SUPPORT FOR REAL-TIME COMPUTING WITHIN GENERAL PURPOSE OPERATING SYSTEMS

Supporting Co-Resident Operating Systems*

Gregory Bollella *Kevin Jeffay*

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

Abstract: Distributed multimedia applications are typical of a new class of workstation applications that require real-time communication and computation services to be effective. Unfortunately, there remains a wide gap between the development of real-time computing technology in the research community and the deployment of real-time solutions in commercial systems. In this work we explore technology for allowing two operating systems, a general purpose operating system and a predictable real-time kernel, to co-exist on the same hardware. We discuss the problems of *multiplexing* shared devices and *partitioning* shared data structures to accommodate two operating systems, and present a CPU executive that allows the IBM Microkernel (a derivative of the Mach microkernel) with an OSF/1 server to co-exist with a simple real-time kernel we have built. We also extend the traditional theory of scheduling periodic tasks on a uniprocessor to accommodate the case where a real-time kernel is allocated only a fraction of the total CPU capacity.

1. Introduction

Recent advances in high-speed networks, workstation processors, and digital audio and video acquisition/compression/display hardware have led to tremendous interest in the problems of supporting the real-time computation and communication requirements of distributed multimedia applications [1-7]. However, although much research has been performed on the design of real-time operating systems, and much is on-going in the distributed multimedia domain, the vast majority of computers in use today use operating systems that provide no support for real-time computing beyond allowing tasks to manipulate their execution priority. Operating systems that (1) allow programs to specify their real-time performance requirements and (2) guarantee these requirements are met, have been, and are still, largely confined to academic and industrial research laboratories.

To date there have been two identifiable approaches to marrying real-time and non-real-time technologies. First, there have been several attempts to add real-time features such as periodic tasks and priority-inversion-free interprocess communication mechanisms into existing general purpose operating systems such as Unix and Mach [1, 3, 4]. The second approach is to design a general purpose but dynamically configurable operating system kernel that is capable of accommodating user or application specific process, scheduling, and memory management modules [11]. Here we investigate a third approach: supporting co-resident operating systems. Our interest lies in developing a methodology for allowing existing, well-understood real-time systems technology to co-exist with commercial, general purpose operating systems to support applications such as desktop multimedia conferencing. Thus our emphasis is not so much on the development of new real-time systems technology *per se*, but rather on developing a vehicle for the incorporation and use of existing technology in conventional desktop operating systems.

Our approach is to partition the central processor and other system resources into two virtual machines — a machine running a largely unmodified general purpose operating system and a machine running a real-time kernel — and multiplex accesses to the physical hardware by the virtual machines. By allowing the co-existence of two operating systems, this approach allows a clean separation of concerns between the requirements and desired solutions for non-real-time, general purpose computing services and those for real-time services. If we are successful, one should be able to "mix-and-match" general purpose operating systems and real-time kernels to suit the needs of the system user.

Our approach is based on a set of simple, small, executives that execute on the bare machine and allow a general purpose operating system and a real-time kernel to

* Supported in part by grants from the IBM and Intel Corporations.

In: Proceedings of the IEEE Real-Time Technology and Applications Symposium, Chicago, IL, May 1995, pp. 4-14.

share the hardware in such a fashion that (1) the real-time kernel executes in a predictable manner so that it is possible to analyze the conditions under which real-time tasks will be guaranteed to be feasible, and (2) the general purpose operating system can function correctly with few modifications. We have identified two generic technologies that are required to achieve these goals. The first is a facility for multiplexing the co-resident operating systems' execution on all shared devices. The second is a scheme for partitioning all shared, serially reusable resources such as disk sectors, network buffers, screen pixels, *etc*., between the operating systems.

Presently we have prototype executives for the CPU and a few other devices. The executives multiplex accesses to each shared device and partition device resources so that the above two goals are met. The most complex executive is that for the CPU. It operates by executing the general purpose system and preempting its execution at precise, constant intervals to execute the real-time system. Thus at a high-level, our overall system is similar to a cyclic executive [8] that alternates between the execution of two programs that happen to be other operating systems. Alternatively, this work could be viewed as an implementation of a coarse grained pure-processor-sharing scheduling algorithm [9] in which two programs (two operating systems in our case) make forward progress at precise rates. Other related works include the design of IBM's VM operating system [10] and the general time-division multiplexing techniques used in bus and communication protocols. Other attempts to directly embed real-time systems technology into general purpose operating systems include the HeiTS network communications system [12]. In HeiTS, a real-time kernel is embedded into a network device driver and used to maintain real-time communication of multimedia data across a network.

While our overall approach bears a strong resemblance to those followed in the design of several other systems, we believe this work contributes a useful, general methodology for solving the pragmatic problem of allowing existing (and thus future) real-time and non-real-time systems technology to co-exist on the same machine. This approach provides a clean separation of real-time and non-real-time concerns and makes possible an analysis of the feasibility problem for periodic tasks that are allocated a fraction of the processor's capacity. This analysis contributes to our knowledge of real-time resource allocation problems for architectures such as cyclic executives and the others mentioned above.

In the remainder of the paper we describe the design and implementation of our executive in more detail. Our executive is capable of executing the IBM Microkernel (a derivative of the Mach microkernel) with an OSF/1 server in parallel with a simple real-time kernel we have developed. Section 2 describes our overall framework for supporting co-resident operating systems and discusses the multiplexing and partitioning problems in detail. Section 3 describes the implementation of our executive and assesses both its performance and the impact of the executive and real-time kernel on the performance of the microkernel. We also demonstrate the degree to which the executive can ensure a precise execution rate for the real-time kernel. Section 4 describes our real-time kernel and Section 5 shows how the existing theory of scheduling periodic tasks on a single processor can be extended to incorporate the fact that the real-time kernel only executes periodically. Section 6 discusses some of the fundamental design issues and limitations of our approach. We conclude in Section 7 with some plans for extending this work to other general purpose operating systems such as Windows and OS/2.

2. Supporting Co-Resident Operating Systems

Two basic low-level operations required to support co-resident operating systems are *multiplexing* and *partitioning*. Multiplexing is the allocation of consumable resources (*e.g.*, CPU processing time, network bandwidth, *etc.*) to the co-resident operating systems. Every shared device in the system that represents a consumable resource requires multiplexing. Partitioning is the allocation of serially reusable resources (*e.g.*, buffers, address spaces, screen pixels, *etc.*) to the co-resident operating systems. Every shared device in the system that contains serially reusable resources requires partitioning. These operations are described in greater detail below. In the following we use the abbreviations RTK and GPOS to refer a real-time kernel and general purpose operating system respectively.

2.1. Multiplexing

The multiplexer is a small executive that controls when and for how long each operating system may use a shared device. There are two fundamental problems in the design of a multiplexer. First, the multiplexer must precisely allocate (schedule) execution time on a shared device to each operating system. This implies that the multiplexer must be able to gain control of the device at well-defined, precise times, and maintain control of the device for well-defined, precise durations. The acquisition and allocation process must occur at precise times if the RTK is to guarantee real-time services on the device.

In general, the solution to the device acquisition and control problem is to either provide a mechanism such as an interval timer to insure that the multiplexer is executed

at (precise) periodic intervals, or provide a mechanism such as an interrupt to ensure that the multiplexer is executed whenever the RTK requests the service of the device. In either case, once a multiplexer commences execution, it must logically execute with all interrupts disabled until it has allocated the desired units of the consumable device resource to the appropriate operating system. For example, for the CPU multiplexer described below, we use an interval timer to invoke the multiplexer at regular intervals and trap all operating system calls that manipulate the current interrupt mask to ensure the multiplexer's timer interrupt is never disabled. Moreover, to ensure that the RTK executes without interference from the GPOS, all interrupt sources currently in use by the GPOS are disallowed from interrupting the RTK. (In general, the converse is not required to be true.) As illustrated in Section 3, our experience to date indicates that the device acquisition and control problem is solvable without requiring detailed knowledge (*e.g.*, source code) of either operating system.

The second problem, and one that complicates solutions to the first, is that of maintaining the consistency of data structures that are used to control the operation of, and describe the current state of a device. Unless all device operations are atomic, critical sections will exist between the co-resident operating systems with respect to their individual use of the device. These critical sections must be discovered and locked so that one operating system cannot preempt the second's use of the device while the second is executing inside a critical section.

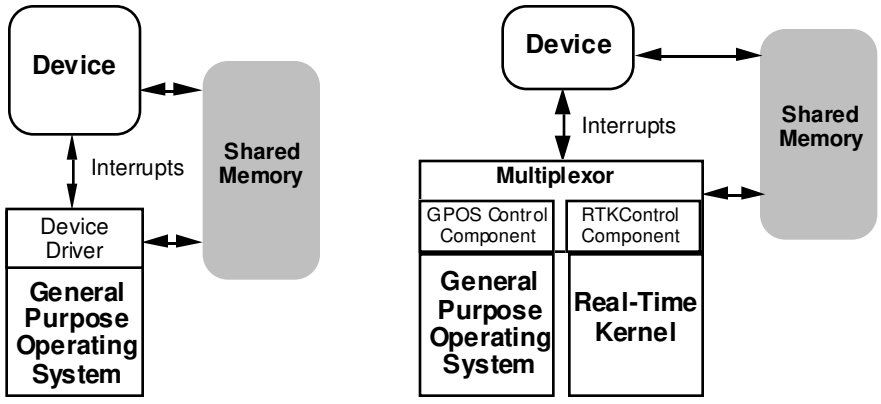
The critical section problem is not as easily solved as the acquisition and control problem. The ideal solution is to examine the source code for each operating system to discover precisely how it interacts with the device, however, this is clearly not always feasible. For peripheral devices this problem is manageable as often the both the abstract paradigm of a GPOS's interaction with a device, as well as the device manufacturer's paradigm of device interaction, are well documented in developers' guides. In general, the critical sections of interest often involve static memory locations or special instruction opcodes for manipulating special registers and thus critical sections can be located by searching the object code and the binary can often be modified without access to the source. In designing our current CPU multiplexer, however, we relied on the source code for the IBM Microkernel to verify we had locked all inter-operating system critical sections.

2.2. Partitioning & Device Management

Most peripheral devices such as the screen, keyboard, mouse, disks, audio/video devices, and network adapters, typically contain serially reusable resources that must be shared between the co-resident kernels. These resources must be partitioned, either statically or dynamically between the two systems.

Our general framework for device management is illustrated in Figure 1. Each shared device has a multiplexer, a GPOS control component, and an RTK control component. The multiplexer is as described above. The GPOS and RTK control components implement the GPOS's and RTK's desired view of the device and are responsible for partitioning whatever shared resources exist on the device. The control components are, in essence, device drivers modified to operate in concert with each other.

The details of resource partitioning are necessarily device specific. For simple devices such as a memory mapped display (console), the GPOS control component can be the unmodified driver for the display and the RTK control component can be a simple user program executing on the GPOS. In this scenario, the RTK control component requests screen real-estate and instructs the GPOS's display manager to ignore the pixels allocated to the RTK compo-



a) Traditional organization. *b*) Proposed organization.

Figure 1: Logical device driver organization for co-resident operating systems.

nent. Processes within the real-time kernel are then free to write directly to the screen memory corresponding to these pixels. (Indeed, this scenario is precisely how many video display systems work since they cannot rely on the window system to display video data at the desired frame rate.) Moreover, in this case, since screen output is performed by writing to physical memory locations, and these operations are atomic, the multiplexer is trivial to construct since there are no inter-operating system critical sections.

Note that generic devices that are used only by the GPOS require no special effort to support. Their interrupts will only be serviced, and their device drivers will only execute when the GPOS executes. Thus the RTK and the CPU multiplexer need not have any knowledge of their existence. Devices that are used only by the RTK do not require a multiplexer but do require an RTK control component (*i.e.*, a device driver) and may require a GPOS control component. The GPOS control component may be required to handle interrupts from the device that occur during the execution of GPOS.

3. A CPU Executive and Its Performance

Here we describe the design, implementation, and performance of a CPU executive for an IBM PS/2 workstation. The executive multiplexes the CPU between the IBM Microkernel with an OSF/1 Unix server, and a simple generic real-time kernel we have developed. It also partitions two low-level system data structures between the two operating systems. The data structures are used by the hardware to realize tasks and address spaces. The IBM/OSF/1 system executes normally on top of the executive while the real-time kernel executes in a highly predictable manner. The following subsections describe the multiplexer and the partitioning of data structures, and demonstrate that the real-time kernel executes in a predictable manner.

In building the CPU executive we had access to the source code for the IBM Microkernel (but not for OSF/1) and used the code to verify our understanding of how to partition data structures. The design of the CPU executive required the combined addition and modification of less than 100 lines of assembler in the microkernel.

3.1. Hardware Description

The hardware used was an IBM PS/2 Model 95 workstation (66 MHz Intel 80486 processor). A number of integrated circuits on the motherboard are of particular interest to our implementation. There are two programmable interrupt controllers (PICs) (Intel 8259A peripheral support chips [13]), one programmable interval timer [13], and one real-time clock (RTC) (a Motorola MC146818A Real-Time Clock Plus Ram [14]). The PICs multiplex interrupts from peripheral devices and other circuits on the motherboard onto the single interrupt line of the CPU. In addition, the PICs provide a means to assign priorities to interrupt generators and a bit mask which can enable and disable each device. Each PIC has eight input lines and there are two cascaded PICs on the motherboard. Inside the CPU, interrupts from the PICs are enabled and disabled by setting or clearing the interrupt (IF) bit in the *i*486 flags register. Setting and clearing the IF bit is performed by an *i*486 machine instruction.

In addition to being able to keep real-time, the RTC also provides a programmable interval timer. The interval may be set to one of 16 values from 30.5 microseconds to 500 milliseconds. At the end of each interval the RTC will generate an interrupt. As described below, we use the RTC to provide periodic interrupts to the CPU executive at the (arbitrarily chosen) rate of one interrupt every 488.281 μs. We will refer to intervals of this duration as *slots*.

3.2. Multiplexer

The paradigm of CPU multiplexing we adopt is to alternate between executing the GPOS for mc_{nrt} contiguous time units, and executing the RTK to for mc_{rt} contiguous time units. Borrowing terminology from the cyclic executive literature [8], we call an execution of a GPOS followed by an execution of the RTK a *major cycle*. The execution of each operating system is referred to as either a real-time or a non-real-time minor cycle. The duration of a real-time and non-real-time minor cycle is given by parameters mc_{rt} and mc_{nrt} respectively. The length of a major cycle MC is thus $mc_{nrt} + mc_{rt}$. The logical behavior of a CPU multiplexer is then completely described by the pair (mc_{nrt}, mc_{rt}).

The multiplexer is driven by interrupts from the RTC. The multiplexer allocates the CPU to the co-resident operating systems in units of *slots*. Presently mc_{nrt} = 24 slots = 11.72 ms, mc_{rt} = 2 slots = .976 ms, and MC = 26 slots = 12.7 ms. Thus the GPOS receives approximately 92% of all processor cycles (minus overhead) and the RTK receives the remaining 8% (minus overhead). Both the size of a slot and the size of minor cycles are tunable parameters. The small slot and minor cycle sizes used here were chosen to stress the implementation and to ensure good response time for jobs of the GPOS.

In order to guarantee precise execution of the RTK, the multiplexer must ensure that its RTK clock interrupts are never disabled and that the time between the occurrence of a clock interrupt and the start of the execution of its handler is bounded. We can recognize all locations in the microkernel that manipulate the IF bit in the flags register

and modify the kernel so that interrupts remain enabled but configure the PICs so that only the RTC can generate an interrupt. Since the RTC interrupt is handled by the CPU executive, the microkernel is unaffected by this modification.

The implementation of the CPU executive itself requires that interrupts be disabled to enforce mutual exclusion inside critical sections within the executive. The maximum length of these critical sections is approximately 200 CPU cycles. On the hardware used here, each cycle is 15 nanoseconds, hence the maximum time by which the RTC interrupt can be delayed because of actions of the executive is approximately 3 μs.

The RTC interrupt handler is designed as an *i*486 hardware task with an *i*486 task descriptor in the interrupt vector table. When the hardware detects this type of descriptor on an interrupt, the complete state of the currently executing process is saved in a *i*486 Task State Segment (TSS) by the hardware. This type of interrupt handling mechanism is more costly than a typical interrupt, however, the RTC handler is implemented this way to minimize changes to the GPOS and to provide complete separation between the GPOS and the CPU executive. The RTC interrupt handler calls the CPU executive to determine which of three cases the executive may be in. The cases are: the start of a real-time minor cycle, the start of a non-real-time minor cycle, or the start of a slot in the middle of a minor cycle. If the system is in the middle of a minor cycle, the RTC interrupt handler resets the RTC and PICs, and calls the CPU executive dispatcher to resume the preempted task. If it is time to start a real-time or non-real-time minor cycle, the executive saves the state of the previously executing task and resumes the appropriate operating system.

This implementation of the multiplexer required only two basic modifications to the IBM microkernel. First, as described above, all operations on the *i*486 flags register involving the IF bit had to be trapped and emulated to ensure the RTC interrupt is never disabled. These operations occurred in only a few places such as the first level interrupt handler of the microkernel and were easy to locate. Second, the microkernel code that performs a task switch had to be treated as a critical section as it manipulated data structures used by the hardware task switching mechanism. Although the IBM Microkernel does not use the hardware task switch mechanism, the CPU executive does. Hence these data structures must always be in a consistent state in order for the CPU executive to use the hardware mechanisms for saving and restoring GPOS state. This was accomplished by disabling all interrupts during execution of the microkernel task switching routines. The microkernel was, of course, already disabling interrupts for this routine; we simply had to ensure our interrupt disabling emulation code did not execute in this one case.

3.3. Partitioning of System Data Structures

There are three *i*486 data structures that are used by virtually all *i*486 operating systems and thus must be partitioned between the GPOS and the RTK. These are the interrupt vector table, the global descriptor table and the TSS. Since we are using an existing GPOS, the GPOS currently has these data structures mapped in its kernel's address space. The problem here is to discern the mapping and partition the structures.

The interrupt vector table is an array of *i*486 memory descriptors in the GPOS kernel. After the *i*486 acknowledges an interrupt from the PICs, the master PIC places an 8-bit value on the bus. This value is both the interrupt number and an offset into the interrupt vector table. The PICs are initialized with the value to use as the interrupt number for each of the 16 devices that may be attached. The memory descriptors in the interrupt vector table contain information about a segment of memory, including its virtual address in either kernel or user space, permission values, and type information for the object residing at that memory location. The partitioning of this table must allow the CPU executive to install an entry in the table for the RTC interrupt handler. The IBM Microkernel did not use the RTC and thus that entry in the interrupt vector table was free. In general, for devices that are not shared between operating systems or the CPU executive, the partitioning of the interrupt vector table is straightforward. For devices that are shared, such as a disk, the partitioning must occur at a higher-level such as in the device driver (see Figure 1).

The global descriptor table (GDT) also holds a number of *i*486 memory descriptors, however, unlike the interrupt vector table, it can hold descriptors for data segments as well as code segments. The *i*486 descriptor table contains several thousand entries, however, less than 10 are used by the IBM Microkernel. We used descriptors in the GDT to identify RTK data and code segments. This was necessary because the IBM Microkernel used the values of the code and data segment offsets in the GDT to identify the type of code (user or system) currently executing and made various internal decisions based on that determination. Thus, the GDT itself was partitioned and that partitioning resulted in the partitioning of the microkernel's address space. Although the partitioning of the GDT and the microkernel address space was needed for this particular implementation, it is not necessary in general.

The TSS is required by the *i*486 for two reasons. It holds the address of the stack an interrupt handler must use when it is interrupting a user process and it contains a per-

mission mask that is checked when a user process access an IO port. The IBM Microkernel used only those two fields in the TSS. The other fields of the TSS hold enough information to completely restore a process to running state after it has been swapped out. The CPU executive used the unused fields in the TSS for just this purpose.

3.4. Performance

There are two primary performance issues of interest. The first concerns the CPU executive's ability to reliably invoke the RTK at periodic intervals. The second is a measurement of the overhead of the CPU executive and the performance degradation suffered by the GPOS.

The CPU executive is designed to gain control of the machine at the start of every slot (once every 7.8 ms in these experiments). The RTK can predictably execute tasks in real-time if and only if it is guaranteed to execute at well-defined times and for precise durations. Thus, an important measure of the predictability of the executive and therefore of the RTK, is the observed deviation in slot duration. Figure 2 shows a histogram of the observed deviations in slot times from 7.8 ms. This data was generated for a "quiet" system in which no user processes were running on either the GPOS or RTK, and for a "busy" system in which a copy of the IBM Microkernel was being compiled from a local disk, telnet and ftp sessions were active, and the RTK was executing real-time tasks. As expected, the distribution of deviations is approximately symmetrical since if slot i is lengthened by t μs, slot $i + 1$ is typically shortened by t μs.

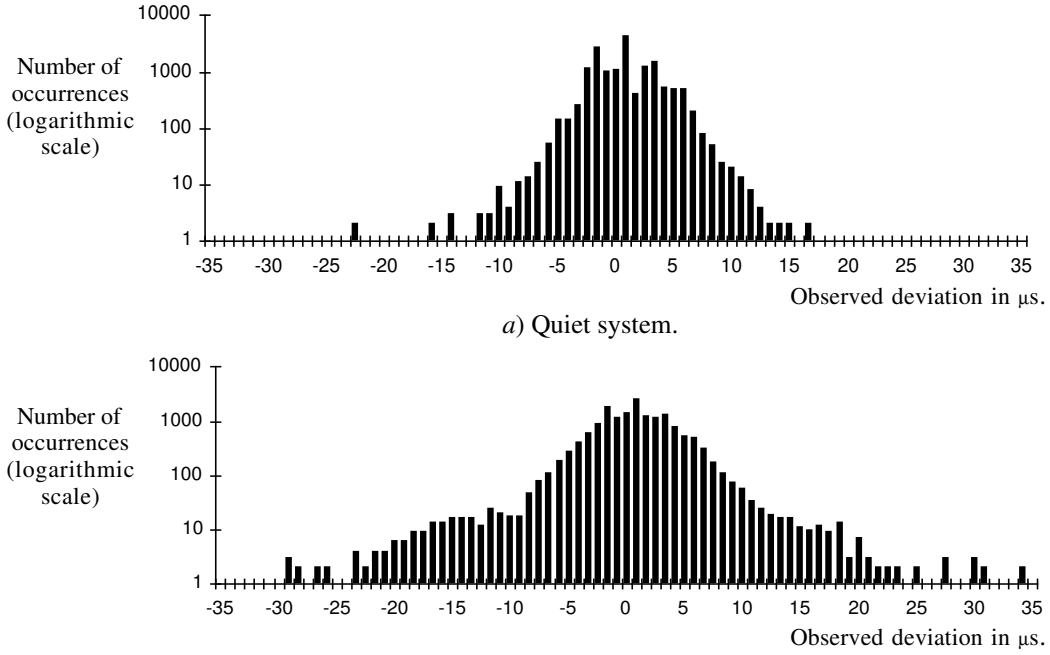
The worst empirically measured deviation in a slot duration was 34 μs or .04%. In general, deviation is due to a number of factors which are generic and thus transcend our particular system architecture. Deviation is caused by

- inter-operating system critical sections which require all interrupts be disabled (the dominant cause),
- critical sections in the CPU executive (resulting in deviations of ±3 μs),
- the CPU executive timer interrupt handler, and
- the measurement code itself (without the histogram data collection code, the largest observed deviation was 27 μs).

DMA memory cycle stealing also contributes to the variation as it inflates the execution times of the aforementioned critical sections.

Although the control of slot durations is imperfect, *it is paramount to realize that the phenomena that causes variable slot durations are generic and thus applies to all real-time kernels*. Indeed, for the Real-Time Mach kernel, recent measurements indicate that some of its timing and scheduling mechanisms exhibit considerably more variation than our CPU executive and RTK [25].

To assess the overhead of supporting co-resident kernels, we executed a computationally and I/O intensive task —



a) Quiet system.

b) Busy system.

Figure 2: Histogram of deviation in the duration of slots over a 2:08 minute interval.

the compilation of the IBM Microkernel — and measured how its execution time was affected by device executives and the RTK. Table 1 assesses the overhead of the CPU executive. Shown are the times required to compile the microkernel under a variety of environments. Experiment 1 consists of building the kernel on an unmodified version of the microkernel. The second experiment repeated the kernel build but on a version of the microkernel that had been modified to emulate interrupt enabling and disabling.

The second experiment had no CPU executive present hence it shows the increase in elapsed time due to the overhead of the interrupt enable/disable emulation. This overhead increased the build time approximately 2%.

The third experiment extended the second by adding in the CPU executive and generating RTC timer interrupts (every 488 μs) that are handled by the executive. In this experiment there is no RTK hence this illustrates the total overhead of the CPU executive. When the CPU executive runs it simply returns control to the GPOS. This overhead added an additional 9% to the kernel build time.

It is important to note that this increase is only due in part to the actual interruption of the GPOS and the execution of the CPU executive. When the GPOS is interrupted as frequently as it is, its main memory cache performance starts to suffer due to context switching. We believe disk throughput is also similarly effected. Additional experiments are on-going, however, we conjecture that as much as 4-5% of the increase in kernel build time from experiment 1 to 3 is due to memory cache degradation. The total observed increase in elapsed time for the kernel build imposed by the CPU executive alone is approximately 12%.

The final experiment executes real-time tasks inside the RTK to consume all cycles allocated to the RTK. This slows down the kernel build by 20%.

Table 2 shows the fraction of the processor consumed by each component of the CPU executive and by the real-time tasks. Given the size of the multiplexer time slot, the RTK should have had 7.7% of the processor, however, the real-time tasks were able to consume approximately 6.9% of the total processor cycles. This difference is explained by the fact that the RTK is subject to the same overhead as the GPOS for processing RTC timer interrupts. As RTC interrupt processing consumes 8.4% of the processor, the available processing time in each real-time (and non-real-time) slot is decreased by 8.5%. This reduces the expected maximum real-time task utilization to approximately 7.1%. The difference between the observed utilization and this bound is due to scheduling overhead within the RTK.

In summary, we are able to multiplex and partition CPU resources such that the RTK executes in a highly-predictable manner and the GPOS both functions correctly and is slowed down by only 12% above the degradation expected by the presence of the real-time tasks. All this is accomplished with a minimal number of modifications to the GPOS.

Table 1: Execution times in seconds for an IBM Microkernel build.

Experiment	Elapsed Time
1. Unmodified IBM Microkernel	180
2. CPU executive present but not running	184
3. CPU executive running, RTK idle	201
4. CPU executive running, 2 RTK tasks	216

Table 2: Measured overhead in % of all processor cycles consumed.

System Component	%
Interrupt disabling emulation	2.17
RTC Interrupt processing	8.46
Real-time task execution	6.94
Total of all non-GPOS code	16.67

4. A Simple Real-Time Kernel

Here we briefly describe our real-time kernel. As the kernel itself is rather mundane, the emphasis is on our paradigm of interaction and communication between GPOS tasks and RTK tasks. This paradigm is specific to the IBM Microkernel and the RTK was written with the knowledge that it would co-exist with the IBM Microkernel and thus takes advantage of its features.

The RTK consists of a bare machine kernel and a microkernel server that is used for communication between the IBM Microkernel and the RTK. The physical mechanism for the communication is memory that is shared between the microkernel and the RTK. Real-time tasks are written and compiled within the OSF/1 server that executes on top of the microkernel. A task is written as a C function. Upon request from a Unix process, the RTK microkernel server copies the code that will become an RTK task, from user space to microkernel space (using existing IBM Microkernel system calls) and communicates a number of parameters to the RTK via shared memory. The parameters include the logical starting address of the task, the period of the task, and the cost in CPU time required by the task during one period. These parameters are

used by the RTK's scheduler to determine when to execute the task. The RTK also constructs a TSS for this task by setting the instruction pointer field of the TSS to the address passed in from the RTK microkernel server and a set of initial values common to all RTK tasks. The task is invoked in the next real-time minor cycle and once during every one of the task's periods thereafter.

5. Theoretical Underpinnings

In order for our technology to be useful, programmers must be able to determine the conditions under which their real-time tasks will be feasible. For the particular design of a CPU executive that allows the real-time kernel to execute periodically, we develop necessary and sufficient conditions for executing a set of periodic tasks in real-time inside a real-time kernel.

5.1. Related Work

Our starting point for this analysis is the hybrid static/dynamic priority scheduling model presented in [22] and solved exactly in [23]. The model considered in these works partitions all computations performed in the system into those scheduled according to a static priority assignment and those scheduled according to a dynamic priority assignment. In essence, the system executes in two modes: whenever work with a static priority assignment arrives in the system it is scheduled for execution by a static priority scheduler, whenever there is no static priority work remaining, work with dynamically assigned priority is scheduled for execution by a dynamic priority scheduler. In this model static priority work takes precedence over dynamic priority work. For example, in [23] a real-time system consisted of a set of interrupt handlers that executed in response to periodic interrupts, and a set of periodic application tasks. Interrupt handlers were assigned a static priority equal to the interarrival time of their corresponding interrupt (*e.g.*, a rate-monotonic priority assignment) and application tasks used their current deadline as their execution priority. Interrupt handlers always had priority over application tasks.

The results of the analysis of hybrid static/dynamic priority scheduling models from [23] can be applied here by, for example, modeling the system as a set of real-time periodic tasks and a single static priority periodic task with execution cost mc_{nrt} and period MC that "executes" the non-real-time processing workload. However, since in this vision of the system there is only one "static priority task," our system is a special case of the more general model studied in [23]. In [23] the feasibility conditions were expressed in terms of a recurrence relation. For the special case considered here, we can actually generate a more appealing closed form solution.

5.2. Formal Model and Analysis

We consider time to be a sequence of (discrete) clock ticks. Ticks are indexed by the natural numbers and the interval between successive clock ticks is referred to as a time unit. We further consider a dichotomy of time units: *real-time units* and *non-real-time units*. A major cycle is a sequence of $MC = mc_{nrt} + mc_{rt}$ time units; mc_{nrt} contiguous non-real-time time units and mc_{rt} contiguous real-time time units. Time is organized as an endless sequence of major cycles. Real-time tasks execute only during real-time time units and non-real-time tasks execute only during non-real-time time units.

For simplicity, we consider a real-time task to be a periodic task [22]. Specifically, task T is a pair (c, p) where c is the maximum amount of processor time required to execute task T to completion on a dedicated uniprocessor, and p is the interval between successive invocations of T. That is, a task initially invoked at some time t with successive invocations occurring every p time units thereafter. The i^{th} invocation of T occurs at time $t + (i-1)p$ and must complete execution no later than the $deadline$ of $t + ip$. This will require that c units of processor time be allocated to the execution of T in the (closed) interval $[t + (i-1)p, t + ip]$. If this does not occur then task T is said to have missed a deadline at time $t + ip$. We assume task invocations occur at clock ticks and that parameters c and p are expressed as integer multiples of time units.

We define a task set τ as a set of n tasks, $(c_1, p_1) \ldots (c_n, p_n)$. With respect to a task set, the primary measure of interest is $feasibility$. A task set is $feasible$ if it is possible to schedule the tasks such that each invocation of each task completes execution at or before its deadline.

Lemma 1: For all l, $l \geq 0$,

$$\left\lfloor \frac{l}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, l \bmod MC - mc_{nrt}) \qquad (1)$$

is the greatest lower bound on the number of real-time processor units in the interval $[t, t+l]$.

Proof: For all t, $t \geq 0$, the interval $[t, t+l]$ contains at least $\lfloor l/MC \rfloor mc_{rt}$ real-time processor units and at least $\lfloor l/MC \rfloor mc_{nrt}$ non-real-time processor units. Of the remaining

$$l - \left(\left\lfloor \frac{l}{MC} \right\rfloor mc_{rt} + \left\lfloor \frac{l}{MC} \right\rfloor mc_{nrt} \right) = l - \left\lfloor \frac{l}{MC} \right\rfloor MC = l \bmod MC$$

processor units in the interval, at most mc_{nrt} of these can be non-real-time units. Thus, if $mc_{nrt} > l \bmod MC$, then

at least $l \bmod MC - mc_{nrt}$ additional processor units must be real-time units. Therefore the greatest lower bound on the number of real-time processor units in the interval $[t, t+l]$ is

$$\left\lfloor \frac{l}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, l \bmod MC - mc_{nrt}). \qquad \square$$

Theorem 2: A set of periodic tasks $\tau = \{(c_1, p_1), (c_2, p_2), ..., (c_n, p_n)\}$ can be executed on a CPU multiplexer with parameters (mc_{nrt}, mc_{rt}) if and only if for all $L \geq 0$

$$\left\lfloor \frac{L}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, L \bmod MC - mc_{nrt}) \geq \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i \quad (2)$$

where $MC = mc_{nrt} + mc_{rt}$.

Proof: ($\Rightarrow$) A set of tasks can be scheduled only if for all $l \geq 0$, the amount of processor time available to real-time tasks in the interval $[0, l]$, is at least as big as the work requested by invocations of tasks with deadlines in $[0, l]$.

In $[0, l]$, each real-time task will require $\lfloor l/p_i \rfloor c_i$ units of processor time to ensure no invocation of the task misses a deadline in the interval $[0, l]$. Thus the work requested by jobs of all tasks in $[0, l]$ is $\sum_{i=1}^{n} \lfloor l/p_i \rfloor c_i$.

By Lemma 1, the amount of processor time available to real-time tasks in the interval $[0, l]$ is at least (1). Thus, since (1) is a greatest lower bound, τ can be scheduled only if

$$\left\lfloor \frac{l}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, l \bmod MC - mc_{nrt}) \geq \sum_{i=1}^{n} \left\lfloor \frac{l}{p_i} \right\rfloor c_i$$

Note that no assumptions are made (or needed) about whether or not time 0 corresponds to the start of a major cycle.

($\Leftarrow$) To show the sufficiency of (2) we show that a if task system τ satisfies (2) for all L, $L > 0$, then a deadline driven scheduler will succeed in scheduling τ. This is shown by contradiction.

Assume for all L, $L > 0$, τ satisfies (2) but yet a real-time task in τ misses a deadline when scheduled according to a deadline driven algorithm. Let t_d be the earliest time at which a deadline is missed and let t be the greater of:

- the end of the last contiguous sequence of real-time processor units occurring prior to t_d in which no real-time task executed (or 0 if all processor units up to time t_d have been consumed), or,
- the latest time prior to t_d at which an invocation of a real-time task with deadline after time t_d executes (or 0 if such an invocation does not execute prior to t_d).

By choice of t, no invocation of a real-time task with deadline after t_d executes in the interval $[t, t_d]$. If deadline driven scheduling is performed then the processor demand in the interval $[t, t_d]$, is $\sum_{i=1}^{n} \lfloor (t_d - t)/p_i \rfloor c_i$. Moreover, at least

$$\left\lfloor \frac{t_d - t}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, (t_d - t) \bmod MC - mc_{nrt})$$

real-time processor units are available for real-time tasks in $[t, t_d]$. Since a deadline is missed at time t_d it follows that

$$\sum_{i=1}^{n} \left\lfloor \frac{t_d - t}{p_i} \right\rfloor c_i > \left\lfloor \frac{t_d - t}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, (t_d - t) \bmod MC - mc_{nrt}).$$

However this contradicts our assumption that τ satisfies (2) for all L. Hence if τ satisfies (2) then a deadline driven scheduler will succeed in scheduling τ. It follows that satisfying (2) for all L, $L > 0$, is a sufficient condition for feasibility. $\square$

The proof of Theorem 2 also establishes the optimality of the deadline driven scheduling algorithm for scheduling real-time task sets on a co-resident real-time kernel as the condition that is necessary for feasibility is sufficient for ensuring the correctness of the EDF algorithm.

Note also that as was the case with the task model in [23], feasibility here is *not* a function of processor utilization. In particular it is not the case that the relation

$$\sum_{i=1}^{n} \frac{c_i}{p_i} \leq \frac{mc_{rt}}{MC} = 1 - \frac{mc_{nrt}}{MC}$$

is a sufficient condition for feasibility (consider the case of an infeasible single task $(c, p) = (3, 12)$, $c/p = .25$, when $mc_{rt} = 2$, $mc_{nrt} = 5$, $MC = mc_{rt} + mc_{nrt} = 7$, $mc_{rt}/MC = .286$). In general it is possible to construct both *feasible* task sets with processor utilization equal to mc_{rt}/MC and *infeasible* task sets with processor utilization arbitrarily close to 0. An interesting special case in which feasibility is a function of processor utilization occurs when for all tasks, each c_i is a multiple of mc_{rt} and each p_i is a multiple of MC [24].

5.3. Complexity of Deciding Feasibility

In general, deciding if a task set satisfies (2) requires exponential time as (2) must be evaluated for all L up to, for example, the least common multiple of the periods of

the tasks. However, if $\sum_{i=1}^{n} c_i/p_i < mc_{rt}/MC$, (*i.e.*, the real-time tasks do not use all the processor time available to them) then we can apply a result (Theorem 3.6) from [23] which states that a set of tasks will be feasible if and only if (2) holds only for values of L that are multiples of one of the p_i up to the value

$$B = \sum_{i=1}^{n} \frac{c_i}{p_i} \Big/ 1 - \frac{mc_{rt}}{MC} \quad .$$

In this case feasibility can be decided in time $O(n^2 + P)$, where P is the period of the smallest task [23].

6. Discussion

The utility of our approach to integrating real-time and non-real-time technology depends largely on our implicit premise that executives can be constructed with modest effort for all shared devices and that GPOSs and RTKs (if not developed from scratch) can be modified to accommodate these executives. Our work to date is an existence proof that for certain devices and operating systems the premise is true. Here we comment on two of the fundamental requirements for our approach to be successful.

The first requirement is a source of accurate timing interrupts. On the PC motherboard there are two potential sources and most PC operating systems including the IBM Microkernel, Mach, Windows, and OS/2 use either one or the other [15, 16, 18, 21]. Thus, at least for the PC, there is likely to always be a good source of timer interrupts. Should a source of interrupts not be available then one can always add a separate timing board. This is, in fact, standard practice for configuring machines to run the Real-Time Mach operating system on a PC [26]. Thus it is reasonable to assume all systems will have a reliable source of timer interrupts.

The second is the requirement that inter-operating critical sections can be discovered and locked without access to source code. We are confident that this will be the case for peripheral devices as device driver interfaces are typically well documented. Thus, in the worst case one can simply develop a new device driver for devices that are to be used by the RTK. While writing device drivers is not an enjoyable task, they typically must be written from scratch for most devices that are to be attached to a real-time system. Thus in the worst case our framework does not incur more work than is typically required of others.

The ability to deal with critical sections affecting the CPU will largely be a function of the design and structure of the GPOS. The most difficult critical section for the *i*486 architecture involves the TSS. However, since operating systems such as the IBM Microkernel, OS/2, and Windows do not use the hardware task switch feature, much of the complexity of managing the TSS is mitigated. Only additional study will determine if this is serendipity or not.

7. Summary and Conclusions

Distributed multimedia applications are typical of new and emerging applications that require real-time communication and computation services to realize their full potential. Commercial desktop operating systems, while providing limited or no real-time support, remain the primary platform of choice for these applications. In this work we have investigated a method for allowing real-time kernels to co-exist with general purpose operating systems on a single computer. This provides a means of presenting users and applications with the best of both worlds at a modest cost.

Our basic approach is to develop executives for each shared device that allocate the consumable resources of the device to the co-resident operating systems, and partition the serially reusable resources between the two systems. We have demonstrated the feasibility of our approach by constructing a set of executives to allow the IBM Microkernel and OSF/1 to co-exist with a simple real-time kernel we developed. These executives should also work with the Mach kernel.

The initial prototypes of the executives impose an overhead of 12% on the general purpose and real-time operating systems. More importantly, the real-time kernel is as predictable as a kernel executing on a bare machine. In addition, for the resource allocation model used by the CPU executive we have developed necessary and sufficient conditions for determining when a set of periodic tasks will be feasible when executed on a co-resident real-time kernel that only receives a fraction of the overall processing time.

Many open questions remain. Our present system has three main parameters whose absolute and relative settings are not well understood. The parameters are: the duration of a time slot — the lowest level unit of CPU allocation — the duration of a real-time minor cycle and the duration of a non-real-time minor cycle. Studies to understand the performance implications of parameter choices and the trade-off between choices are on-going. We are also applying our executives to other operating systems, most notably OS/2 and Windows.

8. References

[1] *Processor Capacity Reserves: Operating System Support for Multimedia Applications*, Mercer, C.W., Savage, S., Tokuda, H., IEEE Intl. Conf. on Multimedia Computing and Systems, Boston, MA, May 1994, pp. 90-99.

[2] *Adaptive Real-Time Resource Management Supporting Modular Composition of Digital Multimedia Services*, M.B. Jones, in Network and Operating System Support for Digital Audio and Video, Proc., Fourth Intl. Workshop, Lancaster, UK, November 1993, D. Shepherd, *et al.* (Eds.). Lecture Notes in Comp. Sci., Vol. 846, pp. 21-28, Springer-Verlag, Heidelberg, 1994.

[3] *Dynamic QOS Control Based on Real-Time Threads*, H. Tokuda, T. Kitayama, in Network and Operating System Support for Digital Audio and Video, Proc., Fourth Intl. Workshop, Lancaster, UK, November 1993, D. Shepherd, *et al.* (Eds.). Lecture Notes in Comp. Sci., Vol. 846, pp. 124-137, Springer-Verlag, Heidelberg, 1994.

[4] *Workstation Support for Time-Critical Applications*, J.G. Hanko, E.M. Kuerner, J.D. Northcutt, in Network and Operating System Support for Digital Audio and Video, Proc., Second Intl. Workshop, Heidelberg, Germany, November 1992, R.G. Herrtwich (Ed.). Lecture Notes in Comp. Sci., Vol. 614, pp. 4-9, Springer-Verlag, Heidelberg, 1992.

[5] *Scheduling and IPC Mechanisms for Continuous Media*, Govindan, R., Anderson, D.P., Proc. ACM Symp. on Operating Systems Principles, ACM Op. Sys. Review, Vol. 25, No. 5, October 1991, pp. 68-80.

[6] *Kernel Support for Live Digital Audio and Video*, K. Jeffay, D.L. Stone, F.D. Smith, *Computer Communications*, Vol. 15, No. 6, (July/August 1992) pp. 388-395.

[7] *Support for Continuous Media in the DASH System*, Anderson, D.P., Tzou, S.-Y., Wahbe, R., Govindan, R., Andrews, M., Proc. Tenth Intl. Conf. on Distributed Computing Systems, Paris, France, May 1990, pp. 54-61.

[8] *The Cyclic Executive Model and Ada*, Baker, T.P., Shaw, A.C., *Real-Time Systems*, Vol. 1, No. 1, (June 1989), pp. 7-26.

[9] *A Scheduling Philosophy for Multiprocessing Systems*, Lampson, B.W., *Comm. of the ACM*, Vol. 11., No. 5, (May 1968), pp. 347-360.

[10] *VM/370 Asymmetric Multiprocessing*, Holley, L.H., Parmelee, R.P., Salisbury, C.A., Saul, D.N., *IBM Systems Journal*, Vol. 18, No. 1, (1979), pp. 47-70.

[11] *SPIN — An Extensible Microkernel for Application-Specific Operating System Services*, Bershad, B.N., Chambers, C., Eggers, S., Maeda, C., McNamee, D., Pardyak, P., Savage, S., Sirer, E.G., Op. Sys. Review, Vol 29, No. 1, (January 1995), pp. 74-77.

[12] *Implementing HeiTS: Architecture and Implementation Strategy of the Heidelberg High-Speed Transport System*, Hehmann, D., Herrtwich, R.G., Schulz, W., Schütt, T., Steinmetz, R., Proc. Second Intl. Workshop on Network and Operating System Support for Digital Audio and Video, Springer-Verlag, LNCS, Vol. 614, 1992.

[13] *Peripheral Components*, Intel Corporation, Mt. Prospect, IL, 1993.

[14] *Motorola Semiconductor Technical Data,* Motorola Microprocessor Data, Motorola Design-NET, 602-244-6591.

[15] *An OS/2 High Resolution Software Timer*, D. Williams, IBM Personal Systems Developer, 1991.

[16] *The Design of OS/2*, H.M. Deitel and M.S. Kogan, Addison-Wesley, Reading, MA, 1991.

[17] *The IBM Personal System/2 Hardware Interface Technical Reference*, IBM, 1st Edition, 1988.

[18] *The IBM Personal System/2 and Personal Computer BIOS Interface Technical Reference*, IBM, 2nd Edition, 1988.

[19] *The IBM Personal System/2 Model 95 XP 486 Technical Reference*, IBM, 4th Edition, 1992.

[20] *Microprocessors: Volume II*, Intel Corporation, Mt. Prospect, IL, 1993.

[21] *Undocumented Windows — A Programmer's Guide to Reserved Microsoft Windows API Functions*, Schulman, A., Maxey, D., Pietrek, M., Addison-Wesley, Reading, MA, 1992.

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

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

[24] *A Slotted Architecture For Real-Time Processing*, G. Bollella, Technical Report, Department of Computer Science, University of North Carolina, July 1992.

[25] *A Flexible Real-Time Scheduling Abstraction: Design and Implementation*, Lo, S.L.A., Hutchinson, N.C., Chanson, S.T., Technical Report, University of British Columbia, 1994.

[26] C. Mercer, personal communication, 1994.