Real-Time Support in General Purpose Operating Systems

Kartik Gopalan

Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400

kartik@cs.sunysb.edu

Abstract

The growth in the number of different types of real-time applications, i.e, applications with timing constraints on their execution, has led to much rethinking of the design of timesharing multiuser operating systems. This growth has motivated enormous research efforts in the area of real-time operating system design. The principal themes of these research efforts have been to build effective architectures for real-time operating systems, to explore various real-time scheduling algorithms, and to devise efficient resource allocation mechanisms for various system resources. The goal of this report is to present the fundamental design issues in supporting real-time capabilities in a general purpose operating system. We survey representative works in this field and propose an integrated framework for efficient multi-resource allocation and scheduling.

1 Introduction

In this survey, we present the design issues in providing support for real-time applications in a General Purpose Operating System (GPOS). A GPOS is an OS that is time-sharing, multi-programming, multi-user, and capable of managing multiple resources (such as CPU, disk I/O, network, memory subsystem etc.). The user applications running on GPOS are dynamic in that they have unpredictable starting, blocking and terminating behaviour, and their resource requirements might vary widely with time.

As defined in [28], real-time systems are those in which correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced. Real-time applications can be broadly categorized into *hard* and *soft* real-time applications. *Hard* real-time applications are those that absolutely need to complete execution by the deadline of the computation. They have no value if finishing after their deadlines and in fact the result can sometimes be catastrophic. Examples of such systems include automated industrial assembly lines, space shuttle and flight systems. *Soft* real-time applications, on the other hand, do not have such a strict requirement. They have maximum value if they finish before deadlines but still retain some value if they finish after their deadlines. An example of such an application is a video playback application where the video frames need to be displayed once every 33 ms, or even a text editor where the response to keystrokes needs to be *quick*. However, these tasks retain some of their value even if they miss their deadlines.

In recent years, there has been a proliferation of real-time applications that require guaranteed *Quality of Service* (QoS) from their host operating system. The earliest real-time applications to appear on the scene, such as those in industrial and automation systems, required timely availability of CPU and other I/O resources. Their requirements were satisfied by customized operating systems that dedicated complete computing and I/O power to these applications. Such a solution no longer applies with the arrival of a newer class of real-time applications that need guaranteed resources from a General Purpose Operating System. These applications need to co-exist with other non-real-time applications on the same machine without degrading the latter's performance significantly. One class of such applications are multimedia applications which place heavy demands on CPU, network, disk and memory subsystems of the machine. Another class of such applications are those requiring fast response times such as online trading financial systems. Also, applications run by internet service providers such as in e-commerce to clients through Service Level Agreements (SLA), web hosting and other ASP functions fall into this category since they need to provide guaranteed computing, disk and network bandwidth functions.

To support this new class of real-time applications, a GPOS needs to provide a rich set of new functionalities that support guaranteed resource availability. The traditional role of GPOS has been that of a democratic resource manager, which tries to fairly allocate equal shares of system resources among multiple applications, and achieves good average-case performance on the whole. However, real-time applications need special treatment from the host operating system so that they can meet their performance requirements. In this case, a GPOS needs to suitably reserve the right amount of resources for real-time application while making sure that performance of non-real-time applications does not suffer unduly along the way.

A key requirement from real-time applications is the need for *predictable* behaviour from the operating system. *Predictability* signifies the ability of the operating system to provide bounded *worst-case* delays on execution times. Specifically, the operating system tasks (such as scheduling and memory management) and synchronization primitives (such as semaphores, signals and message passing mechanisms) need to have deterministic execution and blocking times. Frequently, real-time computing is misinterpreted to mean *fast-computing*, whereas in fact what it really means is *predictable* and *timely* computing.

The reason that the operating system kernel is responsible for reserving and managing system resources is because it is in the unique position to handle resource contention among applications and to act as an arbitrator. Specifically, operating system has complete control over CPU scheduling, memory management, and peripheral resource accesses.

This report is organized as follows. In section 2 we first discuss fundamental issues in real-time operating systems design. In section 3 we present the current state-of-art in this field. Given the vast amount of work that has been done in this field, we only highlight representative works and solutions that have proven to have significant influence. In section 4, we present work in the area of integrated resource allocation and scheduling, which includes our work at Stony Brook on Integrated Real-Time Resource Scheduling. In section 5, we classify various real-time systems according to certain basic properties. Finally in section 6, we outline future research trends in real-time systems.

2 Design Issues in Real-Time Operating Systems

In general, a real-time operating system claiming to provide timeliness guarantees to its applications must address the following issues in its architecture. (1) First of all, applications need a mechanism to specify their resource and timing needs to the operating system. (2) Given the application's requirements, the operating sytem needs to have an admission control procedure to accept or reject the application's request. (3) Once admitted, real-time applications are goverened by a scheduling mechanism which multiplexes the system resources among all admitted applications according to their resource requirements. (4) If application threads communicate with each other or access shared system resources, some form of real-time synchronization mechanisms are needed to bound the waiting times. These issues are discussed in this section in further detail.

2.1 Resource Requirements Specification

Resource requirements specification is concerned with the question of how real-time applications express their resource needs to the operating system. For instance, an application could specify the level of QoS it needs in terms of delay bounds, acceptable jitter, acceptable packet loss rate, or even the price it is willing to pay if the system charges the application for the QoS guarantees provided.

Need for *declarative* resource specification interface has been recognized in the literature [32, 36]. *Declarative* specifications mean that applications specify what QoS they require rather than how it is to be provided. As an example, a declarative CPU resource specification can be "execute a particular function with periodicity of P." A non-declarative specification in this case could be stating the exact CPU scheduling policy that can execute this function periodically. A declarative resource specification is obviously more desirable since it hides the implementation details from the application programmer and is supposedly more intuitive.

2.2 Admission Control and Usage Monitoring

Admission control is concerned with the decision procedure that the OS uses to decide whether it can admit a new application with certain level of QoS requirement. This involves mapping the QoS level requested by applications to actual resources needed and checking whether the required amount of resource is indeed available. Admission control is required to predict if applications will meet their timeliness criterion.

Admission control can be *deterministic* or *statistical*. *Deterministic* admission control means the system does not promise more resources to applications than what is available. *Statistical* admission control means, what the system promises to applications is more than what is available, with the assumption that usually not all application need their reserved resources simultaneously. *Deterministic* admission control guarantees resource availability and predictability to applications and is desirable when hard guarantees are required. *Statistical* admission control fares better in terms of efficient utilization of resources and is desirable where probablistic performance guarantees are acceptable.

A closely related issue is *monitoring* the usage of resources by a real-time application that has been admitted to the system. Usage monitoring is necessary to ensure that an admitted application does not consume more than its allocated resource limits. When excessive usage occurs, the operating system uses some feedback mechanism to inform or warn the application about its current resource usage. This mechanism could be something as simple as a signal delivered to the application or something more sophisticated as an upcall from the operating system to some handling routine that the application or middleware registers.

2.3 Resource Scheduling

Resource scheduling is the decision process that the operating system uses to allocate shared resources among competing applications. Each resource in the system needs an associated scheduler that determines which tasks access the resource in what order. For instance, a CPU scheduler multiplexes the CPU resource, a disk scheduler multiplexes the disk bandwidth and a network packet scheduler multiplexes the network link bandwidth. In addition, there there is a need for a global scheduler that takes an overall view of the progress of each application in the system and coordinates the policies of individual schedulers. Since *predictability* is of primary concern in real-time systems, each resource scheduler needs to be predictable, i.e., the time taken to arrive at a scheduling decision must be bounded.

Scheduling algorithms can be classified under two broad categories, namely, *priority* based and *proportional sharing* based.

2.3.1 Priority Based Scheduling

This category of schedulers assign explicit or implicit relative priorities to the applications' tasks and give immediate control of the resource to the task with highest priority. Based on how the priorities are assigned, priority based schemes can be further classified into the following two classes.

- *Static Priority Based Scheduling*: This class of schedulers *statically* assign a priority to all tasks of an application at the time of admitting the task into the system. The criterion of assigning priority can be based on some property of the application, such as periodicity, user previlige, or some pre-determined policy.

  For instance, *Rate Monotonic Scheduling* (RMS) [22] schedules the CPU resource among a set of periodic applications by assigning higher priority to the application with smaller period. Similarly under the *Expedited Forwarding* policy of Diffserv framework [13], network packets are marked as belonging to different classes and static priorities are assigned to each class. Packets belonging to the highest priority class are serviced on the outgoing link before any lower priority ones.

  Although static priority schemes are simple to implement, they do not offer much flexibility in efficient allocation and scheduling of resources. Static Priority based schemes can also potentially suffer from the problem of *starvation* where the lower priority tasks never get control of the resource due to a steady inflow of higher priority tasks. Hence these schemes are usually used in conjunction with some form of rate control mechanisms that limit the rate at which higher priority tasks arrive to some previously agreed upon rate.

- *Dynamic Priority Based Scheduling*: This class of schedulers *dynamically* decide the priority of a task based on factors such as current resource demands and application's requirements. The goal of such schedulers is to have higher flexibility in making resource allocation and scheduling decisions.

  As a simple example, non-real-time systems use *Shortest Job First* scheme in which the shortest task among a set of ready tasks is given the highest priority. Similarly, UNIX and its variants use multi-level feedback queues, where each queue has a priority and tasks of an application can be switch between different priority queues based on whether the application is currently I/O bound or CPU bound. Tasks in the same queue are scheduled by some other scheduling policy such as FIFO or Round Robin. These schemes try to maximize the resource utilization and minimize wait times for applications.

  In the real-time context, a widely used scheme called *Earliest Deadline First* [22] assigns highest priority to the task with earliest deadline among a set of ready tasks. Specification of successive deadlines could be explicit or implicit. Most real-time operating systems support some form of implicit deadline specification for tasks since it is more intuitive to programers and users of real-time applications. For instance, if an application specifies that it needs to execute periodically once every T milliseconds, the scheduler can infer successive deadlines of the tasks of this application as the time at which each successive period completes. EDF is a greedy approach to scheduling and does not perform very well when the resource demand is very high. Hence, like static priority schemes, it needs to be used in conjunction with some form of admission control mechanism in order to be effective during overloads.

2.3.2 Proportional Sharing Based Scheduling

In this class of schedulers each application is assigned a weight proportional to the fraction of the resource reserved for it. A proportional sharing scheduler guarantees that the application will get its reserved share of the resource and provides a delay bound on the time the applications' tasks wait in the queue of the resource. Some examples of proportional share schedulers are as follows :

- *Weighted Round Robin* scheduler maintains a ready queue in which tasks from all applications are queued. The scheduler has a *service cycle* in which it services the tasks in queue giving them their share of the resource. Each application has a share of the service cycle proportional to its reservation. The scheduler traverses the ready queue, scheduling the tasks for an amount of time equal to the corresponding application's share of the service cycle.

- *Hierarchical Proportional Share* schedulers maintain a hierarchy of reservations where each node in the hierarchy represents a reservation domain having a share of the resource. Applications reservations correspond to the leaf nodes in the hierarchy. Each node gets a fraction of the share of its parent node's reservation. Any spare reservation of a node that is not allocated to any of the children is shared by all children of the node. The scheduler services the leaf nodes (applications' tasks) according to their share dictated by the hierarchy.

- *Generalized Processor Sharing* (GPS) [26] and *Weighted Fair Queuing* [10] are packet servicing discplines at the queuing points in the network and also have implications in scheduling of other types resources such as CPU and disk.

  GPS guarantees end-to-end delay bound to a session provided its traffic is leaky-bucket constrained at the source. A GPS server (scheduler) serves N queues for N sessions at a fixed rate r. It is a *work-conserving* discipline i.e., it never stays idle if there are packets to be serviced in any of the queues. A session with packets waiting in its queue to be served is said to be *backlogged*. Let the resource share of a session S_i be ϕ_i and let $W_i(t_1, t_2)$ be the amount of session S_i traffic served in the time interval $[t_1, t_2]$. The following holds for any two sessions S_i and S_j that are *backlogged* throughout the interval $[t_1, t_2]$.

$$\frac{W_i(t_1, t_2)}{W_j(t_1, t_2)} \geq \frac{\phi_i}{\phi_j} \tag{1}$$

  Further, the minimum service rate guaranteed to session S_i under GPS is

$$r_i \geq \frac{\phi_i}{(\sum_{j=1}^{N} \phi_j)} r \tag{2}$$

GPS is an idealized server that does not transmit packets as entities and assumes that it can serve all backlogged sessions simultaneously. The packet-by-packet approximation of GPS is called *Weighted Fair Queueing* (WFQ). In WFQ, when the server is ready to transmit the next packet at time τ, it picks the first packet from the set of currently backlogged sessions, that would complete service in the corresponding GPS system if no additional packets were to arrive after time τ. Let d^p_{GPS} and d^p_{WFQ} be the time at which a packet p gets serviced under GPS and WFQ respectively and L_{max} be the maximum packet length. It can be shown that

$$d^p_{WFQ} - d^p_{GPS} \leq \frac{L_{max}}{r} \tag{3}$$

which means that WFQ cannot fall behind GPS with respect to service given to a session by more than one maximum packet size. Since WFQ services packets by the order of their *virtual finish times*, it can be viewed as a form of EDF scheduling where deadlines of packets are determined by their departure under corresponding GPS system.

It is easy to extend GPS for use in CPU scheduling, since the essential nature of CPU and network link resources is similar. Just as the time to service a packet on a network link is independent of the servicing of the preceding packet, similarly, time to schedule a task on CPU is independent of the preceding task scheduled. Hence WFQ can be applied without major modifications to scheduling of CPU resource. However, we cannot say the same about disk resouce. Service time of a disk request is effected by the seek delay in moving disk head from the position at which preceding request was serviced. Hence, in order to be effective for disk, WFQ needs to take into account seek delay overheads.

In another classification scheme [28], four classes of real-time scheduling algorithms are identified based on (1) whether a real-time system does admission control, (2) when it does admission control, and (3) whether an explicit schedule of execution is generated according to which tasks are dispatched at runtime. These four classes of scheduling algorithms are as follows.

- Static Table Driven approach : This class of schedulers perform admission control for a set of tasks prior to their execution, and compute an execution schedule that is then used at run time. This approach is suitable for scheduling a set of periodic tasks and by construction it is highly predictable but least flexible.

- Static Priority-driven Preemptive approach : This class of schedulers perform admission control prior to execution, but do not generate any explicit schedule. Tasks are assigned priority that reflect their time constraints. At runtime, tasks are executed based on a highest priority first policy.

- Dynamic Planning-based approach : Unlike previous approaches, this class of schedulers can accomodate dynamically arriving tasks. The feasibility of admitting a task is computed at runtime when the task arrives. The task is then given the guarantee that it will meet its time constraints and an execution plan for that task is then constructed. Such schedulers are useful to support strict performance guarantees in a GPOS environment.

- Dynamic Best-effort approaches : This class of schedulers do not perform admission control and provide no guarantees to tasks. However, the system tries its best to meet the time constraints of the tasks. Such scheduling approaches are suitable for soft-real-time applications in a GPOS environment.

2.4 Real-Time Threads and Synchronization

The POSIX threads (PThreads) model is priority driven and preemptive, with signal handling, mutual exclusion, and synchronized waiting. The thread attributes supported in PThreads include scheduling policy, scheduling priority, and stack size. However these attributes are not sufficient to support threads with timing constraints such as deadlines, start time and periodicity. PThreads also provide mutex and condition variables as primary means of synchronization. However, the mutex mechanism does not bound the blocking time. Bounded waiting time on condition variables is possible via `pthread_cond_timedwait()`. In order to do this, the mutex associated with the condition variable needs to be locked by the waiting thread, but the lock operation is an unbounded blocking operation. In summary, synchronization primitives in real-time systems need predictable and bounded blocking time semantics.

Priority Inversion is a problem that occurs in real-time systems using priority-based scheduling. Essentially, unbounded blocking may be caused when a high priority activity cannot preempt a low priority activity. For instance, consider a low-priority thread T_l that enters a critical region after acquiring a mutex lock. While T_l is in the critical region, a high priority thread T_h might attempt to enter the same critical region and block while trying to acquire the mutex lock. Now suppose other threads $T_{m_1}...T_{m_k}$ with higher priority than T_l but lower priority than T_h become active, then T_l can be preempted by these new threads causing T_h to wait for an unbounded duration for lower priority threads to finish.

The common solution to priority inversion is the *Priority Inheritance* mechanism [30]. The basic idea is that a low priority thread inherits the priority of a high priority thread when the latter blocks for access to a resource held by the former. In the above example, T_l will inherit the priority of T_h during the time when it is in critical section. This way T_h's waiting time is bounded by the worst-case execution length of the critical section.

A phenomenon similar to priority inversion, but less harmful in terms of side-effects, also occurs in the context of WFQ scheduling of CPU resource. Again, consider a thread T_1, with a share of resource being scheduled, that enters a critical region after acquiring a mutex lock. While T_1 is in the critical region, a thread T_2, might attempt to enter the same critical region and block while trying to acquire the mutex lock. Since T_2 is blocked, the queue corresponding to its session is not backlogged and hence the WFQ scheduler will skip this queue. On the other hand, if the mutex lock was not held, T_2 would not have blocked for acquiring the lock, its session queue would be backlogged, and scheduler would have given its queue its fair share of the resource. This problem is particularly acute if thread T_1 has a small share of the resource while thread T_2 has a large share. Consequently, thread T_1 makes slow progress in releasing the mutex lock whereas large portions of thread T_2's share go unused. Note that unlike in the case of priority based scheme, there is no starvation of either thread T_1 or thread T_2 since WFQ is inherently fair to all sessions and serves each session's queue that is backlogged.

A possible solution to this problem is for the WFQ scheduler to allocate thread T_2's share of resource to thread T_1 during the time T_1 is in critical section. This is justified since thread T_2 cannot make progress until thread T_1 releases the mutex lock. Hence it only makes sense to give T_2's share of resource to T_1 so that T_1 can make faster progress in releasing the mutex lock.

2.5 Predictable Memory Management

Virtual memory system can be a source of unpredictability in a real-time system. Whenever a page fault occurs, the executing thread is put to sleep for an unbounded period while the missing page is fetched from secondary storage. Most real-time systems get around this problem by pre-allocating and loading all the pages of a real-time application at startup time. Furthermore, either no virtual memory system is supported or if it is, the pages of real-time application are "wired" to the physical pages of memory so that they never get swapped out.

Another source of unpredictability are the TLB misses during execution of a program. Basically, a missed TLB lookup causes the corresponding page table entry to be brought from main memory into TLB, resulting in lost CPU cycles possibly when the application is close to its deadline. One way to get around this problem (as in Spring system described below) is to preload the TLB with the required page table entries at the time of context switch. However, this solution works only if the scheduling scheme is non-preemptive and there are no page faults during the execution of the task.

3 Representative Works

Considering the numerous real-time systems that have been developed over the past two decades, it is difficult to describe each of them in detail. In this section, we choose to examine representative real-time systems that have had significant impact in the research and development of real-time operating systems.

3.1 The Spring System

The Spring Kernel [33] provides real-time support for applications running on multiprocessor and distributed environments. Its *Reflective* architecture considers the current system state in making scheduling decisions, and allows for graceful degradation of application performance. Spring is an elaborate system that aims

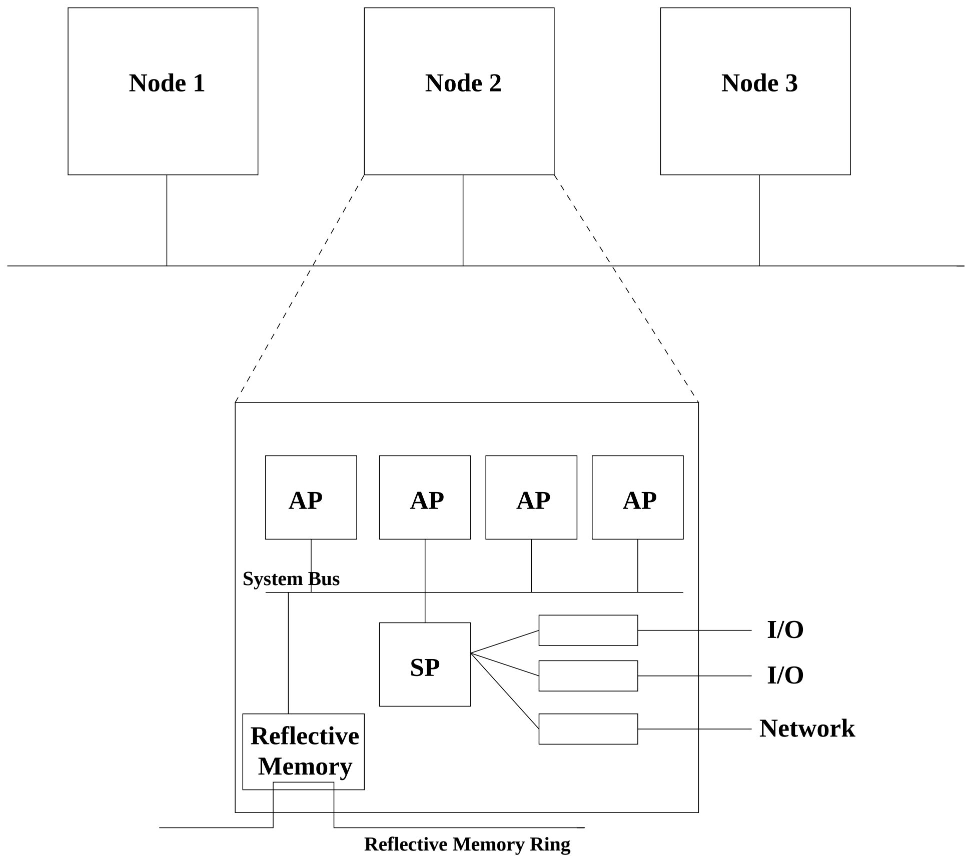

Figure 1: *Hardware Architecture of a Spring Node. A Spring node contains one System Processor (SP) and multiple Application Processors (AP). SP is responsible for scheduling of application tasks on APs and controlling peripheral I/O. Each processor has some assocoated memory shared among processors via the system bus. In addition, each Spring node has a reflective shared memory that is connected across all nodes in a Spring cluster via a fibre-optic ring. Writes to this memory are broadcast to all the nodes in Spring cluster.*

for absolute predictability in dynamic hard real-time environments. It is based on the phiolosophy that to obtain absolute predictability, one must know the *worst case execution time* (WCET) of each task executed in the system (as opposed to just average case execution time). The chief source of runtime predictability in Spring is its dynamic, planning based approach to resource allocation and scheduling which ensures that tasks never block on resources (as would occur in priority based systems).

From the Hardware perspective, as shown in Figure 1, a node running the Spring kernel is a multiprocessor with one System Processor (SP) and multiple Application Processors (AP). SP is devoted to system activities such as scheduling and interacting with peripheral devices. The APs execute application code according to the schedules created by SP. Physical memory resident on each processor board can be referenced directly by any processor in the node.

Two languages (SDL and Spring-C) are used to develop user programs. Spring-C [25] is a modified version of ANSI C meant for use in a real-time environment. Spring-C does not allow programmer to perform operations whose duration cannot be predicted. The Spring-C compiler does compile-time timing analysis and breaks the computation at potential blocking points into a series of precedence-related tasks. Blocking points are points in execution where a process attempts to acquire a resource and has to wait for the resource to be released by another process. Spring completely avoids blocking points by constructing an explicit plan of resource usage at the time of admitting a new process or thread.

The System Description Language (SDL) is used to express timing and resource requirements of an application program. This information is used by the kernel and scheduler to meet the requirements of real-time applications. SDL is generated by hand (or by using a design tool) in certain situations, such as when the user instructs the Spring kernel how much physical memory to allocate at boot-time. Another way SDL is generated is through the invocation of the Spring-C compiler. SDL specification describes the number of tasks, the precedence relationship between tasks, the resource requirements and WCET of each task.

The Spring scheduler is a user-level process running on SP and is responsible for scheduling user applications on the APs. In Spring, a task is an indivisible unit defined by its resource requirements, WCET, and precedence constraints with respect to other tasks. At compile time, a thread is broken down into a set of tasks that are related by precedence constraints. A process consists of a group of threads. When an application program begins execution, a scheduling request, tagged with a deadline, value, and eligibility time, is sent to the scheduler. The Spring scheduler then forms a new task set consisting of all tasks already in the system and those tasks in the new request, and attempts to construct an execution plan for this new task set. If every task in the new task set is schedulable, the new request is accepted for execution. Otherwise, the request is rejected, and the APs continue executing the schedule that was present before the arrival of the request. Thus the Spring scheduler does admission control via a two-part guarantee:

- A new request will only be accepted if every task in the request can be scheduled.
- Once accepted, every task is guaranteed to be executed to meet a hard deadline, irrespective of future scheduling requests at the same or lower levels of importance.

The details of the Spring scheduler algorithms can be found in [16] and [29].

Virtual memory is not supported in the Spring kernel because of its inherent unpredictability. Execution is limited to the processors on which the process was physically loaded. Spring provides predictable memory operations by preallocating physical pages at process creation time and pre-loading the pages into memory. At context switch time, the TLB is explicitly loaded with page table entries needed by the new thread and the scheduler schedules tasks in a non-preemptable fashion. This avoids unpredictability in execution times due to TLB misses. Thus the price paid for better predictability is higher context switch time.

For inter-node communication, Spring uses SCRAMNet, which is a replicated shared memory architecture. Each node has a SCRAMNet board with 2 MB of shared memory connected to a fiber-optic register insertion ring. Writes to this memory are broadcast via the ring to all nodes in a Spring cluster.

3.2 Real-Time Mach

Real-Time Mach [35] is a microkernel based operating system which provides distributed real-time OS mechanisms and services for real-time and multimedia applications. The architecturtre of RT-Mach is shown in Figure 2. The philosophy behind RT-Mach is based on *priority driven preemptive scheduling*. RT-mach adds the notion of *temporal protection* that isolates the timing behaviour of one application from another.

One technique to achieve temporal protection is *lazy evaluation*, which means that the system never does anything until it is absolutely necessary. One example of lazy evaluation is copy-on-write. Here a message is not copied at the time the message is sent, but delayed till the message is actually accessed by the receiver. Similarly, all the network protocol processing is done in a protocol stack library that executes in the context of the process sending or receiving packets[20]. At memory management level, address space is not copied at address space creation time but at the time when it is first accessed by the application. The advantage of lazy evaluation is that it tends to push the execution of most activities to the application context and thus temporally isolates one application from another application. However, lazy evaluation alone cannot predict as when slowdowns will occur. Large amounts of execution time may be required at the wrong time to finally perform the address-space copying or protocol processing, causing the aplication to miss deadlines.

RT-Mach supports a wide range of CPU scheduling policies such as fixed priority scheduling, earliest deadline first policy and round-robin policy, which can be chosen to be active dynamically. It supports the notion of *processor reserve* [19] in which applications request C units of computation every T units of time and get assigned a rate-monotonic priority based on the requested resource need. Multiple threads of control can belong to the same processor reserve. RT-Mach also performs admission control to prevent system overloads and enforces computation time limits. RT-Mach also allows co-existence of real-time and non-real-time threads. A thread is described by a procedure name and a stack descriptor. In addition, a real-time thread has additional *timing attributes* such as whether it has *soft* or *hard* deadlines, whether it is *periodic* or *aperiodic*, and in the former case, its worst-case execution time, period, start-time, etc.

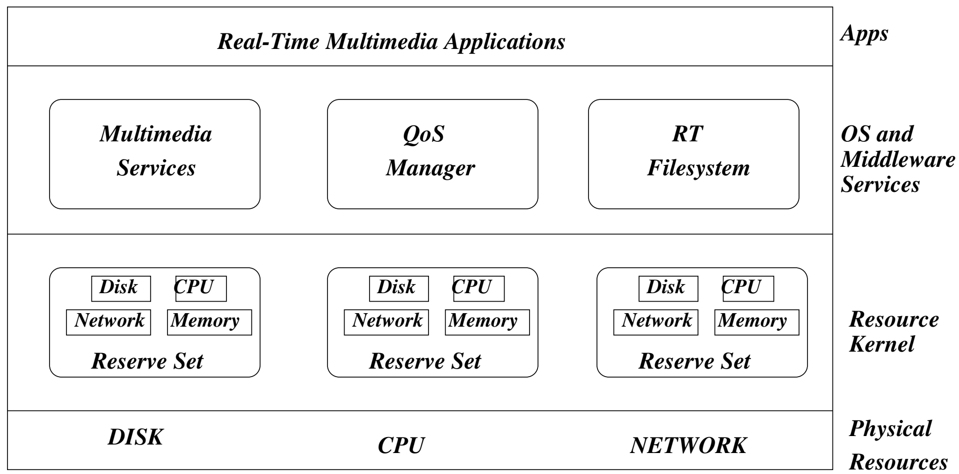

Figure 2: *Architecture of RT-Mach. Multimedia applications use OS and middleware services to specify their QoS requirements. These services in turn reserve resources on behalf of applications in the form of reserve sets. Each reserve set represents a share of the underlying physical resources.*

Synchronization primitives in RT-Mach, such as mutexes and condition variables, support real-time semantics in which earliest deadline first is used as queueing policy for waiting threads. In addition, RT-Mach also addresses the problem of priority inversion commonly encountered in priority-based systems [23] by using the priority inheritance mechanism. Priority of the thread holding a resource is increased to the maximum of the priorities of threads waiting for the resource. This way, the blocking time of a higher priority thread is bounded by the length of the critical section.

3.3 Nemesis

Nemesis [21] is a microkernel based operating system developed at University of Cambridge. It is designed to support time-sensitive applications requiring a consistent Quality of Service (QoS) from all system resources such as CPU, memory, Network, and Disk. In a microkernel environment, an application is typically implemented by a number of processes, most of which are servers performing work on behalf of more than one client. This leads to enormous difficulty in accounting for resource usage. In a traditional kernel-based system, applications spend most of their time in the kernel, leading to similar problems. In contrast to either of the approaches, the guiding principle in the design of Nemesis is to structure the operating system in such a way that the vast majority of functionality comprising an application could execute in a single process, or *domain*. Nemesis has an extremely small lightweight kernel, and performs most operating system functions in shared libraries which execute in the application's domain. This leads to a *vertically* structured operating system.

The Nemesis kernel consists of a scheduler and a small amount of code known as Nemesis Trusted Supervisory Code (NTSC). NTSC is used for inter-domain communication and to interact with the scheduler.

Domain encapsulates the executing state of a Nemesis application. Each domain has an associated *scheduling domain* and *protection domain*. The former determines the CPU share allocation while the latter determines the access rights of the virtual address space.

The NTSC interacts with domains and the kernel scheduler via a per-domain area of shared memory known as Domain Control Block (DCB). Portions of DCB are mapped read-write into the domain's address space while others are mapped read-only to prevent modification of priviliged state (such as scheduling and accounting information, domain's privilige level etc). The DCB contains a vector called *activation handler*. Whenever a domain is given control of the CPU, it is upcalled by Nemesis kernel. The purpose of the upcall is to place information in the activation handler using which QoS-aware applications can assess their progress and can adapt to resource availability. One such information passed during upcall is whether the domain is currently running on guaranteed time of merely being offered some of the spare time in the system. A QoS aware application can take this into account before deciding to adapt to apparent changes in system load.

Nemesis employs a *split-level* scheme for scheduling CPU resource. The scheduling among domains is

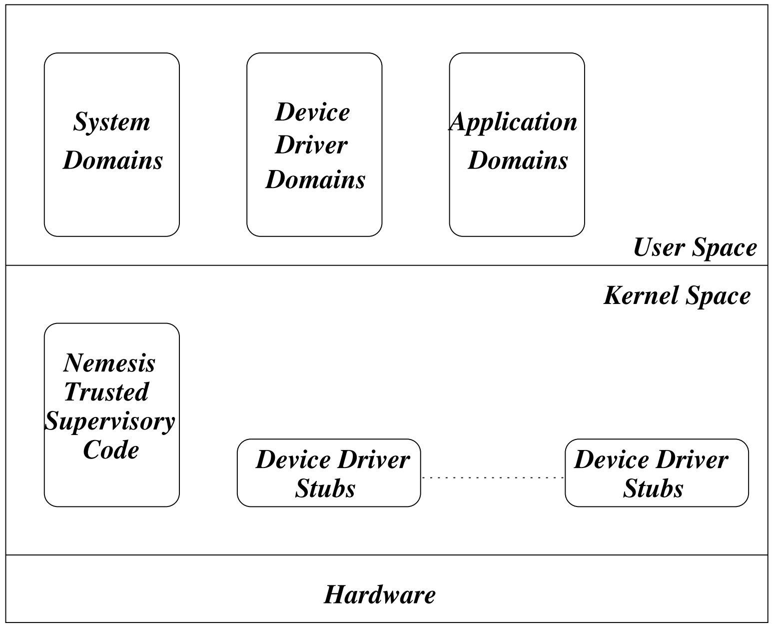

Figure 3: *Architecture of Nemesis. Applications, device drivers and system processes execute as user level domains. Kernel consists of Nemesis Trusted Supervisory Code (NTSC) and small device driver stubs that send event notifications to corresponding device driver domains.*

performed at a low-level by the kernel and scheduling among threads of a domain is performed at user level by the domain itself. Domains can specify their QoS parameters in the DCB in terms of their periodicity, slice of CPU time every period, willingness to accept spare CPU time, and a latency hint indicating how soon a domain should be rescheduled after unblocking. The inter-domain scheduler in kernel is a variant of Earliest Deadline First (EDF) algorithm where the deadlines are derived from the QoS parameters of the domain and are purely internal to the scheduler.

Nemesis does not have a concept of kernel-threads. Instead, it provides user-level threads packages as shared library code with lightweight synchronization primitives such as mutexes and condition variables. Domains may use these user-level schedulers to internally distribute the CPU time provided by kernel scheduler.

Nemesis is a single address space operating system i.e., any accessible region of physical memory appears at the same virtual address in each protection domain. Use of single address space has two principal advantages. (1) It removes the need to flush a virtually addressed cache and the TLB on context switches between domains helping reduce the cache related penalties of context switches. (2) It allows an inter domain communication mechanism in which the kernel is not involved in copying the bulk data transferred between domains but only in the transfer of small meta-data describing the bulk data to be transferred. The actual transfer of bulk data is achieved by mapping of shared memory regions among communicating domains.

Nemesis device drivers, such as for network and disk I/O are implemented as priviliged domains which are scheduled exactly the same way as other domains, but with permission to switch between kernel and user modes in order to perform priviliged I/O operations on behalf of other applications. The kernel demultiplexes hardware interrupts and invokes small device specific stubs that are registered by device driver. These stubs usually do little more than send notification to the appropriate device driver. Hence the amount of code that executes in the kernel on behalf of the applications is kept to a minimum, reducing the QoS-crosstalk problem between applications. The client applications send a stream of I/O requests to the driver on pre-established connections. The requests are serviced by the corresponding resource scheduler in the driver according to QoS parameters associated with the connection.

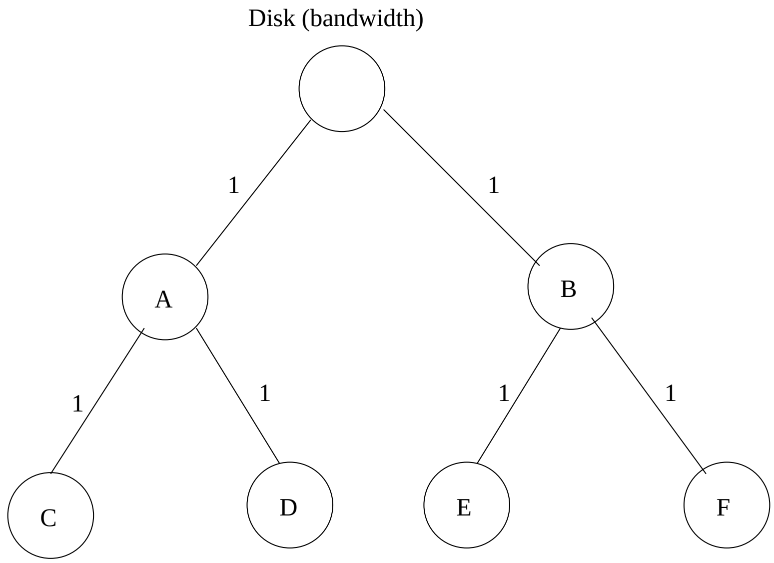

Figure 4: *Scheduler Hierarchies in Eclipse/BSD. Each node in the hierarchy is assigned a share of its parent's reservation of the resource. Internal nodes of the tree, i.e.* scheduler nodes, *implement a scheduling algorithm for descendant nodes. Actual resource requests are queued at leaf nodes, i.e.* queue nodes.

3.4 Eclipse/BSD

Eclipse/BSD [6, 4] is an operating system derived from FreeBSD that provides flexible and fine-grained Quality of Service support for applications. One of the design goals of Eclipse/BSD is to provide QoS support for a large set of server applications without imposing significant changes to their design.

Eclipse/BSD supports the notion of a hierarchy of reservations. The hierarchy consists of two-kinds of nodes: *schedulers* and *queues*. The scheduler nodes implement a scheduling algorithm for ordering resource requests from immediate descendant nodes. Scheduler nodes can have queues and scheduler nodes as immediate descendants. Queue nodes are leaves of the hierarchy where actual resource requests are queued. This framework of scheduler and queue nodes supports a variety resource schedulers. For example, a scheduler node with a single descendant node represents a FIFO scheduler. A scheduler node, whose immediate descendants are all queues could implement (1) priority scheduling if the queues are assigned priorities, or (2) proportional-share scheduler based on weighted fair queuing if the queues are assigned weights.

Eclipse/BSD implements hierarchical, proportional-share CPU, disk and link schedulers [26, 3] at the device driver level. Each node in the hierarchy represents a reservation with its *share* equal to its weight divided by the sum of the weights of its parents' children (including its own weight). For example, Figure 4 shows a hierarchy for disk bandwidth allocation. Scheduler nodes A and B each represent a reservation of 50% of disk bandwidth. At the next level, each of the queue nodes C, D, E, and F, represent a 25% reservation of of the overall disk bandwidth or 50% of their parent's reservation.

Eclipse/BSD uses a single-level Move-To-Rear List Scheduling (MTR-LS) [6] for CPU, a hierarchical YFQ scheduling for disk [5], hierarchical WF^2Q scheduling for output link [3]. The resource schedulers are dynamically reconfigurable in the sense that the scheduling hierarchy and the associated weights can be altered without stopping the scheduling itself.

Eclipse/BSD implements a `/reserv` file system that provides an API and namespace to access, use and reconfigure system resource schedulers. The directory nodes in the `/reserv` file-system correspond to the nodes in the scheduler hierarchy and thus represent reservations. Each resource has one directory such as `/reserv/cpu`, `/reserv/wd0` (disk), `/reserv/fxp0` (network) etc. The `/reserv` file system API provides means to add and delete reservations and to alter scheduling weights throughout hierarchy.

Eclipse/BSD implements a *tagging* mechanism to associate reservations with schedulable operations on objects. Reservations are associated with references to an object rather than to the object itself, thereby allowing multiple applications to share the object without having to share the reservation. For instance, in the case of an open file, the corresponding file descriptor is tagged with a reservation. For a socket, the corresponding socket desciptor is tagged with a reservation. In the case of CPU, each process is tagged with a reservation.

Signalled Receiver Processing (SRP) [7] is used for network input processing in Eclipse/BSD. SRP demultiplexes incoming packets before network and higher layer protocol processing and processes input packets in the context of respective CPU reservations. This is similar to the lazy protocol processing mechanism in Nemesis and RT-Mach.

Eclipse/BSD is being used to guarantee QoS to server applications, specifically to differentiate the performance of different web sites hosted on the same platform.

3.5 Scheduler For Multimedia And Real-Time Applications (SMART)

SMART [24] was designed to support execution of multimedia audio and video applications in conjunction with traditional non-real-time applications. SMART associates two attributes with executing tasks, namely, *urgency* and *importance*. An *urgent* task is one which has an immediate time constraint. An *important* task is one with a high priority, or one that has been least serviced proportionally among applications with the same priority. An urgent task may not be the one to execute if it requests more resources than its fair share. Conversely, an important task need not be run immediately. Importance of a task is expressed by its *priority* and *share* of resources whereas its urgency is expressed by *time constraints*. Urgency is specific to real-time applications, whereas importance is common to all applications.

Applications with higher priority are favoured when there is contention for a resource. Among applications of the same priority, each application receives an allocation of resource in proportion to its respective share whenever there is contention for resources. Importance of an application is represented by a *value-tuple* which has two components: priority and *biased virtual finish time* (BVFT). Priority is specified by the user and whereas BVFT is a dynamic quantity representing the degree to which each task has received its proportional share of resource. The SMART scheduler makes the scheduling decision as follows.

- If the task with the highest value-tuple is a non-real-time task, schedule that task.
- Otherwise create a candidate set consisting of all real-time tasks with higher value-tuple than that of the highest value-tuple non-real-time task.
- By using given deadlines and service time estimates of tasks, find the task with earliest deadline whose execution does not cause any tasks with higher value-tuple to miss their deadlines. This is achieved by considering each candidate in turn, starting with the one with the highest value-tuple. The algorithm attempts to schedule the candidates into a working schedule which is initially empty. A candidate is inserted in deadline order in this schedule provided its execution does not cause any of the tasks already in the schedule to miss its deadline. The scheduler simply picks the task with the earliest deadline in the schedule.
- If a task would miss its deadline, send a notification to inform the respective application.

SMART scheduler belongs to the category of best-effort real-time schedulers since it does not provide any guarantees to real-time applications that they will meet their deadlines. However it tries its best to meet the application deadlines. By introducing admission control, this drawback can be overcome and SMART can provide reservations with guaranteed resource availability.

3.6 Real-Time Extensions to Linux

In this section, we look at two real-time extensions to the Linux operating system - RTLinux from New Mexico Institute of Technology and KURT Linux from University of Kansas.

3.6.1 RT-Linux

Real-Time Linux (RTLinux) [2] is a popular hard real-time operating system which allows running special real-time tasks and interrupt handlers on the same machine as the standard Linux. As shown in Figure 5, the standard Linux runs as the lowest priority execution thread of RTLinux. A small hard-realtime kernel shares one or more processors with standard Linux. This allows the system to support applications to perform data acquisition, systems control and robotics, while still serving as a standard Linux workstation. Standard

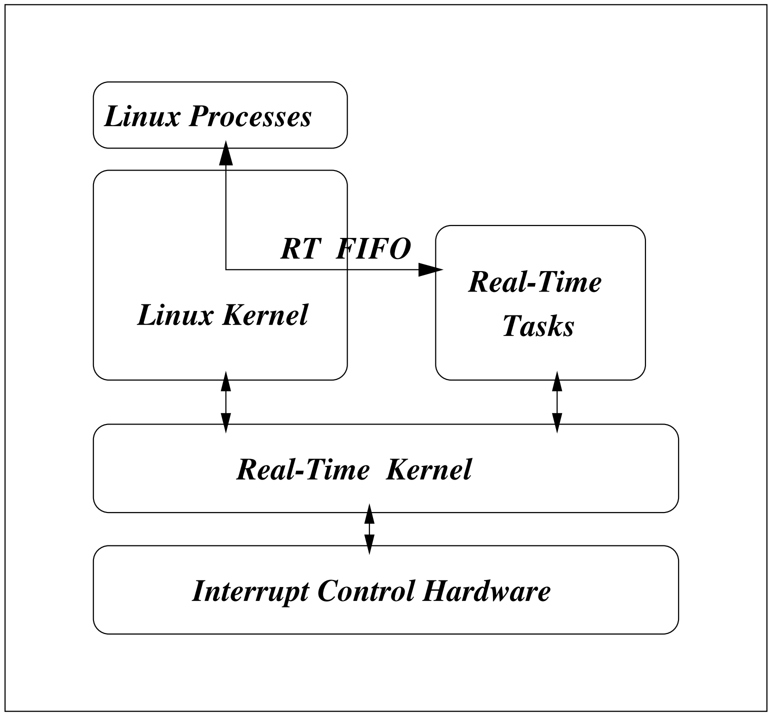

Figure 5: *Architecture of RTLinux. At the lowest level, a real-time kernel intercetps all hardeware interrupts and multiplexes them between real-time threads and normal Linux kernel. The Linux kernel runs as the lowest priority thread of the real-time kernel. Real-time threads can communicate with ordinary Linux processes via FIFO channels.*

Linux can provide scheduling precision of 10 milliseconds for threads using POSIX real-time capabilities (described in section 2.4). RTLinux goes a step further and provides submillisecond scheduling granularity.

The key idea behind RTLinux is the software emulation of interrupt control hardware. When Linux tells the hardware to disable interrupts, the realtime subsystem intercepts the request, records it, and returns to Linux. Linux is never allowed to disable hardware interrupts and is thus prevented from adding latency to the realtime subsystem's interrupt response time. When an interrupt arrives, the RTLinux kernel intercepts the interrupt and either invokes a realtime handler for the interrupt or if the handler indicates that it wants to share the interrupt with Linux, the interrupt is marked pending. When Linux enables interrupts again, any pending interrupts are delivered and the Linux handlers are invoked.

The scheduling among real-time processes could be either simple priority driven scheme, earliest-deadline first policy, or a rate-monotonic scheduling. The real-time processes can communicate with non-real-time Linux processes using FIFO queues or shared memory. Real-time tasks are written as kernel modules in RTLinux and run in kernel address space.

RTLinux is a practical choice for dedicated systems not working under heavy loads. RTLinux does not have real-time scheduling for resources other than the CPU. There is no mechanism for integrated resource management and QoS guarantees which are essential under heavy loads.

3.6.2 KURT Linux

KURT Linux [31] adds real-time capabilities to the standard Linux operating system by modifying its core functionalities. This is in contrast to RTLinux which leaves the standard Linux kernel untouched but adds a real-time layer between standard Linux and the hardware. Another difference between RTLinux and KURT Linux is that the former is designed to support hard real-time applications while the latter is designed for *firm* real-time applications.

KURT Linux improves the resolution of standard Linux timer. In standard Linux, hardware timer is programmed in *periodic mode* to interrupt the CPU at a rate of once every of 10 milliseconds. On one hand, this rate is not suitable for real-time applications that have timing constraints at a finer granularity. On the other hand, simply increasing the periodic rate at which hardware timer interrupts the CPU is not good for overall system throughput since too much time is spent by the OS in processing the timer interrupts. KURT Linux gets around this problem by programming the hardware timer in *one-shot mode*, i.e., as an aperiodic device that, whenever required, can be programmed to generate an interrupt at microsecond resolutions. The price to be paid is increased overhead in reprogramming the timer chip every now and then.

KURT Linux implements an explicit planning based real-time CPU scheduler. The real-time applications need to specify explicitly the times at which their real-time events need to occur. The system consists of a KURT core and real-time modules (RTMods). KURT core is responsible for planning a schedule for real-time events and invoking appropriate RTMods according to the plan. The RTMods are loadable kernel modules that implement the functionality of specific real-time applications and execute in kernel mode. RTLinux also adds support for VBR (variable bit rate) class of traffic in the standard implementation of ATM under Linux. It also modifies the disk subsystem to reduce durations in which interrupts are disabled.

3.7 Real-Time Extension to Windows NT

Windows NT was designed as a general purpose operating system. However, its popularity and market forces have pushed Windows NT towards use in real-time environments. By itself, Windows NT supports preemptive priority based scheduling with four process priority classes and 32 thread priority levels. However, there is no support for priority inheritance in Windows NT. Windows NT was not designed with predictability in mind, hence it is not suitable for use in hard real-time environments. [27] studies how Windows NT can be used to support real-time applications that can tolerate delay/response times in the range of tens to hundreds of milliseconds and to what extent unpredictable parts of Windows NT can be masked.

As a more aggressive approach, efforts have been made towards extending or modifying Windows NT to support real-time applications. There are essentially two approaches. In the first approach, a real-time OS coexists with Windows NT. In the second approach, real-time kernel becomes part of Windows NT and executes as a device driver. We take a look at some of these approaches below.

- INtime [17] is a real-time kernel, developed by Radisys, which runs along with Windows NT but in a separate address space. The two operating systems communicate via extensions to Windows NT API. INtime supports real-time threads and runs Windows NT as its lowest priority thread. It modifies the hardware abstraction layer (HAL) in order to prevent Windows NT from masking interrupts reserved for real-time tasks. This approach is similar to the one used in RT-Linux to run Linux kernel as a lowest priority thread of another real-time kernel.

- Rtx is a real-time extension to Windows NT [8] developed by VenturCom. The Rtx subsystem primarily consists of two components - a real-time application programmer's interface (RTAPI) and a real-time subsystem (RTSS) which resides inside Windows NT as a device driver. RTSS supports thread priorities and handles priority inversion. The threads are scheduled by priority and within same priority, in round robin order. All RTSS threads run before any NT threads can run. RTX also support fast clocks and timers. Interrupts are handled by modifying the hardware abstraction layer (HAL) of Windows NT, in order to isolate interrupts between NT and the RTSS.

Two other real-time operating systems from the Windows family are described below.

- The Rialto [14] operating system, developed at Microsoft Research, is designed to support firm real-time applications by means of CPU reservations of the form "reserve X units of time out of every Y units". A *scheduling graph*, precomputed from the set of CPU reservations, determines which tasks execute in a given *graph cycle*. Scheduling among feasible tasks in a graph cycle is done in EDF order. Applications negotiate the desired QoS parameters with a resource planner object which tries to maximize the user perceived utility of system resource usage. The concept of CPU reservation from Rialto operating system has been applied to the NT operating system in [15].

- Windows CE [38] is a modular real-time embedded operating system with a small footprint. It provides priority based scheduling of threads with 256 priority levels and handles priority inversion using priority inheritance mechanism. Interrupt processing is performed in two steps. First an interrupt service routine (ISR) handles the interrupt and wakes up an interrupt service thread (IST). IST performs

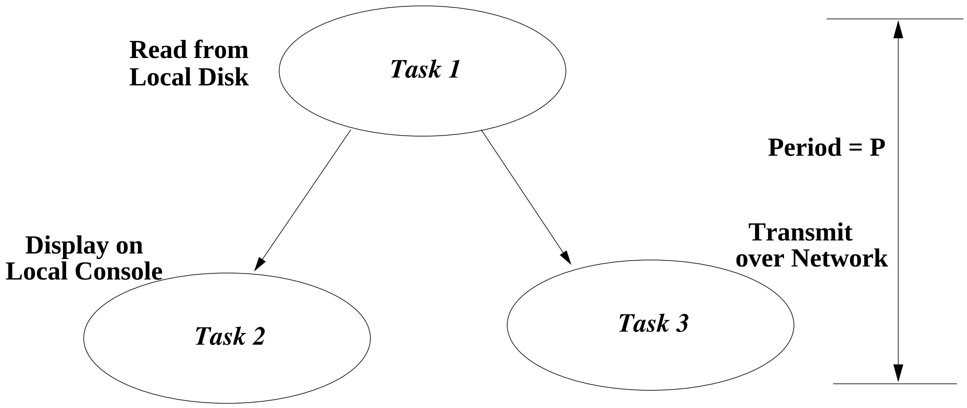

Figure 6: *Example of Task Precedence Graph of a video playback application in IRS. Task 1 reads a frame of video from local disk. Task 2 displays the frame on local video console. Task 3 ships the frame to a remote client. Tasks 2 and 3 become eligible for execution only upon completion of Task 1, but can proceed concurrently after becoming eligible.*

additional interrupt processing if it is the highest-priority thread ready to run. Most of the interrupt handling actually occurs within the IST. Windows CE lets a thread set timer or go to sleep with resolution of 1 millisecond.

4 Integrated Real-Time Resource Management and Scheduling

Real-time multimedia applications that utilize multiple systems resources, such as CPU, disks, and network links, require "coordinated" scheduling of these resources to meet their application-level deadlines. Most state-of-the-art operating systems support at best independent resource allocation and deadline-driven scheduling for individual resources, but lack a uniform model to to coordinate efficient resource allocation and scheduling of heterogenous resources the same machine. Some research efforts in this direction are discussed in this section.

4.1 IRS

At Stony Brook, we have designed and implemented a unified framework for resource allocation and real-time scheduling called Integrated Real-Time Resource Scheduling (IRS) [11] which provides soft real-time guarantees to applications requiring access to multiple resources. In the IRS model, real-time applications are modeled as a periodic execution of a Task Precedence Graph (TPG) as shown in Figure 6. A TPG is a representation of individual tasks of an application and the precedence constraints among them. Each task in a TPG corresponds to consumption of one type of resource. The TPG, its period, jitter and bandwidth requirements of individual tasks, are explicitly specified when an application registers itself as a real-time process in the beginning of program execution.

The novel and most important component of the *IRS* framework is its resource allocation and deadline assignment algorithm. Given the TPG and period of a mutimedia application, *IRS* transparently allocates heterogenous resources and computes delay budget (deadline per period) for each task under the constraint that the entire TPG must be completed within one period. If TPG can complete within the specified period it is admitted for execution. Given the current resource availability in the system, the algorithm starts by allocating the unused resource capacity to the tasks of TPG and computes the minimal delay budget required to complete each task. These minimal delay budgets are then iteratively *relaxed* by apportioning the available slack in delay budget of the TPG according to the load on each resource. The algorithm stops the relaxation process at a heuristically optimal resource allocation where the TPG can still meet its periodic deadlines. The advantage of the relaxation mechanism is that it aims to balance the load among different resources and thus, in the long term, allows more real-time applications to be admitted into the system.

An *IRS* prototype has been implemented as part of LINUX operating system running on Intel Pentium-based machines. Figure 7 shows the software architecture of *IRS*. Real-time applications requiring *IRS*

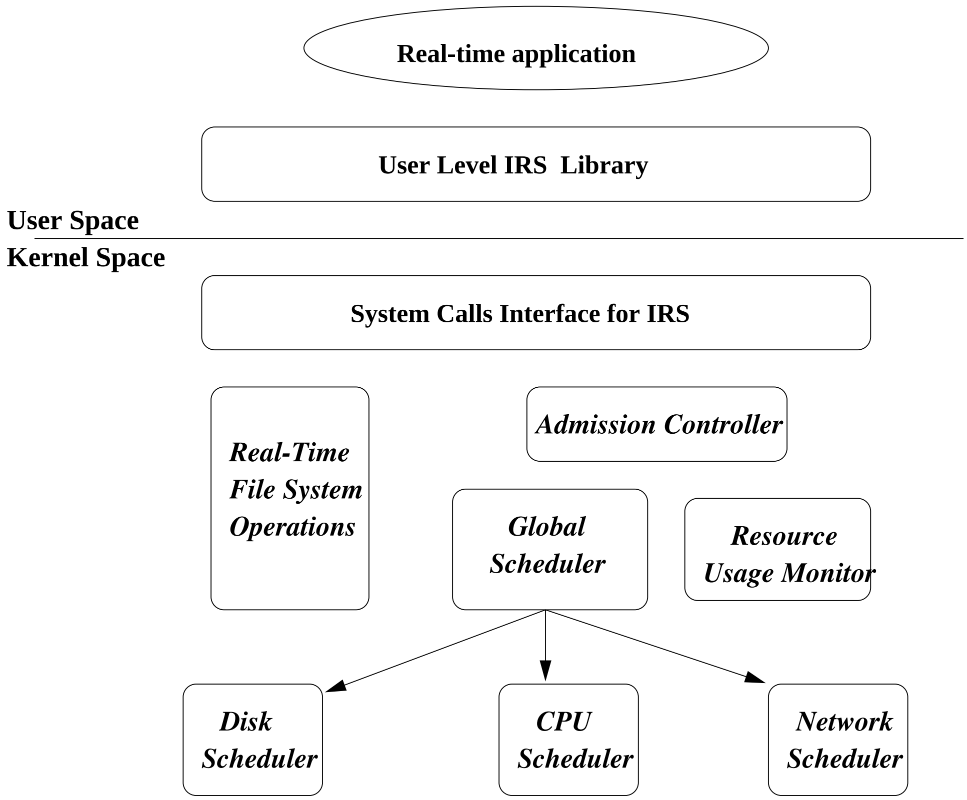

Figure 7: *The software architecture of the IRS. Real-time applications link to a user-level* IRS *library that uses* IRS *specific system calls to communicate resource requirements to the OS. An admission controller makes admission decisions and allocates resources to applications. A global scheduler dispatches tasks from application's TPG to individual resource schedulers according to their precedence constraints. Real-time file-system operations allow real-time I/O through disk subsystem. A resource usage monitor tracks applications' resource usage and provides feedback to applications.*

services are programmed using an *IRS* API and linked with a user level *IRS* library that uses a set of *IRS* specific system calls to interact with the kernel subsystem. An *admission controller* makes admission decisions and performs task deadline assignments at the time a new application registers itself as a real-time process. A usage monitor constantly keeps track of the resource consumption of individual tasks to detect misbehaved applications. *IRS* is based on a two-level resource scheduling architecture, which consists of a *global scheduler* that has a complete knowledge of each application's task graph and resource requirements, and a set of *local schedulers*, each corresponding to a particular resource. The *global scheduler* is responsible for making sure that a task's dependencies are all satisfied before it is placed in the request queue of the corresponding resource. *Local schedulers* manage individual resources and perform scheduling decisions based on both task deadlines and resource utilization efficiency. The global scheduler acts as a glue between local resource schedulers using its global knowledge of task deadlines and resource requirements for a given application. To the global scheduler, individual resource schedulers are black-box building blocks whose specific scheduling algorithm is completely hidden. Real-time file-system operations allow real-time I/O through disk subsystem [12].

4.2 Continuous Media Resource Model

Continuous Media Resource (CM-resource) model, presented in [1], is a theoretical framework designed to provide end-to-end performance guarantees to applications using *continuous media* (such as digital audio and video). *Continuous media* is termed as such because data in such media is perceived as changing continuously over time. In CM-resource model, applications reserve resources based on their worst-case needs, and resources offer hard performance bounds. However, unlike traditional hard real-time systems, the model allows resources to work ahead, making more resource capacity available to bursty non-real-time traffic. A *meta-scheduler* coordinates among CPU, network and file system components, negotiating end-to-end guarantees on behalf of the clients. However, the meta-scheduler is not involved in the actual usage of the components.

Workload in CM-resource model is described in terms of *linear bounded arrival process* of discrete *messages* representing units of work. The client specifies the workload in terms of its maximum message size, maximum message rate and workahead limit. The workahead limit allows client applications and devices to generate bursts of messages that would otherwise exceed this rate.

The meta-scheduler establishes *sessions* (reservations) with each of the resources to be used by a client before the client can access the resources. A *compound session* S is a sequence of individual sessions $S_1 ... S_N$ in which output interface of S_i is the input interface of S_{i+1}. Each resource exports three procedural interfaces for creating sessions, namely `reserve()`, `relax()`, `free()`. These functions are used as follows.

- A call to `reserve()` requests a session with a specific workload and the smallest possible logical delay.
- Every resource that can accept the session returns the corresponding minimum possible logical delay bound D_i for the session and reserves enough capacity to provide this bound.
- The meta-scheduler chooses a specific delay bound $D_i^{'} \geq D_i$ for each resource such that the total end-to-end cost of using multiple resources is minimized under the constraint that the sum of D_is is smaller than total end-to-end delay D required by the requesting client. The meta-scheduler calls `relax()` to increase the delay bound to $D_i^{'}$ corresponding to each resource.
- When the resource usage is finished, `free()` deletes the session.

Like the IRS model described above, CM-resource model addresses efficient resource allocation and end-to-end guarantees for applications using multiple resources. However, unlike the IRS framework which handles multiple resources usage related by arbitrary *directed acyclic graph* of precedence constraints, the framework of CM-resource model deals only with a compound session consisting of linear chain of sessions on an end-host. Also, the definition of cost funtions for each resource and the exact algorithm to solve minimal cost delay assignment problem is left unspecified.

4.3 QoS Resource Allocation Model (Q-RAM)

Modern multimedia applications need to satisfy performance requirements along *multiple QoS dimensions*, such as sampling rate of audio data, end-to-end latency of audio stream, video frame rate, resolution of video, etc. In Q-RAM [18], the problem of apportioning *multiple finite resource* to satisfy the QoS needs of multiple applications along *multiple QoS dimensions* is considered.

Each application is assumed to have a finite set of *discrete quality level* choices along each QoS dimension. Each discrete quality level can be achieved by a set of combinations of resource allocation. For instance, a video server can compress files to reduce network bandwidth requirements at the expense of increased CPU cycle requirement. Alternatively, the video player could avoid compression to save CPU cycles but consume higher network bandwidth. The application is responsible for specifying its *QoS profile*, i.e. user's possible choices in quality space, and *Resource Profile*, i.e. the possible mappings between resource requirements and points in quality space.

Every application accrues a certain *utility* (value) along each QoS dimension as resources are allocated to it. *Application utility* is defined as a weighted sum of dimension-wise utilities. For a system hosting multiple applications, the overall *system utility* is defined as weighted sum of individual application utilities.

The MRMD problem can be stated as follows. Given a set of applications $T_1, ..., T_n$, assign qualities (q_i) and allocate resources (r_i) to applications such that the system utility is maximized, under the constraint that

- Each application gets the a minimum requested quality along each dimension.
- The sum of each type of resource allocated to each application does not exceed the corresponding maximum resource available with the system.

The above problem is shown to be NP-hard in [18]. The work compares three strategies to solve this problem. Two traditional methods of dynamic programming and mixed integer programming can be used to compute optimal solutions to this problem. However, their running times are quite high to be of any practical value. A third strategy, based on *local search* technique, is several orders of magnitude faster than the first two techniques and yields solutions that are within 5% of the optimal solutions. However, unlike IRS, the solutions proposed are based on static set of applications to be executed at the same time and do not consider dynamic arrival and completion of applications.

4.4 Stony Brook Video Server and Rether

Stony Brook Video Server (SBVS) [9] is an enterprise video server which provides an integrated scheduling mechanism to guarantee both network and disk transport bandwidth. It uses a generic technique called cyclic scheduling to guarantee I/O bandwidth. While users are consuming data over the network during cycle i, the disk subsystem fills the buffers with data needed for cycle $i + 1$.

SBVS guarantees network bandwidth to the video server application by means of Rether protocol [37]. Rether is a real-time protocol over shared communication media such as ethernet or wireless network. It implements a software token passing mechanism at data link layer among a set of cooperating nodes in the network and guarantees deterministic channel access once every token rotation cycle.

4.5 Resource Brokers Architecture

In [39], a framework and simulation results for a QoS and contention aware, multi-resource reservation algorithm is presented. This work addresses the problem of how to determine the best end-to-end QoS level for an application under the constraint of current resource availability. This goal is different from that of *IRS* which tries to increase the overall resource utilization efficiency in the system while satisfying the end-to-end QoS requirement of the application. Under this framework each node in the distributed component based service environment contains Resource Brokers (RB) for reservation of individual resource and a QoS Proxy for coordinating the reservation activities of individual RBs. An RB is responsible for tracking current resource availability and reserving and dereserving the corresponding resource. The QoS proxy collects resource availability information from individual RBs and executes a multi-resource reservation algorithm. This framework assumes that each resource exports discrete levels of QoS that are available to applications and that translation functions are available to map multiple QoS levels of resource to the actual resource requirements. Given the end-to-end QoS requirement of an application and the translation function between QoS levels and the resource requirement, the multi-resource reservation algorithm computes an end-to-end multi-resource reservation plan using a Quality Resource Graph (QRG). QRG is a directed acyclic graph with transitions between nodes representing input and output quality levels of service components. The algorithm tries to maximize the application's QoS level under the constraint of current resource availability. The framework addresses the issue of reservation but not that of scheduling of multiple heterogenous resources in order to guarantee end-to-end QoS to application.

5 Summary

In this report, we studied the various issues involved in supporting real-time applications in a general purpose operating system. We also looked at representative systems that have had significant influence on the progress of real-time operating systems research. Based on our study, Table 1 presents a classification of the different systems according to basic properties of real-time operating sytems as described in Section 2.

6 Future Trends

An enormous amount of research effort has gone into design and implementation of real-time systems and their unique properties motivate many worthwhile frameworks and implementations developed over the last two decades. However, the state-of-the-art real-time systems are still far from being considered fully mature. Below we list some of the open research issues in supporting real-time applications in a GPOS.

System	Class of Applications	Scheduling Policy	Real-Time Threads Support	Priority Inheritance Mechanism	Integrated Resource Allocation	Predictable Memory Management
Spring	Hard	Explicit Planning	Yes	N/A	Yes	Yes
RT-Mach	Firm	Priority, EDF, Round Robin	Yes	Yes	No	No No
SMART	Soft	Priority, Proportional Share	No	No	No	No
Nemesis	Soft	EDF Variant	No	N/A	No	Yes
Eclipse/BSD	Soft	Heirachical Proportional Share	No	N/A	No	No
RTLinux	Hard	Priority,EDF, RMS	Yes	No	No	No
KURT Linux	Firm	Explicit Planning	Yes	N/A	No	No
INtime	Hard	Priority	No	Yes	No	Yes
Rtx	Hard	Priority	No	Yes	No	No
Rialto	Soft	EDF	No	N/A	No	No
Windows CE	Soft	Priority	No	Yes	No	No
IRS	Soft	Integrated Resource Scehduling	Yes	No	Yes	No

Table 1: This table summarizes the support for six basic real-time features by various research and commercial real-time systems discussed in this report. These features are discussed in Section 2.

- One area which requires further research is the interface by which application programmers specify their resource requirements to the system. There is still a considerable lack of consensus in the research community towards a common application programming interface for developing real-time applications. There are practically as many interfaces as there are implementations of real-time systems. The interfaces either require the programmer to do much of the hard work in figuring out the resource requirements or make too many simplifying or restrictive assumptions as to the functionality exposed to the programmer. Two significant efforts towards standardization are POSIX and Win32 programming interfaces. However these are still far away from addressing full range of flexibility required by the real-time application programmers.

- Another area of research that has received little attention is co-ordinated allocation and scheduling of heterogenous resources in a real-time system. This was discussed at some length in section 4.

- Fault Tolerance issues in real-time systems is another area which requires further research. Time bound fault-recovery is important in systems where applications have timing constraints and their real-time guarantees will be violated if the fault-recovery times cannot be bounded.

- Verification of real-time systems is another interesting area of research. Techniques to methodically prove that a real-time system will actually meet the guarantees that it promises and the entire system itself performs predictably is worthy of further exploration.

- Detailed accounting of kernel activities that are performed on behalf of user applications is another area requiring further exploration. Microkernel based operating systems, such as RT-Mach and Nemesis, try to overcome this problem by pushing the kernel functionalities into user space. However, considering that monolithic kernels are still dominant as general purpose operating systems, and also considering recent attempts to build real-time functionality into existing monolithic kernels such as Unix and Windows, it becomes important to have detailed accounting of kernel activities on behalf of user applications. This is important in order to appropriately charge resource usage to user applications which remain in kernel mode most of the time.

- Relational database management systems have QoS requirements such as guaranteed number of transactions per unit time and bounded response times to relational queries. Recent explosion in e-commerce and other online transactions has increased the importance of being able to provide such guarantees. Providing such guarantees to database systems from an underlying general purpose operating system forms another interesing area of research.

References

[1] D.P. Anderson, "Metascheduling for Continuous Media", ACM Transactions on Computer Systems, Vol. 11, No. 3, August 1993, pp. 226-252.

[2] M. Barabanov and V. Yodaiken, "Real-Time Linux", Linux Journal, February 1997.

[3] J.C.R. Bennett and H. Zhang, "WF^2Q: Worst-case Fair Weighted Fair Queuing", In Proceedings of IEEE INFOCOM, San Francisco, pp 120-128, March 1996.

[4] J. Blanquer, J. Bruno, E. Gabber, M. Mcshea, B. Ozden, and A. Silberschatz, "Resource Management for QoS in Eclipse/BSD", Proceedings of the FreeBSD 1999 Conference, Berkeley, California, October 1999.

[5] J. Bruno, J. Brustoloni, E. Gabber, M. McShea, B. Ozden, and A. Silberschatz, "Disk Scheduling with Quality of Service Guarantees", ICMCS99, June 1999.

[6] J. Bruno, E. Gabber, B. Ozden, A. Silberschatz, "The Eclipse Operating System: Providing Quality of Service via Reservation Domains", In Proceedings of USENIX 1998 Annual Technical Conference, New Orleans, Louisiana, pp 235-246, June 1998.

[7] J. Brustoloni, E. Gabber, A. Silberschatz and A. Singh, "Signaled Receiver Processing", Proceedings of the USENIX'2000 Annual Technical Conference, June 2000, San Diego, CA, pp. 211-223.

[8] B. Carpenter, M. Roman, N. Vasilatos and M. Zimmerman, "Rtx Real-Time Subsystem for Windows NT", USENIX Workshop on Windows NT System Engineering, August 1997.

[9] T. Chiueh, C. Venkatramani, M. Vernick, "Design and Implementation of the Stony Brook Video Server", In Software – Practice and Experience, January 1997.

[10] A. Demers, S. Keshav, S. Shenkar, "Analysis and Simulation of a Fair Queuing Algorithm", Internet Research and Experiences, Vol 1., 1990.

[11] K. Gopalan, and T. Chiueh,"Integrated Real-Time Resource Scheduling", Technical Report TR-56, Experimental Computer Systems Labs, Department of Computer Science, SUNY at Stony Brook.

[12] K. Gopalan and T. Chiueh, "Real-Time Disk Scheduling Using Deadline Sensitive SCAN", Technical Report TR-92, Experimental Computer Systems Labs, Dept. of Computer Science, State University of New York, Stony Brook, NY, January 2001.

[13] V. Jacobson, K. Nichols, K. Poduri, "An Expedited Forwarding PHB", RFC 2598, June 1999.

[14] M. Jones J. Barrera III, A. Forin, P. Leach, D. Rosu, and M. Rosu, "An overview of the Rialto Real-Time Architecture", In Proc. of the Seventh ACM SIGOPS European Workshop, 249-256, Sept 1996.

[15] M. B. Jones and J. Regehr, "CPU Reservations and Time Constraints: Implementation Experience on Windows NT", Proceedings of the Third USENIX Windows NT Symposium, Seattle, Washington, pages 93-102, July 1999.

[16] M. Humphrey, J.A. Stankovic,"Predictable Threads for Dynamic, Hard Real-Time Environments", IEEE Transactions on Parallel and Distributed Systems, Vol. 10, No. 3, March 1999, pp. 281-296

[17] INtime Web Pages : http://www.TenAsys.com/intime.htm

[18] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen, "A scalable solution to the multi-resource QoS problem", Proceedings of IEEE RTSS'99, December 1999.

[19] C. Lee, R. Rajkumar, and C. Mercer, "Experiences with Processor Reservation and Dynamic QoS in Real-Time Mach", In Proceedings of Multimedia Japan, May 1996.

[20] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar, "Predictable Communication Protocol Processing in Real-Time Mach", In the proceedings of IEEE Real-Time Technology and Applications Symposium, June 1996.

[21] I.M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barharn, D. Evers, R. Fairbairns, and E. Hyden, "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications", IEEE Journal on Selected Areas In Communications, Vol. 14, No. 7, pp. 1280-1297, September 1996.

[22] C.L. Liu and J.W. Layland, "Scheduling algorithms for multiprogramming environment in a hard real-time environment," Journal of the ACM, 20(1), 47-61, (1973).

[23] T. Nakajima, T. Kitayama, H. Arakawa, H. Tokuda, "Integrated Management of Priority Inversion in Real-Time Mach", IEEE Real-Time Systems Symposium, December 1993

[24] J. Nieh and M.S. Lam, "The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications", Sixteenth ACM Symposium on Operating System Principles, St. Malo, France, October 1997.

[25] D. Niehaus, "Program Representation and Execution in Real-Time Multiprocessor Systems", Ph.D. Thesis, University of Massachussetts, Amherst, 1994.

[26] A.K. Parekh and R.G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks-the single node case", IEEE/ACM Transactions on Networking, pp 347-357, June 1993.

[27] K. Ramamritham, C. Shen, O. Gonzalez, S. Sen, S. Shirgurkar, "Using Windows NT for Real-Time Applications : Experimental Observations and Recommendations", In Proceedings of Real-Time Technology and Applications Symposium, Denver, June 1998, pp. 102-111.

[28] K. Ramamritham and J. A. Stankovic, "Scheduling Algorithms and Operating Systems Support for Real-Time Systems", Proceedings of the IEEE, Vol. 82, No. 1, pp. 55-67, January 1994.

[29] K. Ramamritham, J.A. Stankovic, and P.F. Shiah, "Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems", IEEE Transactions on Parallel and Distributed Systems, Vol 1, No. 2, pp 184-195, Apr. 1990.

[30] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", IEEE Transactions on Computers, Vol. 39, No. 9, 1990.

[31] B. Srinivasan, S. Pather, R. Hill, S. Ansari, D. Niehaus, "A Firm Real-Time System Implementation Using Commercial Off-the-shelf hardware and Free Software", In Proceeding of Real-Time Technology and Applications Symposium, Denver, June 1998.

[32] J.A. Stankovic and K. Ramamritham, "A reflective architecture for real-time operating systems", in Advances in Real-Time Systems, Sang Son, Ed. Prentice-Hall, 1994.

[33] J. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Systems", IEEE Software, Vol. 8, No. 3, pp. 62-72, May 1991.

[34] R. Steinmetz "Analyzing the Multimedia Operating System", IEEE Multimedia, Vol. 2 No. 1, 1995

[35] H. Tokuda, T. Nakajima, and P. Rao,"Real-Time Mach : Towards Predictable Real-Time Systems", in Proceedings of USENIX 1990 Mach Workshop, Oct 1990.

[36] C. Uarrecoechea, A. T. Campbell, and L. Hauw, "A survey of QoS architectures", In 7th International Workshop on Quality of Services (IWQoS'99), 1999.

[37] C. Venkatramani, "Design, Implementation and Evaluation of Rether: A Real-Time Ethernet Protocol", Ph.D. Dissertation, Department of Computer Science, State University of New York at Stony Brook, December 1996.

[38] Windows CE Web Pages : http://www.microsoft.com/windowsce.

[39] D. Xu, K. Nahrstedt, A. Viswanathan, D. Wichadakul, "QoS and Contention-Aware Multi-Resource Reservation", In Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC-9), Pittsburgh, PA, August 2000.