Resource Scheduling in Real-Time Systems: A Survey

Bryan Tower, Derrick Kondo, Manamohan Mysore, and Wesley Leong
(btower,dkondo,mmysore,wleong)@cs.ucsd.edu

November 30, 1999

1 Introduction

Real-time systems are computer systems in which the correctness of an execution depends not only on the logical results but also on the time at which the results are produced. Examples of real-time systems include flight control, space shuttle avionics, robotics and multimedia. Over the past decade, the increasing demand for such applications has motivated active research in the area of real-time operating environments. As time progresses and computers reach every nook and cranny of our daily lives, this field is bound to become not merely more interesting but also indispensable. In this survey paper, we study three important parts of a real-time operating system, namely task scheduling, protocol processing, and disk scheduling. These components play an especially crucial role in multimedia systems, where time guarantees for CPU, network and disk operations are important; further, these components are bound to assume greater importance as the demand for multimedia grows. In the following sections, we shall give a flavor of the issues involved in the design of these components and succinctly describe the major solutions that have been developed.

The paper is organized into four sections. In section 2, we study and classify existing real-time task scheduling algorithms. Where appropriate, we illustrate them with working examples. Section 3 examines the recent work done in the field of real-time protocol processing. Real-Time Mach and the Real-Time Upcall Mechanism are two examples that we present. Section 4 describes five real-time disk scheduling algorithms and briefly examines their merits and demerits. In section 5, we conclude with a brief account of the recent trends observed in this field.

2 Real-Time Task Scheduling

Task Scheduling refers to the apportioning of CPU time among the various tasks in a system. The term *task* is a broad one. It could refer to a process on UNIX, a thread in Mach, or collections of these. Task scheduling algorithms form the backbone of any real-time system since almost every subsystem depends on this facility.

Based on whether the system does schedulability analysis to schedule tasks, the stage at which the analysis is done, and the way the results of the analysis are used at run-time, task scheduling algorithms can be classified into four kinds: *static table-driven, priority-based preemptive, dynamic planning-based and dynamic best-effort*. In the following subsections, we describe the first three classes in detail, point out some specific issues, and where appropriate, illustrate them with existing implementations. We do not spend too much time on the dynamic best-effort scheduling strategy commonly used in today's non-real-time systems since it is a well known concept and in addition, does not provide the much sought-after real-time guarantees.

2.1 Static Table-Driven Scheduling

The static table-driven strategy assumes that the tasks to be scheduled are periodic. It divides time into convenient *time periods* and determines the schedule for a given period of time before that time period begins. Normally, this time period is the Least Common Multiple of the time periods of all tasks. It also statically pre-allocates resources that will be needed for that time period. This approach has the potential of providing deterministic guarantees. Thus, this approach is well-suited for hard real-time systems and systems that handle safety-critical tasks. However, this approach is very inflexible. Adding a new task could require the schedule to be rebuilt from scratch. On the other hand, using the slack time available in an existing schedule to allow new tasks could lead to a suboptimal schedule.

One example of such a strategy is the ARTS kernel [5], which applies a static table-driven scheduling variant to manage real-time threads in a distributed environment. The schedulability test used to draw up the schedule is based on timing constraints, worst-case execution time, and a special *value* function to gauge the thread's utility to the system. The system provides run-time support to allocate the slack time available in the schedule to incoming real-time threads. This approach, while keeping the run-time overhead small, provides only limited flexibility. Other examples of the use of this strategy include MARS [3], and the Spring Kernel[1] [2].

2.2 Priority-Driven Preemptive Scheduling

A priority-driven scheduling approach assigns priorities to tasks and executes them based on priority ordering; it does not draw up a schedule in advance. This strategy is very common in time-sharing systems. In real-time systems, it is crucial that the priority assignments be made according to timing constraints and relative importance. This strategy is very well understood, and this contributes to its wide popularity. In addition, it has the advantage of being very simple to implement. The important issues that need to be considered by a priority-driven scheduler in practice are priority inversions and starvation.

Three common priority assignment policies are Rate-Monotonic (RM), Earliest Deadline First (EDF), and Least Laxity First (LLF). We describe them briefly below.

- *Rate-Monotonic*: The basic RM algorithm is used to schedule periodic tasks, assigning them static priorities based on their period. Tasks of smaller time periods get higher priorities. It can be proved that RM can be used to schedule a single CPU to guarantee the real-time constraints of a large number of *periodic* tasks provided that the total CPU utilization is less than $ln2 = 0.69$ [1]. One obvious limitation of this policy is that the schedulable bound is less than 100%.

- *Earliest Deadline First*: EDF is a *dynamic priority assignment* algorithm. The task with the earliest deadline has the highest priority. The closer a deadline gets, the higher the priority becomes. In this policy, starvation can never happen. Since this priority scheme is dynamic, it involves a run-time overhead, whose effect would need to be limited. One disadvantage of EDF is that critical tasks may not get priority over non-critical tasks because the closeness of deadlines is the only deciding factor.

- *Least-Laxity-First*: LLF is similar to EDF, but it uses a task's laxity instead of deadline to prioritize tasks. Laxity is defined by the following equation:

  $$laxity = deadline\_time - current\_time - CPU\_time\_needed$$

  Laxity is a measure of the leeway available for scheduling a task. LLF takes into account the execution time while of a task

[1]Spring also uses a dynamic planning-based algorithm

EDF does not. While this is an advantage in obtaining more flexible schedules, it allows the possibility of having to throw away more important tasks while retaining less important tasks with lesser laxities.

In addition to these common policies, one finds strategies that use combinations of existing policies to provide flexibility. One such system is Chimera II, which is discussed below.

2.2.1 Chimera II

Chimera II [1] uses a policy called Maximum Urgency First (MUF). This policy combines both dynamic and fixed priority scheduling. This *mixed-priority* scheduling allows tasks with various execution times and changing deadlines to be scheduled with some predictability.

MUF has three stages of priorities: *criticality*, *dynamic priority*, and *user priority*. Criticality has the highest precedence. Like RM, MUF assigns higher criticalities to tasks with shorter execution periods. Criticality creates a *critical set*, which is a group of *critical* tasks totalling less than 100% CPU utilization. The rest of the tasks have appropriate low criticalities. The second level of priorities is assigned dynamically. The dynamic priority value is proportional to the inverse value of the task's laxity. The lesser the laxity, the higher its dynamic priority. The third level of priorities is static user priorities, which are left to the user.

The selection of the next task to be executed happens as follows. If unique, the task with the highest criticality goes next. If not, among the most critical tasks, the one with the highest dynamic priority is chosen; if the dynamic priorities do not break this tie, the user priorities are compared. If all three layers of priorities fail to determine the precedence of the conflicting tasks, the first task to appear in the scheduler will be chosen.

This mixed-priority scheduling scheme has certain advantages over other schemes. The critical set guarantees that each high priority task will be executed before the deadline while enabling non-critical tasks to be squeezed into the remaining time chunks. In this way, it enhances CPU utilization. With dynamic priorities, tasks can reconfigure their execution lengths and execution periods within reasonable bounds without leaving their current criticality range. Tasks with varying execution loads can comfortably coexist.

Another advantage of MUF is that it does not take much implementation effort. It can be easily implemented by mapping criticality, dynamic priority and user priority into one number in that order with criticality occupying the most significant part and user priority occupying the least significant part. This keeps the implementation not only simple but also efficient since the comparison of priorities of two tasks is a one-step process.

2.3 Dynamic Planning-Based Scheduling

In dynamic planning-based scheduling, the feasibility of a schedule is checked at run-time. A dynamically arriving task is accepted for execution only if it is possible to execute it before its deadline. When a task is accepted, it is guaranteed to meet its timing constraints based on certain assumptions about worst case execution time, resource needs, the nature of faults in the system, and other factors. If these assumptions hold, a task will meet its deadline. Dynamic algorithms do not know beforehand the arrival times of tasks and hence can not guarantee optimal performance. Spring scheduling mechanism is an example of a dynamic planning-based scheduling algorithm.

In a distributed environment, a dynamic scheduler has greater flexibility. If one site[2] cannot complete a task, it can try to schedule that task on another site with a lighter load. One can think of four approaches [4] to decide which site to push this task onto. The

[2] a generic name for a processing entity

random scheduling algorithm sends the task to a randomly selected site. The task shuttles around until it finds a site where it can be run, or it gets discarded if it does not find one. In the *focused* addressing algorithm, the task is sent to a site which is believed to have sufficient resources to meet the timing constraints. The *bidding* algorithm accepts bids from the other sites in the distributed system and sends the task to the winner. The *flexible* algorithm is a combination of the focused addressing and bidding strategies.

Dynamic planning-based strategies usually handle aperiodic tasks as efficiently as periodic tasks. This makes them attractive for event-driven real-time systems that handle tasks of aperiodic nature. While providing this flexibility, a price is paid in terms of the run-time overhead involved in deciding feasibility.

2.3.1 Spring

Spring is an example of a system that uses a dynamic planning-based scheduler. It looks at the set of tasks, also accepting tasks that arrive dynamically, and tries to guarantee that all the tasks will complete by their timing constraint. Spring was designed for an environment that is dynamic, large, complex and evolving; for this reason it was designed to be very flexible.

The Spring scheduling algorithm uses two main criteria for classifying tasks, namely *importance* and *timing requirements*. The importance of a task reflects its value to the system. Based on importance and timing requirements, Spring characterizes tasks as *critical*, *essential*, and *unessential*. Critical tasks must meet their deadline, or a catastrophic result may occur. Essential tasks are necessary for the system but will only degrade performance if their timing constraints are not met. Unessential tasks may or may not have deadlines, and they are allowed to execute when they do not affect critical or essential tasks. To meet its goal of supporting tasks of varied nature, Spring incorporates a dynamic scheduler for normal tasks and a static table-driven scheduler for safety-critical tasks.

Further, scheduling responsibilities are organized into four levels. At the lowest level, there are multiple *dispatchers*. Dispatchers take the next task from the list of previously guaranteed tasks and assigns it to a processor.

At the second level is the *local scheduler*, which implements dynamic planning. The local scheduler tries to make sure that tasks meet their guarantees. An incoming task is assessed to determine if it can be incorporated into the list of guaranteed tasks. The local scheduler is responsible for ordering the tasks in the list so that the dispatcher can dispatch the next task in the list.

The third level of scheduling is the *distributed scheduler*. If a task cannot be guaranteed locally, this level tries to find some other processor where it can be run with guarantees. The fourth level is a *metalevel controller*, which adapts various parameters and switches algorithms [2]. While this design provides flexibility, it takes its toll in terms of run-time overhead.

2.4 Dynamic Best-Effort Scheduling

In the best-effort scheduling approach, no schedulability test is done. The decision to run a task is based on a priority value which is computed based on the task's characteristics. In this environment, real-time scheduling algorithms like EDF and LLF do well as long as no overloads occur. However, since there is no test for guarantees and schedulability, best-effort algorithms lack predictability and are in practice sub-optimal. Examples of this strategy are found in most non-real-time systems and in some real-time systems.

3 Real-Time Protocol Processing

Over the last decade, there has been a surge of network-based continuous media (audio and video) applications, and this has motivated research into real-time communication support in operating systems. In what follows, we summarize research that has gone into the design of real-time protocol stacks and closely examine two of the most significant pieces of work ([6], [7]) in the field. Before we do so, it is worthwhile to examine a few issues that guide research in this area. First, real-time protocol processing depends heavily on task scheduling mechanisms in use. Almost all real-time systems map the problem of protocol processing into one of task scheduling. The issue then is to make this "mapping" as efficient as possible. Second, most real-time systems need guarantees on performance and this necessitates efficient mechanisms for admission control and enforcement. Third, it is important that the protocol stack not only keep tabs on application behavior but also "talk to" the network to ensure overall real-time guarantees. In this survey, we do not focus on the issues of admission control and network provisioning. Our goal is to provide an architectural birds-eye-view of real-time protocol stack design.

3.1 Four Different Approaches to Real-Time Protocol Stack Structuring

A study of research in this field reveals four approaches to achieving real-time protocol processing. We discuss each of these in turn.

1. **Prioritized processing**: This is a straightforward modification of FIFO queues into priority-based queues both for the sending and the reception of packets. This approach is very common in practice, the most well-known example being the 4.3BSD Operating System. This scheme takes the least effort to implement and makes the least number of promises. Depending upon the low-level software structure used to handle network traffic, the degree of real-time character varies. The most common low level software structure found in such systems is that of *software interrupts*. In this design, it is possible to have unbounded priority inversions; for example, a deluge of incoming lower priority packets whose processing is not preemptible can completely stop the progress of a higher priority process. Another point to note is that this scheme provides no isolation from other non-network-bound applications.

2. **Shared communication server**: This approach adds a level of indirection to network device access, i.e., all the packets sent to and received from the network pass through a system-wide Communication Server (CS), which handles all protocol processing. This way, it would be possible to ensure fairness and resource allocation to network-bound processes[3]. However, it is imperative for the task scheduling mechanism to shield the CS from CPU-hogging by non-network-bound processes. The big disadvantage of this approach is the extra latency, which real-time systems should be wary of. Since the CS works at the application level, it does not have direct access to convenient in-kernel functions for efficient timer and buffer management; this makes implementation both complex and inefficient.

3. **Processing using prioritized threads**: Another interesting approach is to associate packets with priorities and arrange them to be *shepherded* by threads of matching priority. This assumes that the time needed to process packets is much more than the thread context-switching time. A good example of this approach is the ARTS [5] kernel. Prioritized threads, in conjunction with appropriate priority inheritance mechanisms, can be used to effectively deal with priority inversions. Note that this approach would make sense only in operating systems that already provide thread support. Since the *shepherd threads* normally operate on shared kernel data structures,

[3] It must be noted that for this approach to succeed, the kernel must have sufficient capability to transfer almost all protocol processing responsibility to the CS.

locking and synchronization mechanisms become necessary, making the implementation complicated. Moreover, appropriate task scheduling mechanisms would be needed to isolate the effect of other non-network-bound processes and threads.

4. **Application level protocol processing**: This approach is based on moving most of the protocol stack to the application layer as a linkable library. This would serve several purposes. It would make protocol processing a great deal more preemptible. It would reduce contention for shared data structures considerably. It would also avoid expensive system calls.

   On the other hand, this scheme leads to a significant amount of code duplication. As memories get bigger, this should be a lesser problem. In this category, two variants are Real-time thread-based and Real-time upcall-based designs. These variants are typified by the Real-Time Mach [6] and RTU [7] systems.

3.2 Protocol Processing in RT-Mach

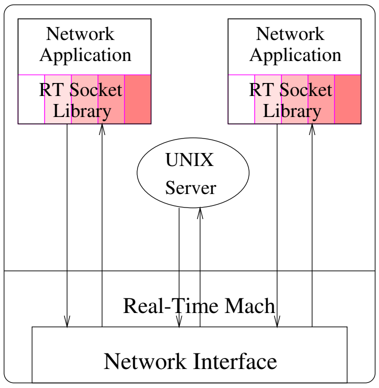

Figure 1: RT-Mach Protocol Structuring

Real-Time Mach is among the very few systems that support hard real-time protocol processing. It implements real-time thread-based application-level protocol processing. The applications are linked to the protocol library, which handles all data transfers. Connection setup and tear-down are managed in a trusted UNIX server. Fig. 1 illustrates the design of the RT-Mach protocol stack.

The application protocol library is composed of five threads: *application_thread* for all user send operations, *network_thread* for packet reception, *network_proxy_thread* for handling other UNIX mechanisms such as `select`, *timeout_thread* for handling timeouts, and *pc_sample_thread* for profiling. These threads correspond to the "compartments" shown in Fig. 1. The two CPU-scheduling policies that RT-Mach supports are rate-based priority and resource reservation (along with suitable admission control). In the first case, each thread is assigned a priority and in the second case, a processor reservation commensurate with its importance. The *application_thread* inherits its priority/reservation from its parent application; the *network_thread* is assigned a priority/reservation based on the expected burstiness and frequency of incoming packets, and the other threads are assigned appropriately small priorities/reservations. RT-Mach simplifies its task by harnessing the real-time thread facility already available to it. Efficiency is achieved by allowing the application to interact directly with the network interface.

The RT-Mach approach might lead to a larger number of other context switches because packet reception cannot be postponed for too long, and the only way it can be processed is to context-switch to the target application thread. Also, this approach would not be feasible in environments without real-time thread support. Since the threads in the application would need to interact asynchronously in the face of preemptions and context switches, locking of data structures would be unavoidable; this leads to run-time overhead and the ensuing reduction in utilization. It could be argued that this cannot be helped much if one desires hard real-time guarantees.

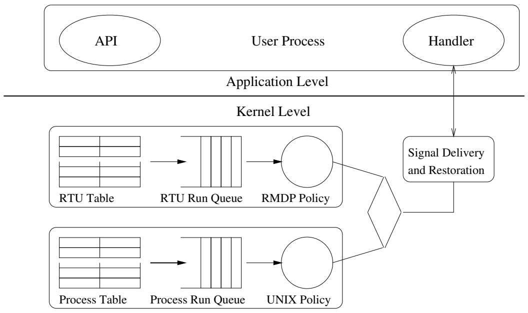

Figure 2: Real Time Upcall Mechanism

3.3 The RTU approach

The Real-Time Upcall (RTU) approach is designed for soft real-time multimedia systems. It is based on the reasonable assumption that protocol processing in multimedia systems is mostly periodic. This approach provides a simple upcall-based mechanism for an application to do real-time packet processing. It does not itself specify a way of structuring a real-time protocol stack. In [7], the authors argue that the real-time upcall mechanism is adequate to build any real-time protocol stack. In this sense, the RTU application-level protocol processing is a very general one.

In this solution, the application registers handlers with the kernel, which periodically upcalls these handlers depending upon the negotiated QoS[4]. It is in these handlers that the application implements code for sending and receiving packets. In fact, this approach quantifies the QoS in terms of the number of packets processed per upcall. The QoS negotiation guarantees that the user process can process B_p packets every time interval T and can process b_p of them without fear of interruption. The RTU mechanism is *not* preemptive. If the RTU of a process is currently in execution and another RTU of higher priority becomes callable, the current RTU could process b_p packets before noticing that it needs to yield the network resource. The scheduling policy used is a Rate Monotonic one; the higher the value of B_p/T, the higher the priority. This scheduling scheme, called the Rate-Monotonic with Delayed Preemption (RMDP), comes along with a simple admission control mechanism, which is beyond the scope of this survey.

[4]QoS: Quality of Service

Fig. 2 illustrates the RTU architecture. The RTU table is akin to the File Table in UNIX. The double arrow between "Signal Delivery & Restoration" and the "Handler" signifies the RTU call and yield mechanism. Note that this mechanism does not modify the underlying basic scheduling structure and thus provides flexibility and portability. The communication between the kernel and the RTU is accomplished through shared memory; checking for waiting, higher-importance RTUs and manipulating the Protocol Control Blocks (the connection-dependent data structures) are among the interactions accomplished through shared memory.

As a result of a non-preemptive design, several advantages are gained at the expense of hard real-time guarantees. Since an RTU processes at least b_p packets in each run, the number of context-switches is reduced. The authors of [7] found that this approach reduced context switching by almost 30%. The non-preemptive design simplifies design by obviating the need to lock data structures shared between RTUs. On the other hand, the RTU mechanism itself is prone to security breaches and resource hogging. An RTU could hold on stubbornly to the shared resource, and the non-preemptive mechanism would not be able to do much about it. The situation is ameliorated by two factors. First, user programs would not directly access the RTU facility but go through a protocol stack that uses the RTU facility. Second, the kernel can keep track of application program misdemeanor thanks to the B_p QoS parameter and take appropriate action.

4 Disk Scheduling for Real-Time File Systems

4.1 Introduction

Real-time databases and multimedia systems have brought home the need for efficient real-time disk scheduling algorithms. In what follows, we survey disk scheduling schemes from an abstract viewpoint but do not describe the actual use of disk scheduling in real-time systems. That examination could comprise a survey in itself.

The key issue in disk scheduling is to reduce the effect of physical movement overhead of the disk parts and in particular, the *seek delay*[5]. Research in the area of disk scheduling attempts to balance two conflicting requirements, namely, disk bandwidth utilization and request response time/fairness. *Real-time disk scheduling* leans more towards providing fast and fair response times; the challenge is to do so without significantly compromising utilization. Disk schedulers need to service requests as they come, keep track of the continuous movement of the disk arm and decide 'online' what is to be done next. Another issue that some disk schedulers address is that of disk bandwidth reservation. A disk bandwidth resource reservation scheme needs to incorporate actual disk delays into the admission control formula while considering that a single disk block transfer can not be preempted.

4.2 Traditional disk scheduling algorithms

The most well-known in this class are the SCAN algorithms, in which the disk arm movement occurs in *sweep*s[6], servicing all requests encountered. There are several variants of this algorithm which try to incorporate real-time capabilities. Of these, we shall discuss FD-SCAN and P-SCAN. The other traditional approaches consist of non-real-time variants of First Come First Served (FCFS), Shortest Seek Time First (SSTF)[7], and Earliest Deadline First (EDF) algorithms.

[5] *disk seek*: radial movement of the disk arm

[6] A *sweep* is one in which the direction of movement is not changed

[7] SSTF: where the request at the least seek distance is serviced first

4.3 Real-Time disk scheduling algorithms

We shall discuss five real-time disk scheduling algorithms. The crucial indices of performance are disk utilization, response time, and overhead involved in decision making.

- **Earliest Deadline First / Just In Time (EDF/JIT)**: The basic idea in this real-time EDF-variant is to make use of the slack times (laxities) in the EDF schedule to service other requests currently at short seek distances. This would use bandwidth more efficiently in addition to maintaining real-time response times.

  This strategy is used in RT-Mach [9], where this algorithm is coupled with an admission control policy; this successfully takes into account the non-preemtibility of a single disk block access. Support is provided to reserve a certain fraction of the disk bandwidth. This is by far the only system that provides hard real-time guarantees on disk access without compromising disk bandwidth utilization. The "slack time" mentioned above translates to the overprovisioning done thereof. Authors of [9] found EDF/JIT to achieve disk bandwidths only 3 to 30% from optimal.

- **Priority SCAN (P-SCAN)**: P-SCAN is a variation of SCAN having multiple priority levels. At any point in time, this algorithm follows a SCAN algorithm in one priority level; when it finds that there is a request in a higher priority level, it switches to that higher level and executes the SCAN algorithm in the direction determined by the request with the shortest seek distance. This algorithm has a small think time and makes good use of the disk bandwidth. Nevertheless, it can lead to unbounded starvation.

- **Feasible Deadline SCAN (FD-SCAN)**: FD-SCAN first finds all *feasible*[8] deadline requests; then, it scans towards the one with the earliest deadline and services all requests it finds on the way. This algorithm has been found to do well [10] in terms of response time and disk bandwidth utilization but has high overhead. Potentially, it will have to reevaluate the feasibility criterion at the end of each block access.

The SSEDO and SSEDV algorithms [8], which are discussed below, use a windowed approach to scheduling. Both order requests based on their deadlines and consider m of the closest deadlines, m being a tunable parameter. A lower value of m makes this algorithm more EDF-like while a higher value of m makes it more SSTF-like. It is possible to tune these two algorithms to balance response time and bandwidth utilization characteristics. These algorithms cannot avoid priority inversions and missed deadlines without reverting to simple EDF; thus, their use is restricted mostly to soft real-time systems.

- **Shortest Seek and Earliest Deadline by Ordering (SSEDO)**: This algorithm is concerned only about the ordering of deadlines and not how much time separates the requests. It assigns weights w_i to requests, and these weights grow geometrically with the ordering of deadlines; specifically, $w_i = \beta^i$, where $\beta \geq 1$ is a tunable parameter. If the current seek distance corresponding to weight w_i is d_i, it chooses the one with the smallest $w_i.d_i$ product. By doing so, a request at a very short seek distance can be given preference over one that has an earlier deadline (lesser weight). Note that this algorithm turns into a windowed SSTF variant if $\beta = 1$.

- **Shortest Seek and Earliest Deadline by Value (SSEDV)**: The SSEDO algorithm does not consider actual laxities, and this can lead to unnecessarily missed deadlines. For example, it could happen that request r_1 has a very close deadline but large seek distance d_1, and request r_2 has a very short seek distance d_2 but a comfortable deadline; SSEDO could prefer r_2 over r_1. Thus, r_1 might miss its deadline. However, if r_1 had been allowed to go first, then both r_1 and r_2 could have processed within their deadlines. To avoid such problems, the SSEDV includes the *remaining lifetime* l_i (time until deadline) of request i into the decision making. The new decision is based on the minimum value of $(\alpha d_i + (1-\alpha) l_i)$, where α is a tunable parameter like β above[9]. SSEDV offers a better tunability than SSEDO.

  Both SSEDO and SSEDV can make scheduling decisions in constant time and spend a time logarithmic in the number of requests to handle new requests. The claim to fame, however, is the tunability they provide. Authors of [8] report that m values of 3 an 4 give the lowest loss probabilities. However, the bandwidth utilization aspects of these algorithms have not been evaluated; such an evaluation would give us a better understanding. Also, we could not find any system that uses these algorithms in practice; such examples would have improved our insight in this matter.

5 Conclusions and Future Directions

In this survey, we have examined three important components of a real-time system, namely, task scheduling, protocol processing and disk scheduling. We have summarized and wherever possible, classified the various solutions that have been developed in each component. We have also evaluated their advantages and disadvantages. This survey could thus be used as a starting point for a

[8] A deadline is *feasible* if it can be met; this takes all overhead into account

[9] Note that $\alpha = 1$ turns SSEDV into a windowed SSTF variant, whereas $\alpha = 0$ turns it into a windowed EDF variant

more extensive study of the field.

The field of Real-time Operating Systems is an active field of research and abounds in unresolved issues. There is a strong motivation to add fault tolerance to distributed real-time systems, and this new topic needs greater research attention. Scheduling schemes need to be developed in which the leftover and over-provisioned CPU, network and disk bandwidth resources can be quickly reclaimed. With booming multimedia systems, there is a need to develop support for scheduling algorithms that support a smooth trade-off between solution quality and timeliness. Until now, solutions have considered resource scheduling from a rather abstract standpoint, without considering actual overhead. The focus of research must shift towards assessing and incorporating this overhead into scheduling schemes. There is a growing need to build complete systems which provide real-time guarantees in various subsystems; in this regard, there is need to study of the effect of real-time guarantees in one subsystem on another.

References

[1] Stewart, D. B., and Khosla, P. K. "Real-Time Scheduling of Dynamically Reconfigurable Systems", *Proc. of the IEEE International Conference on Systems Engineering*, Dayton Ohio, Aug. 1991, pp. 139-142

[2] Stankovic, J. A, and Ramamritham, K. "The Spring Kernel: A New Paradigm for Real-Time Operating Systems", *IEEE Software*, May 1991, 8(3):62-72.

[3] Ramamritham, K., and Stankovic, J, A. "Scheduling Algorithms and Operating Systems Support for Real-Time Systems", *Proc. of the IEEE*, Jan. 1994, pp. 55-67.

[4] Ramamritham, K., and Stankovic, J. A. "Distributed Scheduling of Tasks with Deadlines and Resource Requirements", *IEEE Transactions on Computers*, Aug. 1989, pp. 1110-1123.

[5] Tokuda, H., and Mercer, C. W. "ARTS: A Distributed Real-Time Kernel", *ACM Operating Systems Review 23, 3* July. 1989, pp. 29-53.

[6] Lee, C., Yoshida, K., Mercer, C., and Rajkumar, R. "Predictable Communication Protocol Processing in Real-Time Mach", *Proc. of the Real-time Technology and Applications Symposium*, June 1996.

[7] Gopalakrishnan, K., and Parulkar, G. "Real-time Upcalls: A Mechanism to Provide Real-Time Processing Guarantees", *Technical Report WUCS-95-06, Washington University*, Sept. 1995.

[8] Chen, S., Stankovic, J. A., Kurose, J. F., and Towsley, D. "Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems", *The Real-Time Systems Journal 3*, 1991, pp. 307-336.

[9] Molano, A., Juvva, K., and Rajkumar, R., "Real-Time Filesystems: Guaranteeing Timing Constraints for Disk Accesses in RT-Mach", *In Proceedings of the IEEE Real-Time Systems Symposium*, Dec. 1997.

[10] Abbott, R., Garcia-Molina, H., "Scheduling Real-Time Transactions with Disk Resident Data X Server", *Technical Report CS-TR-207-89, Department of Computer Science, Princeton University*, Feb. 1989.