis, MIT, Department of EE and CS, MIT/LCS/TR-297, May 1983.

[13] Nakajima, T., Kitayama, T., Arakawa, H., Tokuda, H., *Integrated Management of Priority Inversion in Real-Time Mach*, Proc. 14th IEEE Real-Time Systems Symp., Durham, NC, December 1993, pp. 120-130.

[14] Nakajima, J., Yazaki, M., Matsumoto, H., *Multimedia/Realtime Extensions for Mach 3.0*, USENIX Workshop on Micro-Kernels and Other Kernel Architectures, Seattle, WA, April 1992, pp. 161-175.

[15] Tokuda, H., Nakajima, T., Rao, P., *Real-Time Mach: Toward a Predictable Real-Time System*, Proc. of the USENIX Mach Workshop, October, 1990.

Appendix

Here we formally state the model of sporadic tasks used in our implementation and give feasibility conditions for a set of tasks when preemption is allowed at arbitrary points. The conditions are identical to those previously developed for other models of sporadic (and periodic) tasks.

System Model

We consider a real-time system as being composed of a set of *randomly released tasks*. A task is a sequential programs that is repeatedly *invoked* by occurrences of *events* (*e.g.*, device interrupts or the arrival of a message). In the real-time systems literature, two commonly studied paradigms of event occurrences are *periodic*, in which events are generated every p time units for some constant p, and *sporadic*, in which events are generated no sooner than every p time units for some constant p. Here we consider a more general model wherein event occurrences occur at *random.*

Specifically, define a task T as a logically infinite sequence of identical *jobs* J_j. A job is an instance of the sequential program that is task T. The j^{th} job of task T, written J_j, is released at time r_j and must complete execution before a well-defined deadline d_j. More formally, a *randomly released task* T is specified by a triple (J, c, p). J is an infinite sequence of jobs $<J_1, J_2, \ldots>$ of task T. A job J_i is specified by a pair (r_i, d_i) where r_i is the release time of J_i — the earliest time at which J_i may commence execution — and d_i is the deadline of J_i — the time by which the execution of J_i must complete. The task parameter c is the maximum amount of processor time required to execute a job of task T to completion on a dedicated uniprocessor. We assume that jobs are released at random but in sequence, *i.e.*, that $r_i < r_j$ if $i < j$. Moreover, we require that job J_i must complete execution before job J_{i+1} may commence execution. The deadline of a job is defined by the following recurrence relation:

$$d_i = \begin{cases} 0 & \text{if } i = 0 \\ MAX(r_i, d_{i-1}) + p & \text{if } i > 0 \end{cases}$$

where p is a parameter of the task.

We assume time is discrete and clock ticks are indexed by the natural numbers. Task invocations occur at clock ticks; parameters c, p, and r_i are expressed as integer multiples of the interval between successive clock ticks.

In our model, at any point in time an arbitrary number of jobs of a task may be pending (released but not yet completed), however, the deadlines of consecutive jobs are separated by at least p time units.

We define a task system as a set of n randomly released tasks, $(J_1, c_1, p_1) \ldots (J_n, c_n, p_n)$. A task system is *feasible* if it is possible to schedule the tasks such that each job of each task completes execution at or before its deadline.

Note that our model of a randomly released task subsumes the more commonly studied models of both periodic tasks (as defined in [10]) and sporadic tasks (as defined in [12, 8]). For example, a periodic task is simply a randomly released task T wherein all jobs are released every p time units (*i.e.*, for all $i > 0$, $r_i = (i - 1)p$).

Analysis of Randomly Released Tasks

Our analysis is based on Liu and Layland's original analysis of preemptive periodic tasks. It turns out that the feasibility conditions for randomly released tasks are the same as for periodic tasks. Thus the fact that tasks are released randomly does not effect feasibility. The key constraint is that the deadlines have a minimum separation in time.

Theorem 1: A set of randomly released tasks $\tau = \{(J_1, c_1, p_1), \ldots, (J_n, c_n, p_n)\}$, where for all i, $1 \le i \le n$, $J_i = <(r_{i1}, d_{i1}), (r_{i2}, d_{i2}), (r_{i3}, d_{i3}), \ldots >$, will be feasible if and only if

$$\sum_{i=1}^{n} \frac{c_i}{p_i} \le 1. \qquad \text{(A.1)}$$

The interpretation of (A.1) is as in [10], that is, c_i/p_i is the fraction of processor time consumed by jobs of task T_i over the lifetime of the system — the utilization of the processor by T_i.

Proof: ($\Rightarrow$) Consider the interval $[0, L]$ for any $L > 0$. In $[0, L]$ at most $\lfloor L/p_i \rfloor$ jobs of task T_i will have to execute to completion, requiring at most $\lfloor L/p_i \rfloor c_i$ units of work be available in $[0, L]$. If task T_i is released randomly in $[0, L]$ then it is possible for a job of T_i to miss a deadline if $L < \lfloor L/p_i \rfloor c_i$ (for example, whenever jobs are released at most every p_i time units). Therefore, task T_i can be guaranteed to not miss a deadline only if

for all $L > 0$, $\lfloor L/p_i \rfloor c_i \leq L$. Applying this argument to all tasks in τ, it follows that τ will be feasible only if for all $L > 0$,

$$\sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i \leq L.$$

It is straightforward to show that this is equivalent to (A.1) (see [6] for a proof).

($\Leftarrow$) To show the sufficiency of (A.1) we show that the preemptive *earliest deadline first* (EDF) scheduling algorithm can schedule a set of randomly released tasks if the tasks satisfy (A.1). This is shown by contradiction.

Assume that τ satisfies (A.1) and yet a job of a task in τ misses a deadline at some point in time when τ is scheduled by the EDF algorithm.

Let t_d be the earliest point in time at which a deadline is missed and let t_0 be the later of:

- the end of the last interval prior to t_d in which the processor has been idle (or time 0 if the processor has never been idle), or,
- the latest time prior to t_d at which a job of a task with deadline after time t_d executes (or time 0 such a job does not execute prior to t_d).

By choice of t_0, no job with deadline after t_d executes in the interval $[t_0, t_d]$. Therefore, if the EDF scheduling algorithm is used, then the processor demand in the interval $[t_0, t_d]$, is at most

$$\sum_{i=1}^{n} \left\lfloor \frac{t_d - t_0}{p_i} \right\rfloor c_i .$$

Since the processor is fully used in the interval $[t_0, t_d]$, and since a deadline is missed at time t_d it follows that

$$\sum_{i=1}^{n} \left\lfloor \frac{t_d - t_0}{p_i} \right\rfloor c_i > t_d - t_0$$

$$\sum_{i=1}^{n} \frac{t_d - t_0}{p_i} c_i > t_d - t_0$$

$$\sum_{i=1}^{n} \frac{c_i}{p_i} > 1.$$

However this contradicts our assumption that τ satisfies (A.1) for all L. Hence if τ satisfies (A.1) then a deadline driven scheduler will succeed in scheduling τ. It follows that satisfying (A.1) is a sufficient condition for feasibility. □

The proof of Theorem 1 also establishes the optimality of the deadline driven scheduling algorithm for scheduling randomly released task sets as the condition that is necessary for feasibility is sufficient for ensuring the correctness of the EDF algorithm.