AFRL-IF-RS-TR-2002-260
Final Technical Report
October 2002

RESOURCE-CENTRIC REAL-TIME KERNEL AND MIDDLEWARE SERVICES

Carnegie Mellon University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. F215

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

APPROVED: Benjamin Montgomery

BENJAMIN B. MONTGOMERY, 1Lt., USAF
Project Engineer

FOR THE DIRECTOR: [illegible]

WARREN DEBANY, Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 074-0188*

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** October 2002	**3. REPORT TYPE AND DATES COVERED** Final Jun 97 – Dec 01

4. TITLE AND SUBTITLE RESOURCE-CENTRIC REAL-TIME KERNEL AND MIDDLEWARE SERVICES	**5. FUNDING NUMBERS** C - F30602-97-2-0287 PE - 62301E PR - F215 TA - 01 WU - 00
6. AUTHOR(S) Raj Rajkumar	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Carnegie Mellon University Office of Sponsored Research 5000 Forbes Avenue Pittsburgh Pennsylvania 15213-3890	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 3701 North Fairfax Drive Arlington Virginia 22203-1714 / AFRL/IFGA 525 Brooks Road Rome New York 13441-4505	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER** AFRL-IF-RS-TR-2002-260

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Benjamin B. Montgomery, 1st Lt., USAF/IFGA/(315) 330-4624/ Benjamin.Montgomery@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.	**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(Maximum 200 Words)***

This report provides an in depth look at the problem of OS resource management for real-time and multimedia systems where multiple activities with different timing constraints must be scheduled concurrently. Time on a particular resource is shared among its users and must be globally managed in real-time and multimedia systems. A resource kernel is meant for use in such systems and is defined to be one that provides timely, guaranteed and protected access to system resources. The resource kernel allows applications to specify only their resource demands leaving the kernel to satisfy those demands using hidden resource management schemes. This separation of resource specification from resource management allows OS-subsystem-specific customization by extending, optimizing, or even replacing resource management schemes. As a result, this resource-centric approach can be implemented with any of several different resource management schemes. Since the same application may consume a different amount of time on different platforms, the resource kernel must allow such resource consumption times to be portable across platforms, and to be automatically calibrated. Our resource management scheme is based on resource reservation and satisfies these goals.

14. SUBJECT TERMS Real-Time Operating System, Resource Reservation, Resource Utilization, Resource Kernels, Real-Time Java	**15. NUMBER OF PAGES** 24
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Figures

1. Introduction

In this project, we studied in depth the problem of OS resource management for real-time and multimedia systems where multiple activities with different timing constraints must be scheduled concurrently. Time on a particular resource is shared among its users and must be globally managed in real-time and multimedia systems. A resource kernel is meant for use in such systems and is defined to be one that provides timely, guaranteed and protected access to system resources. The resource kernel allows applications to specify only their resource demands leaving the kernel to satisfy those demands using hidden resource management schemes. This separation of resource specification from resource management allows OS-subsystem-specific customization by extending, optimizing or even replacing resource management schemes. As a result, this resource-centric approach can be implemented with any of several different resource management schemes.

The specific goals of a resource kernel are:

- applications must be able to explicitly state their timeliness requirements
- the kernel must enforce maximum resource usage by applications
- the kernel must support high utilization of system resources; and
- an application must be able to access different system resources simultaneously.

Since the same application consumes a different amount of time on different platforms, the resource kernel must allow such resource consumption times to be portable across platforms, and to be automatically calibrated. Our resource management scheme is based on resource reservation and satisfies these goals. The scheme is not only simple but captures a wide range of solutions developed by the real-time systems community over several years.

2. Motivation

Examples of real-time systems include aircraft fighters such as F-22 and the Joint Strike fighter, beverage bottling plants, autonomous vehicles, live monitoring systems, etc. These systems are typically built using timeline based approaches, production/consumption rates or priority-based schemes, where the resource demands are mapped to specific time slots or priority levels, often in ad hoc fashion.

This mapping of resources to currently available scheduling mechanisms introduces many problems. Assumptions go undocumented, and violations go undetected with the end result that the system can become fragile and fail in unexpected ways. We advocate a resource-centric approach where the scheduling policies are completed subsumed by the kernel, and applications need only specify their resource and timing requirements. The kernel will then make internal scheduling decisions such that these requirements are guaranteed to be satisfied. Various timing constraints also arise in desktop and networked multimedia systems. Multi-party video conferencing, mute but live news windows, recording of live video/audio feeds, playback of local audio/video streams to remote participants etc. can go on concurrently with normal computing activities such as compilation, editing and browsing. A range of implicit timeliness constraints needs to be satisfied in such scenarios. For example, audio has stringent jitter requirements, and video has high bandwidth requirements. Disk accesses for compilation should take lower precedence over disk accesses for recording a live telecast.

Two points argue in favor of resource-centric kernels we call "resource kernels":

1. Firstly, operating system kernels (including micro-kernels) are intended to manage resources such that application programs can assume in practice that system resources are made available to them as they need them. In real-time systems, system resources such as the disk, the network, communication buffers, the protocol stack and most obviously the processor are shared. If one application is using a large portion of the system resources, then it implies that other applications get a less portion of the system resources and consequently can take longer to execute. In other words, their timing behavior is adversely affected. Letting kernels take explicit control over resource usage is therefore a logical thing to do to prevent such unexpected side effects.
2. Secondly, our resource model captures the essential requirements of many resource management policies in a simple, efficient yet general form. The implementation of the model can actually be done using any one of many popular resource management schemes (both classical and recent) without exposing the actual underlying resource management scheme chosen. User-level schemes can be used to dynamically downgrade (upgrade) application quality when new (current) resource demands arrive (leave). This feature of the resource model leads to minimal changes from existing infrastructure while retaining flexibility and offering many benefits.

3. Design Goals of a Resource Kernel

The design goals of a resource kernel can be summarized as follows.

G1. Timeliness of resource usage. An application using the resource kernel must be able to request specific resource demands from the kernel. If granted, the requested amount of resources must be guaranteed to be available in timely fashion to the application. An application with existing resource grants must also be able to dynamically upgrade or downgrade its resource usage (for adaptation and graceful degradation purposes). This implies that the kernel must support an admission control policy for resource demands. Conventional real-time operating systems do not provide any such admission control mechanisms, even though an equivalent feature (without enforcement capabilities) could be built at user level.

G2. Efficient resource utilization. The resource kernel must utilize system resources efficiently. For example, a trivial and unacceptable way to satisfy G1 would be to deny all requests for guaranteed resource access. In other words, if sufficient system resources are available, the kernel must allocate those resources to a requesting application. This goal implies that the admission control policy used by the resource kernel have provably good properties. Such proof may be analytical or empirical but our version of the resource kernel provides analytically proven properties. It must be noted that this goal is subordinated to G1, in that guaranteed resource access is the primary goal, and efforts to improve efficiency and throughput cannot happen at the expense of the guarantees. Traditional real-time operating systems leave the matter completely open to the developers, each of whom must use their own schemes to obtain better utilization for their applications.

G3. Enforcement and protection. The resource kernel must enforce the usage of resources such that abuse of resources (intended or not) by one application does *not* hurt the guaranteed usage of resources granted by the kernel to other applications. Traditional real-time operating systems such as those compliant with the POSIX Real-Time Extensions do *not* satisfy this goal.

G5. Portability and Automation. The absolute resource demands needed for a given amount of work can, unfortunately, vary from platform to platform due to differences in machine speed. For example, a signal-processing algorithm can take 10ms on a 2GHz Pentium but take 20ms on a 1GHz Pentium. Ideally, applications must have the ability to specify their resource demands in a portable way such that the same resource specification can be used on different platforms. In addition, there must exist means for the resource demands of an application to be automatically calibrated.

G6. Upward compatibility with fielded operating systems. A large host of commercial and proprietary real-time operating systems and real-time systems exist. Many of these systems employ a fixed priority scheduling policy to support provide real-time behavior, and the rate-monotonic or deadline-monotonic priority assignment algorithm is frequently used to assign fixed priorities to tasks. Basic priority inheritance is used on synchronization primitives such as mutexes and semaphores to avoid the unbounded priority inversion problem when tasks share logical resources. For example, Solaris, OS/2, Windows, Windows NT, AIX, HP/UX all support the fixed priority scheduling policy. The Java virtual machine specification also does. Priority inheritance on semaphores is supported in all these OSs (except Windows NT). POSIX real-time extensions, Unix-derived real-time operating systems such as QNX and LynxOS, and other proprietary real-time operating systems such as pSOS, VxWorks, VRTX, OS/9000, and iRMX support priority inheritance and fixed-priority scheduling. To be accepted, the resource kernel must be upward compatible with these schemes. The priority inheritance scheme is also used or being considered for use in multimedia-oriented systems.

Goals G1-G4 are integral to resource kernels, while goals G5 and G6 are for practicality and convenience. Goals G1, G2, G5 and G6 can be satisfied by appropriate extensions to traditional real-time operating systems which support fixed priority CPU scheduling. For example, a user-level server can perform admission control using a resource specification model similar to ours, and assign fixed priorities based on the resource parameters used by our model. However, in order to satisfy goals G3 and G4, the internals of these operating systems need to be modified in ways similar to our resource kernel design and implementation.

4. The Resource Reservation Model

The resource kernel gets its name from its resource-centricity and its ability of the kernel to

- apply a uniform resource model for dynamic sharing of different resource types,
- take resource usage specifications from applications,
- guarantee resource allocations at admission time,
- schedule contending activities on a resource based on a well-defined scheme, and
- ensure timeliness by dynamically monitoring and enforcing actual resource usage,

The resource kernel attains these capabilities by reserving resources for applications requesting them, and tracking outstanding reservation allocations. Based on the timeliness requirements of reservations, the resource kernel prioritizes them, and executes a higher priority reservation before a lower priority reservation if both are eligible to execute.

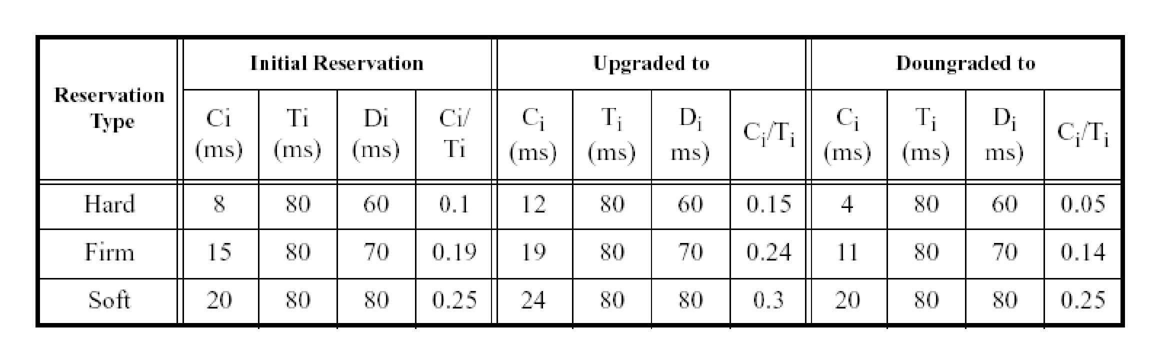
Reservation Type

When a reservation uses up its allocation of C within an interval of T, it is said to be *depleted*. A reservation that is not depleted is said to be an *un-depleted* reservation. At the end of the current interval T, the reservation will obtain a new quota of C and is said to be *replenished*. In our reservation model, the behavior of a reservation between depletion and replenishment can take one of three forms:

1. **Hard reservations**: a hard reservation, on depletion, cannot be scheduled until it is replenished. While appearing constrained and very wasteful, we believe that this type of reservation can act as a powerful building block model for implementing "virtual" resources, automated calibration, etc.
2. **Firm reservations**: a firm reservation, on depletion, can be scheduled for execution only if no other un-depleted reservation or unreserved threads are ready to run.
3. **Soft reservations**: a soft reservation, on depletion, can be scheduled for execution along with other unreserved threads and depleted reservations.

We now evaluate the processor reservation scheme by running different workloads with and without the use of reservations. All our experiments use a PC using a 120MHz Pentium processor with a 256KB cache and 16MB of RAM. We illustrate two basic points in these experiments:

1. the nature of the three types of reservations, and
2. the flexibility to upgrade and downgrade different reservations dynamically.



Reservation Type	Initial Reservation				Upgraded to				Doungraded to			
	Ci (ms)	Ti (ms)	Di (ms)	Ci/ Ti	C_i (ms)	T_i (ms)	D_i ms)	C_i/T_i	C_i (ms)	T_i (ms)	D_i ms)	C_i/T_i
Hard	8	80	60	0.1	12	80	60	0.15	4	80	60	0.05
Firm	15	80	70	0.19	19	80	70	0.24	11	80	70	0.14
Soft	20	80	80	0.25	24	80	80	0.3	20	80	80	0.25

Figure 1. Configuration Parameters for the Resource Kernel experiments illustrated in **Figure 2**.

Three experiments were configured using the parameters given in Figure 1 and the results are illustrated in Figure 2. In these experiments, three threads running *simultaneously* in infinite loops are bound to the three reservations listed in the table above. In the experiments shown on the left, only these three threads are running. In contrast, in the experiment shown on the right, many other unreserved threads in infinite loops are also running in the background and competing for the processor. The behavior of the three types of reservations is illustrated between these two figures.

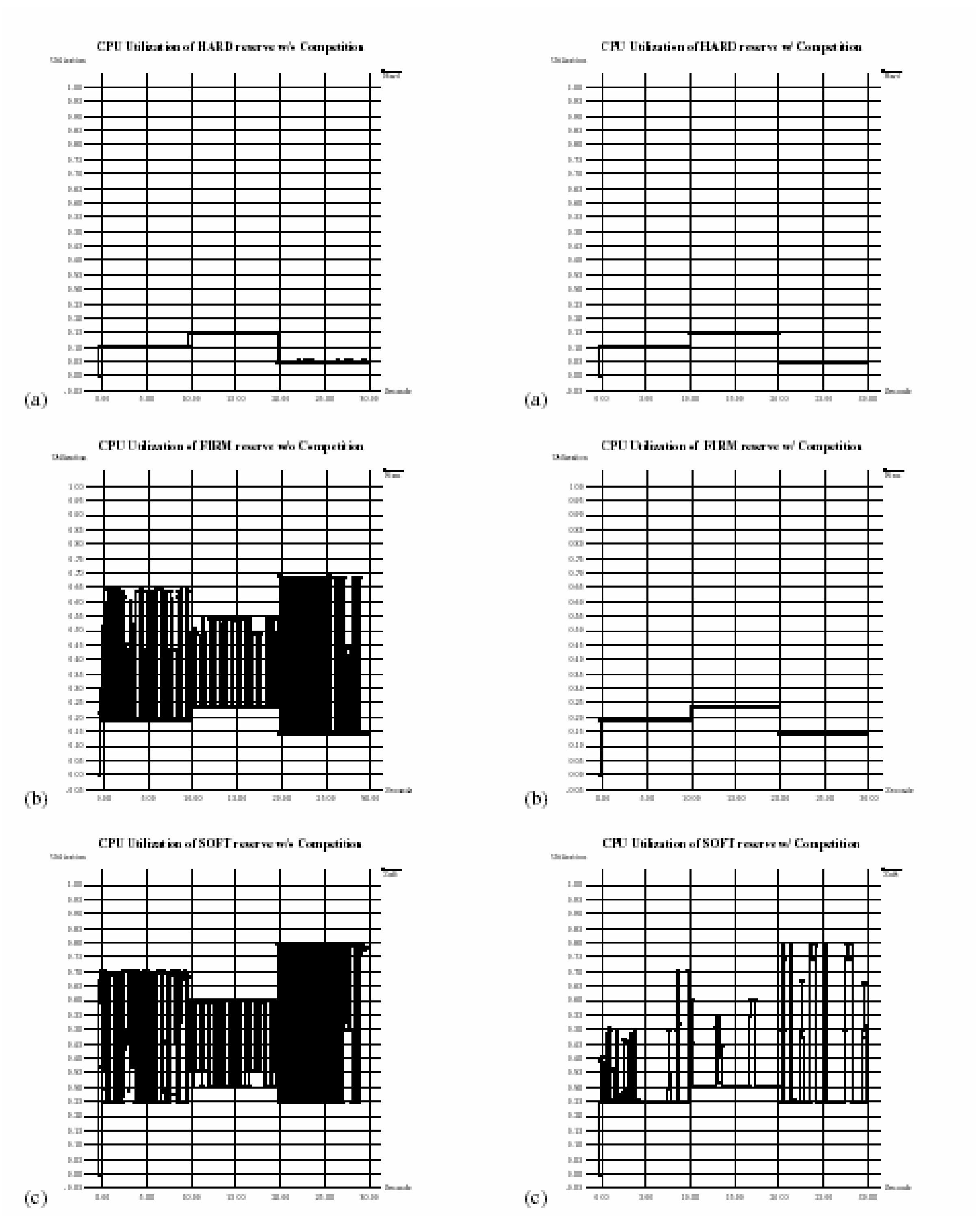


Figure 2. CPU Percentages obtained by infinite-loop tasks with different reservations (that are dynamically resized as per the parameters shown in Figure 1). The left column shows the time obtained by the task when there is no competition from non-real-time (unreserved) tasks, and the right column shows the time obtained by the task when there are other non-real-time

(unreserved) tasks competing for the CPU. (a) Hard reservations are used (b) Firm reservations are used (c) Soft reservations are used.

5. Chocolate: Real-Time Java in Resource Kernels

Chocolate is a real-time Java Virtual Machine that interfaces the real-time Java programming language with the abstractions of a resource kernel. It also supports memory regimes to control allocation time and a protocol to bound priority inversion. This version of Chocolate is implemented on top of NT/RK, an OS environment that includes a "portable resource kernel" within the NT kernel. Our detailed evaluation of Chocolate showed that the overhead introduced by NT/RK is acceptable. A Real-Time Java audio package on Chocolate demonstrated significantly better performance than its non-real-time counterpart. However, our Hartstone benchmark evaluations also showed that our NT/RK implementation does have its drawbacks due to the lack of hard real-time capabilities within Windows NT.

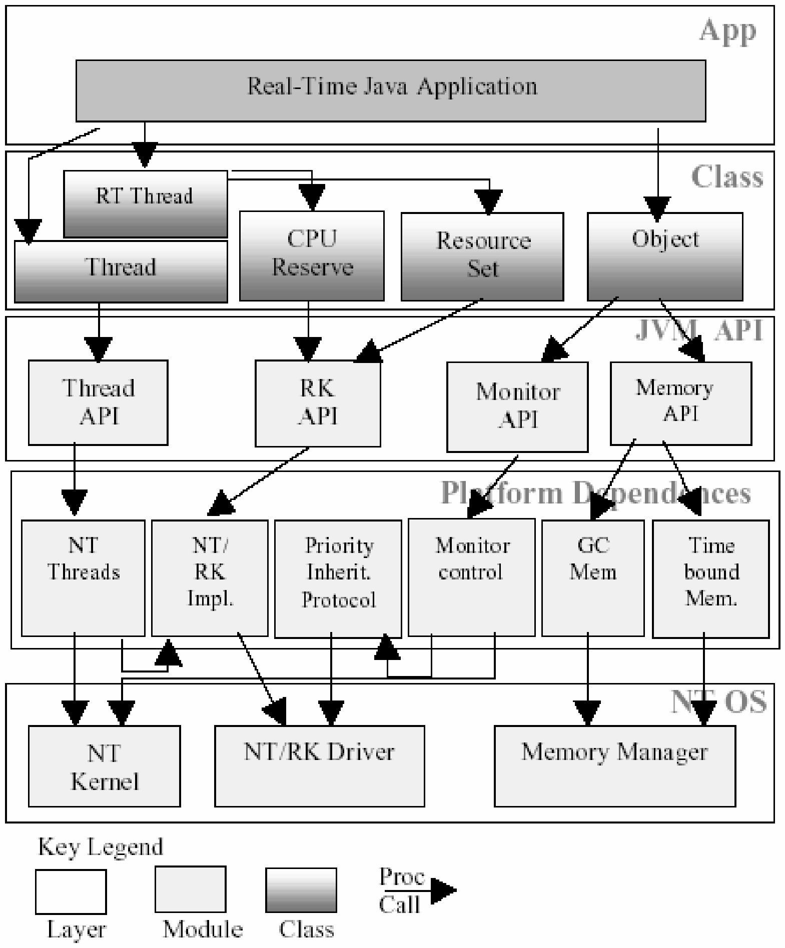


Figure 3. The Architecture of the *Chocolate* Real-Time Java Virtual Machine.

The architecture of Chocolate is shown in Figure 3.

6. Hierarchical Reservation

The temporal isolation properties of resource kernels are appealing enough that many systems desire to recursively apply this reservation model to *each* of their components. This recursive application provides flexible load isolation among applications, users and other high-level resource management entities such as aggregated flows for network bandwidth. The hierarchical reservation study can be applied to hierarchical schedulers that support heterogeneous scheduling algorithms. We propose and analyze a hierarchical reservation model in the context of fixed-priority scheduling, rate-monotonic and deadline-monotonic, as used in systems such as the Resource Kernel. Detailed schedulability analyses under both deferrable-server and sporadic-server replenishment schemes, including exact completion time tests under hierarchical deadline-monotonic schedulers, are presented. We also derive the least upper scheduling bound for hierarchical rate-monotonic schedulers.

We have designed and implemented a hierarchical reservation model as a solution. In our system, any resource management entity, such as a task, an application and a group of users, is able to create a reservation to obtain resource and/or timing guarantees. Resource requests will be granted only if the new request and all current allocations can be scheduled on a timely basis. Each reservation can then recursively create child reservations and become a parent reservation. Different parent reservations can specify different scheduling policies to suit the needs of their respective descendants. For example, one node in the hierarchy may use a deadline-monotonic scheduler, a proportional fair-share scheduler or an earliest deadline first (EDF) scheduler. The resource isolation mechanism will ensure that each child reservation cannot use more resources than its allocation. However, if a child reservation under-uses its resource allocation, those unclaimed resources can be assigned to its siblings. The key challenge of such a system is the capability to grant throughput and latency guarantees to each node in the hierarchy based on its scheduling policy. With that run-time efficiency in mind, we require that admission control for such guarantees be done locally at each level of the hierarchy. An example hierarchy is shown in Figure 4.

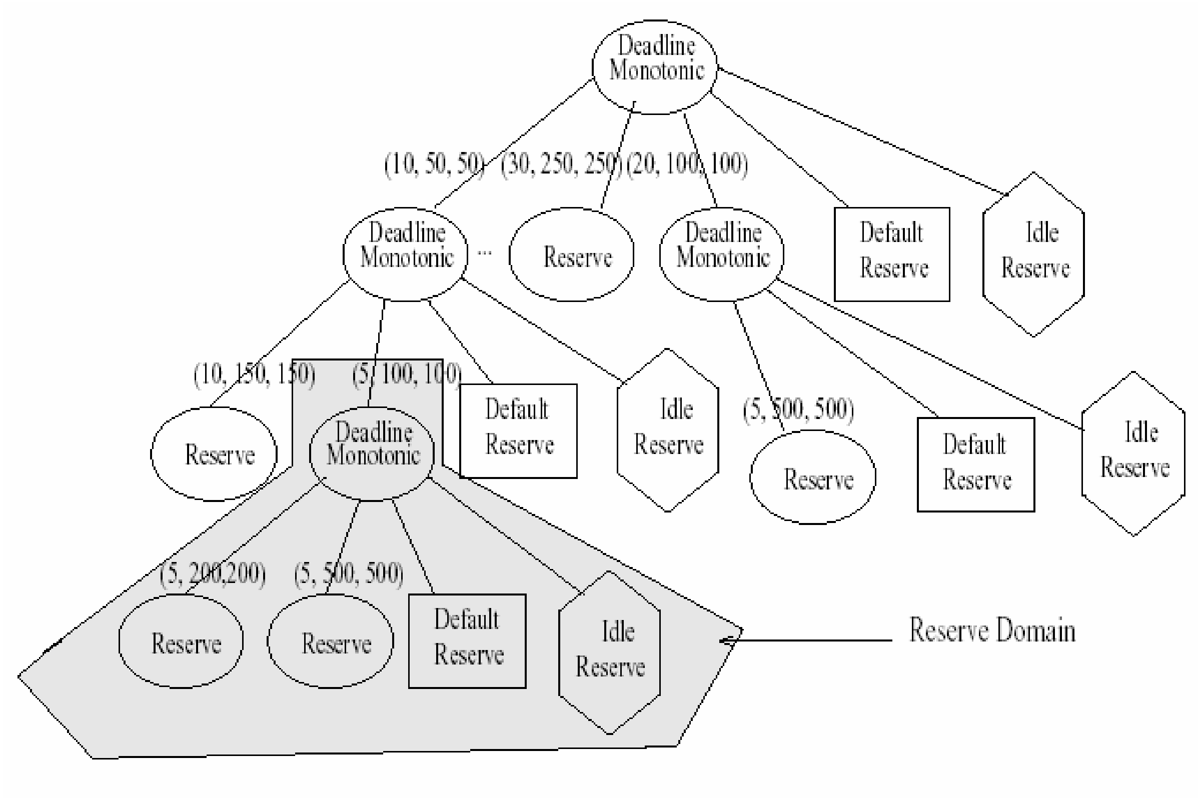


Figure 4. An example set of hierarchical reservations illustrating choice of scheduling policies and hierarchical reservation terminology.

The above hierarchical reservation model provides the following properties:

- *Heterogeneity of Resource Scheduling Policies*: An application is able to choose its own resource scheduling policies that can be real-time, non-real-time, or a combination of both in its own resource partition. In other words, the system avoids scheduling mismatches among schedulers.
- *Hierarchical Enforcement and Protection*: A parent reserve dynamically monitors and enforces actual resource usage of its child reserves so that any timing misbehavior in the lower layer cannot hurt other components.
- *Hierarchical Management of Unused Resources*: If one or more child reserves under-use their reservations, the parent reserve is able to get its entire reserved amount of the resource.
- *Locality of Admission Control*: The parent reserve is responsible for determining the schedulability of its child reservations based on its own reserve specification, its scheduling policy and its children's resource requirements. In other words, the admission control computations does not depend on other reservations outside a given reserve domain.
- *Uniform Reserve Specification*: In order to maintain schedulability analysis locally in each layer of the hierarchy, a uniform reserve specification for both real-time and non-real-time schedulers is used. This abstracts away the details of the scheduling policies.

7. Network Bandwidth Reservation in Resource Kernels

A resource kernel must schedule multiple tasks which have different timing constraints and which access various resources including the CPU, disk and network. These resources, however, are not independent of one another. For example, resources like network bandwidth and disk bandwidth are available on a single node but must be managed by their host OS on the CPU by means of interrupt handlers, device drivers, file-systems and/or protocol services. Hence, in order to obtain guaranteed completion times, an application must therefore obtain both user-mode time on the CPU along with sufficient OS-level time for the network and disk subsystems. In this paper, we investigate the co-scheduling of CPU cycles and network bandwidth. Specifically, we study the problem of obtaining pre-specified network bandwidth received by applications from the external network. Our solution endows the following:

1. Direct control over the flow of network packets into the system based on the requirements of specific applications,
2. Guaranteed and enforced processing time for the received packets,
3. Precise accounting of those processing times, and
4. Elimination of scheduling anomalies.

The Need

The main goal of resource kernels is to provide timely, guaranteed and protected access to system resources. The resources are in general not completely independent of each other. Hence, there are situations where the guaranteed access of one resource disrupts the same of another, giving rise to scheduling anomalies. For example, a multimedia application needs to send real-time data across the network and demands a certain amount of network bandwidth reservation and timely response from the system. A traditional network scheduler mainly involves a packet-queuing discipline that performs quick transmission of packets. But the application itself needs to generate enough data in timely fashion in order to supply an adequate number of packets to the network scheduler. This comes at the expense of a significant number of processor cycles both at the user-space and the system-space. Consequently, this calls for processing guarantees for the application and the packet scheduler at the same time.

It must be noted that the task of receiving data from the network is more processor-centric. The system does not have direct control on the number of packets arriving on its network device. But it can certainly control the processing of these packets in order to provide guaranteed service to various applications with different timing constraints.

Current Practice

If we investigate network protocol processing in the standard Linux operating system (as shown in Figure 5), it does not provide timing guarantees on the processing of inbound packets. This processing is essentially interrupt-driven. The arrival of a packet in the network device generates a hardware interrupt whose job is to capture and store the packets. This in turn generates a software interrupt (Bottom Half) that performs the protocol processing of packets making them ready for the applications. An application, upon being scheduled to run, retrieves packets from the system-space for its own processing. The whole procedure involves considerable amount of system-level processing that is not very much under the control of the application. In other words, the user-level applications will always be preempted by the system-level activities for processing network packets. Moreover, this processing time in the system space is charged to the preempted process, which might not be the process that will eventually receive those packets.

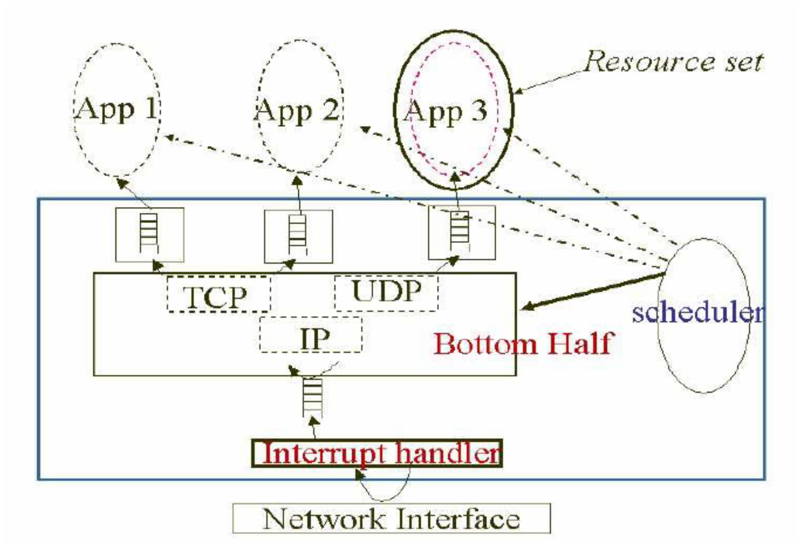


Figure 5. The traditional protocol-processing stack within the Linux operating system.

The persistent handling of incoming network packets at a high priority leads to scheduling anomalies, priority inversion and even overall decreased network throughput. In the extreme case, the packets can arrive at a rate fast enough where the system will spend all of its time on grabbing and storing the packets, and finally dropping them as lost when the queues run out of buffers. Thus, all the inbound data are lost as the application is never scheduled to run, a phenomenon popularly called "receiver-livelock".

The Resource Kernel Network Subsystem

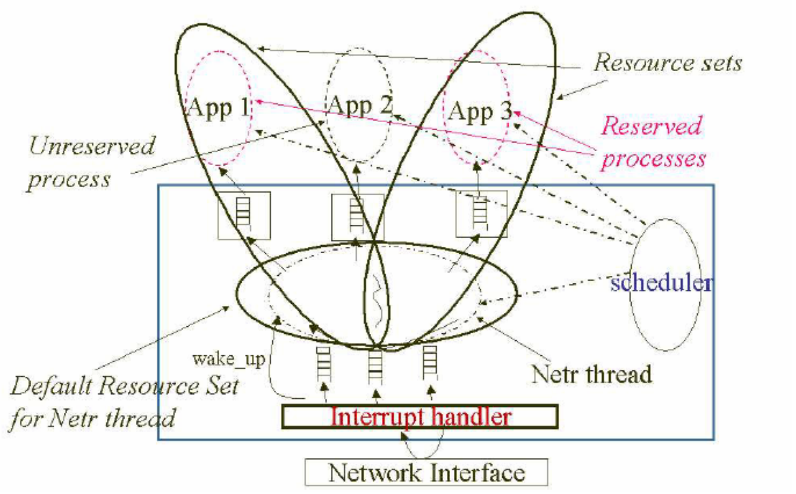


Figure 6. The protocol processing stack inside the Linux/Resource Kernel.

Our real-time network subsystem is called the "NetR" sub-system. Figure 6 shows its architecture in our Linux version of the Resource Kernel called Linux/RK. It comprises mainly of the following components:

netR reserve: Similar to CPU (processor) reserve and network bandwidth reserve, we introduced a new reserve known as "receiving network reserve" or netR reserve. Each netR reserve represents a share of a computing resource, which dictates the rules on how the received packets belonging to a particular reserve should be processed. The main parameters of a netR reserve are:

- *C*: This denotes the volume of data in terms of number of bytes to be received or processed in each period;
- *T*: This is the time duration of the period;
- *D*: This is the deadline in each period within which the processing of the reserved number of bytes should be finished; can be less than or equal to the period T ;
- *Buffer space*: Each netR reserve possesses its own dedicated backlog queue. This parameter represents the maximum capacity of the queue.

A netR reservation requires a prior creation of a CPU reservation under the same resource set. The protocol processing time for network packets gets charged to the CPU reservation. The netR reserve dictates the number of bytes of data to be processed in a given time.

Early de-multiplexing: The single global Linux backlog queue is replaced by one backlog queue for each valid reservation and a default backlog queue for all packets with no reservation. The network interrupt handler is also modified to demultiplex the incoming packets based on their netR reserve, and place them on the appropriate backlog queue.

NetR thread: This is a new kernel thread that replaces the network bottom half and is thereby dedicated to execute protocol processing of arriving network packets. The interrupt handler on receiving a packet wakes up the NetR thread instead of activating the bottom half. The NetR thread handles packets of different reservations using Deadline-Monotonic priorities, which assigns higher priorities to reservations with shorter deadlines (values of D). The unreserved packets at the default backlog queue have the lowest processing priority. This thread is scheduled along with other threads and processes by the CPU scheduler.

While processing packets for a particular netR reserve, this thread dynamically attaches itself to the corresponding CPU reserve of the same resource set. Thus, the NetR thread and the application process may share the CPU reserve of a single resource set and the packets are processed in the context of the appropriate resource set. The NetR thread by itself can be assigned a certain priority or a CPU reserve by default in order to provide desired performance guarantees to the unreserved packets, as in the case of standard Linux. But the user or the system administrator possesses the power to limit the protocol processing time of the unreserved packets so that extra CPU cycles could be utilized in other more "important" or "real-time" tasks in the system. Since the standard Linux kernel is non-preemptive, we made this kernel thread preemptive at packet processing boundaries.

Experimental Evaluation

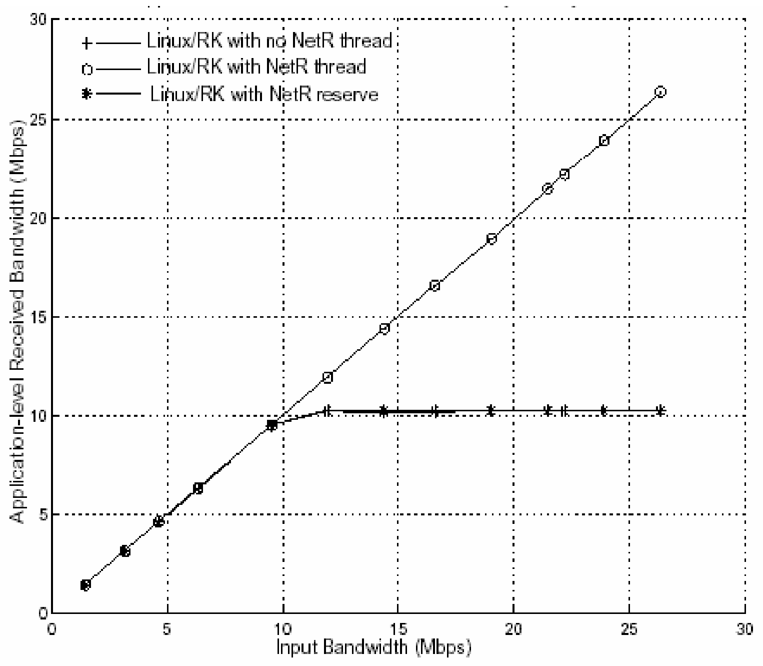


Figure 7. The bandwidth received by an application plotted against the bandwidth sent to the application under different Linux network subsystems.

The results of a sample experiment are shown in Figure 7 to illustrate the benefits of our scheme. A sender process in one machine A sends fixed-sized UDP packets (80 bytes) to a receiver process at another machine B. The sending data rate from A was again varied by means of a network reservation on Linux/RK. The reservation had a period of 4ms. The receiving application received packets using blocking socket I/O and discarded them immediately. It was assigned a Soft CPU reservation that was more than enough to receive and process data at a certain rate. The above figure plots the rate at which the packets were received by the receiver process (in Mbps) against the sending bandwidth of the sender process (in Mbps). According to the figure, the application- level received bandwidth equaled the transmission rate up to a transmission bandwidth of 9 Mbps on standard Linux, beyond which it flattens out. It remained constant up to the measured sending rate of approximately 26 Mbps. This shows that beyond the bandwidth of 10Mbps, buffer-over caused loss of data. Since the buffer-space of the backlog queue is larger than that of the socket queue by default, we conjecture that the over occurred at the socket queue. This happened since the network bottom half, whose job is to take packets out of the backlog queue, process them and put them at the socket queue, ran too often compared to the application process at high input bandwidth. Next, the same experiment was performed with the NetR subsystem. The NetR thread was assigned a priority lower than any CPU-reserved thread but higher than other time- shared threads. With the NetR thread, the application-level throughput remained at par with the sending rate throughout the measurement range.

Using blocking socket I/O, the application blocked on the `recvfrom` system call. And as the protocol processing by the NetR thread took place at a lower priority than the CPU-reserved application, it could only run when the application waited to receive data and hence could not cause the socket queue to over This explains why running protocol processing at a lower priority than the application process in- creases throughput under blocking socket I/O. This is not necessarily true for a few specific applications that perform non-blocking socket I/O. A non-blocking I/O inherently assumes that the I/O process runs at a higher priority and therefore it will preempt the application process.

Next, the receiver created a netR reservation with the fol- lowing specifications: parameters C; T; D were chosen based on the specifications at the sending side network reservation at Abel; the buffer space parameter was assigned to be 40% of the total buffer space reserved for received packets (i.e., the backlog queue in standard Linux) in the system. This reserve was attached to the application process so that the NetR thread could perform protocol processing at the con- text of the CPU reserve of the application. As observed from Figure 7, the received throughput under this condition followed that of the standard Linux subsystem. In this case, when the NetR thread got dynamically attached to the same resource set as that of the application, it not only inherited its reservation but also got ahead of the application thread in the "task-list" queue of the resource set. Therefore, it was scheduled to run before the application causing the same phenomenon of socket-queue over as observed in standard Linux. Suppose the netR reserve had not been attached to the application process directly, but instead only to the sockets involved in the communication so that a separate CPU reserve could be used for the NetR thread. Then, we would have seen similar throughput as in the second case (NetR thread with no netR reserve) if that CPU reserve ran at a lower priority than the CPU reserve of the application process.

The experiment shows that running protocol processing at a higher priority than the user-level process is not always desirable for higher throughput. In other words, the relative priorities of the application process and the protocol processing can be modified in our system depending on the application requirements.

8. Disk Bandwidth Reservation in Resource Kernels

Traditional real-time systems have largely avoided the use of disks due to their relative slow speeds and their unpredictability. However, many real-time applications including multimedia systems, real-time database and C3I applications benefit significantly from the use of disks to store and access real-time data. We have investigated the problem of obtaining guaranteed timely access to files on a disk in a real-time system. Our study focused on several aspects of this problem of providing a real-time filesystem. First, we considered the use of two real-time disk scheduling algorithms: earliest deadline scheduling and just-in-time scheduling, a variation of aperiodic servers for the disk. The latter algorithm is designed to improve disk throughput that can be hurt when a real-time scheduling algorithm such as EDF is applied directly. Admission control policies with practically acceptable properties of performance and usability were provided. Next, we designed and implemented a real-time file-system. This interface provides guaranteed and timely access for multiple concurrent applications requiring disk bandwidth with different timing and volume requirements. Finally, we performed a detailed performance evaluation of the real-time filesystem including its raw performance. We showed the following positive but rather surprising result: our real-time scheduling filesystem not only provides guaranteed and timely access but also does so at relatively high levels of throughput.

The Approach

Many real-time applications like real-time databases and C3I systems can benefit from having access to disks. Desktop multimedia systems also need to read from (or write to) disk storage relatively large volumes of video and audio data. In addition, these streams represent continuous media streams, and must therefore be processed by the disk subsystem in real-time. In other words, it would be very useful in practice if disk bandwidth can also be guaranteed in addition to managing processor cycles. In this section, we present a simplistic disk scheduling algorithm based on earliest deadline scheduling. We then improve the algorithm by exploiting "slack" in the reservations to obtain a hybrid of earliest deadline scheduling and a traditional scan algorithm. Our evaluations of these schemes in Section 4 show that guaranteed disk bandwidth reservation can be obtained at only a small loss of system throughput.

Important Considerations

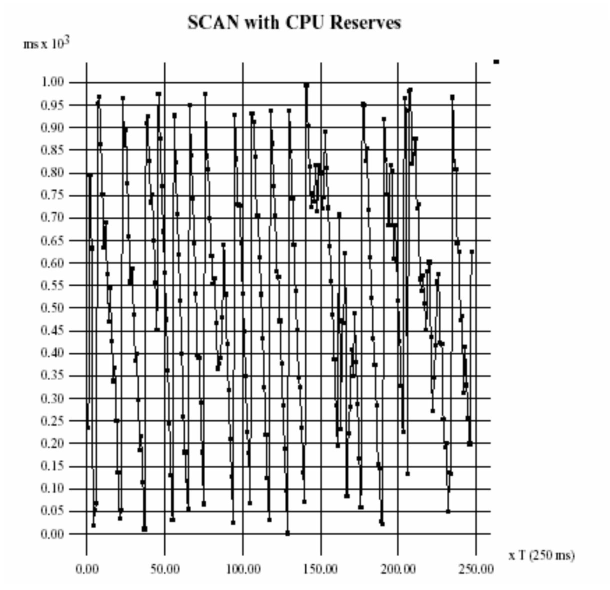
The following important considerations must be taken into account and influence the design of a real-time filesystem:

- **Preemptibility issues**: Once a request is issued to the disk drive, it will not be preempted until it has finished, even if there are higher priority disk requests waiting for service. The time that a higher priority disk request may wait until being serviced is bounded by the longest disk request, which can still be rather long. The duration of the non-preemption window must ideally be small and perhaps even dynamically adjustable depending on the workload.
- **With Preemption**: by implementing fine-grained accesses to the disk, a higher priority disk request can preempt a lower priority disk request midway through the processing of its larger request. Rather than sending the whole disk request in one SCSI command (for example), one can send smaller disk requests successively with several SCSI commands, so that they can be preempted at smaller intervals.
- **Heterogeneity of the workload**: Consider very heterogeneous workloads where there are many small requests with deadlines, but they are prevented from execution due to larger low priority disk requests. Examples of such systems are heterogeneous C3I real-time databases. Consider homogeneous workloads such as multimedia storage servers, where all the requests are periodic ones. SCAN-based schemes are the most effective under these considerations since they avoid expensive disk head movements (seeks).

Filesystem Bandwidth Specification

The resource specification model for disk bandwidth is identical to that of processor reservation in resource kernels. In other words, a disk bandwidth reservation must specify a start time S, a processing time C to be obtained in every interval T before a deadline of D. The processing time C can be specified as # of disk blocks (as a portable specification) or in absolute disk bandwidth time in native-platform specification.

Experimental Evaluation



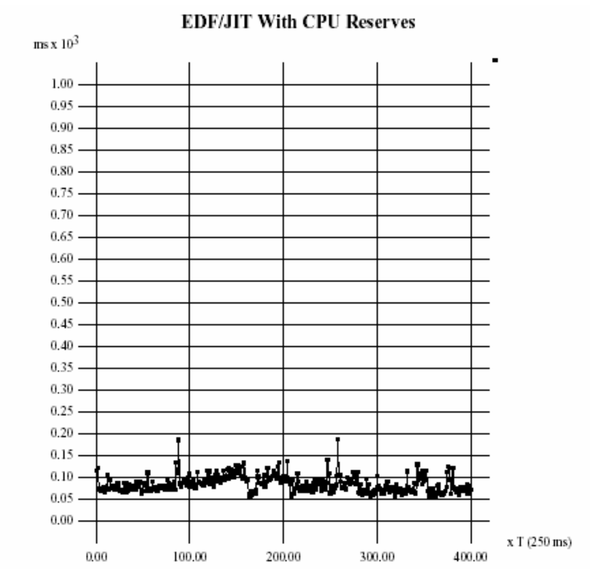
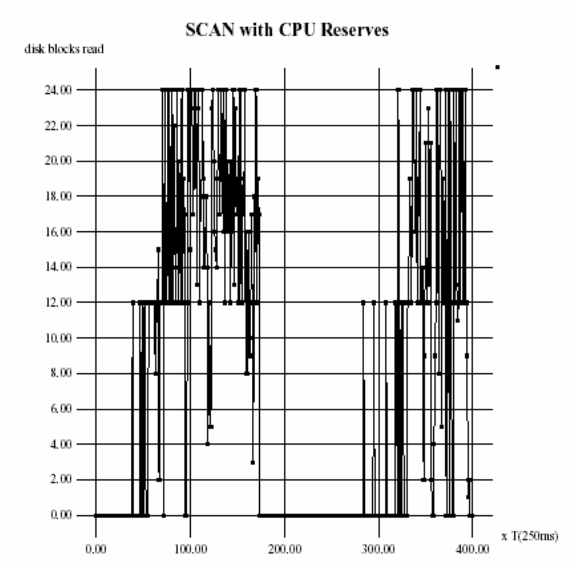


Figure 8. Completion times of disk requests w/o and w/ Disk Bandwidth reserves.



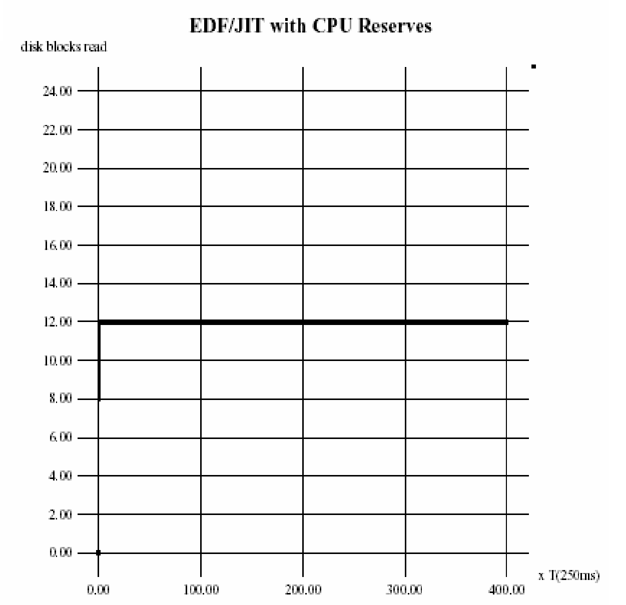


Figure 9. Disk Access per Period w/o and w/ Disk Bandwidth Reserves.

The experimental evaluation of the system is illustrated in the graphs of Figure 8 and Figure 9. The pair of graphs in Figure 8 illustrates how long it takes for a real-time disk access request takes to complete without our scheme and with our scheme respectively. Our scheme uses the Earliest-Deadline-First (EDF) policy with a Just-In-Time slack recovery scheme that allows non-real-time disk requests to get better service. As can be seen, under our scheme, the response times for real-time disk accesses are very small compared to the lack of use of our scheme. The pair of graphs in Figure 9 illustrates how many disk blocks are read by a real-time task over periodic intervals of time (whose value is decided the task's reservation parameters) without our scheme and with our scheme. As can be seen, under our scheme, a constant number of disk blocks are successfully read in each period. This number varies unpredictably and unacceptably without the use of our scheme.

9. QoS-based Resource Allocation

Several applications have the ability to provide better performance and quality of service (QoS) if a larger share of system resources is made available to them. Such examples abound in many domains. Feedback control systems can provide better control at higher rates of sampling and control actuation. Multimedia systems using audio and video streams can provide better audio/video quality at higher resolution and/or very low end-to-end delays. Tracking applications can track objects at higher precision and accuracy if radar tracks are generated and processed at higher frequencies. In many cases, computationally intensive algorithms can provide better results than their less-demanding counterparts. Even interactive systems can provide excellent response times to users if more processing and I/O resources are made available. Conversely, many applications can still prove to be useful and acceptable in practice even though the resources needed for their maximal performance are not available. For instance, a 30 frames/second video rate would be ideal for human viewing, but a smooth 12 fps video rate suffices under many conditions.

We have developed the QoS-based Resource Allocation Model (Q-RAM) to provide a conceptual and analytical framework that addresses the following question: "How does one allocate available resources to multiple concurrent applications?" A unique novelty of Q-RAM is that it allows multiple Quality of Service requirements such as timeliness, cryptography and reliable data delivery to be addressed and traded off against each other. In real-time and multimedia systems, applications may need to have simultaneous access to multiple resources such as processing cycles, memory, network bandwidth and disk bandwidth, in order to satisfy their needs. The solutions that we provide turn out to be very efficient to be used in practice.

Using a video-conferencing application as an example, the following is a sample list of quality dimensions (and their dimensional spaces) that might be associated with any particular application. The list is given to concretely illustrate quality dimensions that might be considered and is not intended to be exhaustive.

- Cryptographic Security (encryption key-length) : 40, 56, 64, 128, 512
- Data Delivery Reliability, which could be
  - maximum packet loss : as a percentage of all packets
  - expected packet loss : as a percentage of all packets
  - packet loss occurrence : as a per packet probability of loss
- Video Related Quality
  - picture format: SQCIF, QCIF, CIF, 4CIF, 16CIF
  - color depth(bits): 1, 3, 8, 16, 24, . . .
  - black/white, grey scale to high color
  - video timeliness | frame rate(fps): 1, 2, . . . , 30
  - low-frame-rate cartoon or animation to high motion picture video

- Audio Related Quality
    - sampling rate(kHz): 8, 16, 24, 44, . . .
    - AM, FM, CD quality to higher fidelity audio
    - sample size (bits): 8, 16, . . .
    - audio timeliness | end-to-end delay(ms): . . . , 100, 75, 50, 25, . . .

The Q-RAM Goals

The goal of Q-RAM is to address two problems:

- Satisfy the simultaneous requirements of multiple applications along multiple QoS dimensions such as timeliness, cryptography, data quality and reliable packet delivery, and
- Allow applications access to multiple resources such as CPU, disk bandwidth, network bandwidth, memory, etc. simultaneously.

Q-RAM uses a dynamic and adaptive application framework where each application requires a certain minimum resource allocation to perform acceptably. An application may also improve its performance with larger resource allocations. This improvement in performance is measured by a *utility function*. Q-RAM considers a system in which multiple applications, each with its own set of requirements along multiple QoS dimensions, are contending for resources. Each application may have a minimum and/or a maximum need along each QoS dimension such as timeliness, security, data quality and dependability. An application may require access to multiple resource types such as CPU, disk bandwidth, network bandwidth and memory. Each resource allocation adds some utility to the application and the system, with utility monotonically increasing with resource allocation. System resources are limited so that the maximal demands of all applications often cannot be satisfied simultaneously. With the Q-RAM specifications, a resource allocation decision will be made for each application such that an overall system-level objective (called *utility*) is maximized.

QoS and Resource Trade-offs

One issue to be dealt with is *QoS Tradeoffs* where a user of an application might want to emphasize certain aspects of quality, but not necessarily others. Users might tolerate different levels of service, or could be satisfied with different quality combination choices, but the available system resources might only be able to accommodate some choices but not others. In situations where a user is able to identify a number of desirable qualities and rate them, the system should be able to reconcile these different demands to maximize the user's preference and to make the most effective use of the system. So it is important for a system to provide a large variety of service qualities and to accommodate specific user quality requirements and delivery as good service as it can from the users' perspective.

An issue related to QoS Tradeoff is *Resource Tradeoffs*. In this case, the tradeoff refers to reconciling or balancing competing resource demands. Resource Tradeoff is often transparent to the user but can be of great help in accommodating user requirements including QoS Tradeoff, especially when the availability of several different resources is not balanced. It arises when an application is able to use an excess of one resource, say CPU power, to lower its demands on another, say network bandwidth, while maintaining the same level of QoS. For example, video conferencing systems often use compression schemes that are effective, but computationally intensive, to trade CPU time for network bandwidth. If the bandwidth is congested on some intermediate links (which is often the case), this benefits the system as a whole. In the case of a mobile client with limited CPU and memory capacity but sufficient link speed with a nearby intermediate powerful server, computationally expensive speech recognition, silence detection and cancellation, and video compression could be carried out on the nearby server. For proxy servers which act as trans-coders/transceivers besides caching data, the proxy servers can distill data for low bandwidth clients (when both server and client have fast CPU, memory and disk bandwidth, but the network link speed in between is limited).

Results

The *general* Q-RAM optimization problem involves multiple resources (MR) and multiple QoS dimensions (MD). The general problem is, therefore, denoted by MRMD. It is useful to identify three special cases of this problem in which either the number of resources is restricted to a single resource (SR) or there is a single QoS dimension (SD) or both. We have found algorithms to solve all the 4 categories of SRSD, SRMD, MRSD and MRMD problems. We illustrate the nature of our results by summarizing how the most complex of these, namely MRMD, is solved. It is to be noted that MRSD and MRMD are NP-hard problems.

We have evaluated and compare three strategies to solve this problem. Two traditional approaches, dynamic programming and mixed integer programming, are used to compute optimal solutions to this problem but we show that their running times are rather high (as might be expected). An adaptation of the mixed-integer programming problem, however, yields near-optimal results with (potentially) significant lower running times. Finally, we present and evaluate an approximation algorithm based on a local search technique that combines multiple resources into a single compound pseudo-resource. This scheme yields a solution quality that is less than 5% away from the optimal solution but is shown to run more than two orders of magnitude faster. In addition, the use of a notion called a *compound resource* allows this technique to be very scalable and robust as the number of resources required by each application increases.

10. Summary

A resource kernel is a resource-centric approach for building real-time kernels that provide timely, guaranteed and enforced access to system resources. A main function of an operating system kernel is to multiplex available system resources across multiple requests from several applications. For example, non-real-time operating systems allocate a time-multiplexed resource to an application based on fairness metrics measured over discrete intervals of time. The key philosophy behind the resource kernel is that precise timing guarantees and temporal protection between applications can be obtained by imposing a well-defined resource usage model on time-multiplexed resources. In other words, an application running on a resource kernel can request the reservation of a certain amount of a resource, and the kernel can guarantee that the requested amount is exclusively available to the application. Since continual monitoring of resource usage is carried out by the resource kernel so as to enforce resource used by any application, such a guarantee of resource allocation gives an application the knowledge of the amount of its currently available resources. A QoS manager or an application itself can then optimize the system behavior by computing the best QoS obtained from the available resources.

The resource kernel concept has been designed in a portable version of a resource kernel and integrated into Linux to create Linux/RK. It allows guaranteed, timely and enforced access by applications to CPU cycles, disk bandwidth and network bandwidth. Support is also available for a Real-Time Java virtual machine on Linux/RK. Linux/RK can be downloaded from the homepage of the Real-Time and Multimedia Systems Laboratory at Carnegie Mellon University (http://www.cs.cmu.edu/~rtml).

List of Publications

1. Saowanee Saewong, Raj Rajkumar, John P. Lehoczky and Mark Klein, "Hierarchical Reservations in Resource Kernels", *Euromicro Conference on Real-Time Systems*, June 2002.
2. Akihiko Miyoshi, Charles Lefurgy, Eric Van hensbergen, Ram Rajamony and Raj Rajkumar, "Critical Power Slope: Understanding the Run-Time Effects of Frequency Scaling", *IEEE Supercomputing Conference*, June 2002.
3. Sourav Ghosh and Raj Rajkumar, "Network Bandwidth Reservation in a Resource Kernel", *IEEE International Symposium on Object-oriented Real-time Distributed Computing*, April 2002.
4. Dionisio de Niz, Luca Abeni., Saowanee Saewong and Raj Rajkumar, "Resource Sharing in Reservation-Based Systems", In *Proceedings of thee IEEE Real-Time Systems Symposium*, December 2001.
5. Dionisio de Niz, Luca Abeni., Saowanee Saewong and Raj Rajkumar, "On Resource Sharing in Reservation-Based Systems", Work In Progress Session, *IEEE Real-time Technologies and Applications Symposium*, June 2001.
6. Miyoshi and R. Rajkumar, "Protecting Resources with Resource Control Lists", *IEEE Real-Time Technology and Applications Symposium*, June 2001.
7. Saowanee Saewong and Raj Rajkumar, "Cooperative Scheduling of Multiple Resources", In the *Proceedings of the IEEE Real Time Systems Symposium*, December 1999, Phoenix, Arizona.
8. Chen Lee, John Lehoczky, Dan Siewiorek, Raj Rajkumar and Jeffery Hansen, "A Scalable Solution to the Multi-Resource QoS Problem", *Proceedings of the IEEE Real-Time Systems Symposium*, December 1999.
9. Kanaka Juvva and Raj Rajkumar, "The Design, Analysis and Implementation of the Real-Time Push-Pull Communications Model", *IEEE Workshop on Real-Time Databases*, 1999.
10. Sourav Ghosh and Raj Rajkumar, "Network Bandwidth Reservation using the Rate-Monotonic model ", *SoftCOM 99*, October 1999.
11. Chen Lee, "On Quality of Service Management", Ph.D. Thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, August 1999. Also available as *Technical Report CMU-CS-99-165*.
12. Chen Lee, John Lehoczky, Raj Rajkumar and Dan Siewiorek, "On Quality of Service Optimization with Discrete QoS Options", In *Proceedings of the IEEE Real-time Technology and Applications Symposium*, June 1999.

13. Shui Oikawa and Raj Rajkumar, “Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior”, In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Vancouver, June 1999.
14. Raj Rajkumar, Chen Lee, John Lehoczky and Dan Siewiorek, "Practical Solutions for QoS-based Resource Allocation Problems", In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1998.
15. Raj Rajkumar, Kanaka Juvva, Anastasio Molano and Shui Oikawa, “Resource Kernels: A Resource-Centric Approach to Real-Time Systems”, *In Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.
16. Anastasio Molano, Kanaka Juvva and Raj Rajkumar, “Real-Time Filesystems: Guaranteeing Timing Constraints for Disk Accesses in RT-Mach”, In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1997.
17. Raj Rajkumar, Chen Lee, John Lehoczky and Dan Siewiorek, "A Resource Allocation Model for QoS Management", In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1997.