Cooperative Scheduling of Multiple Resources

Saowanee Saewong and Ragunathan (Raj) Rajkumar

Real-time and Multimedia Systems Laboratory, Carnegie Mellon University, Pittsburgh, USA
{*ssaewong, raj*}*@ece.cmu.edu*

Abstract

Obtaining simultaneous and timely access to multiple resources is known to be an NP-complete problem [10]. Complete resource decoupling is, therefore, often used for managing end-to-end delays in distributed real-time systems where each processor is scheduled independent of the others. This decoupling approach unfortunately fails when multiple resources must be managed within *a single node. Resources such as disk bandwidth and network bandwidth are available on a single node but must be managed by their host processor by means of device drivers, filesystem or protocol services. The host processor acting as a* controlling resource, *therefore, must play multiple roles. One, it is used by applications on that node. Two, it is used to control and manage other (time-shared)* controlled resources *including disk bandwidth and network bandwidth. These two roles, unfortunately, can often be at odds with one another.*

In this paper, we investigate the problem of co-scheduling controlling and controlled resources. We propose the use of a Cooperative Scheduling Server (`CSS`*), which is a dedicated server that manages one specific controlled resource (like disk bandwidth, network bandwidth, inter-process communication, etc.) while using a controlling resource (like the processor). Two core ideas underlie our approach. First, a single (aperiodic) server is created on a controlling resource (such as a CPU) to handle* all *local requests for a controlled resource (such as disk bandwidth). This implies that conjunctive admission control must be carried out on both the controlling and controlled resources. Secondly, timing constraints at the application level are partitioned into multiple stages, each of which will be guaranteed to complete on a particular resource.* `RTFS` *is a real-time filesystem [2] that provides disk bandwidth guarantees under light CPU loads. With a cooperative scheduling server (*`FS-CSS`*) for this disk-based filesystem, disk bandwidth guarantees can be obtained under* both *heavy CPU and disk workloads. We describe the design and implementation of* `FS-CSS` *for providing disk bandwidth guarantees. We conclude with a detailed performance evaluation of* `FS-CSS`.

1. Introduction

The essential goal of OS resource management for real-time and multimedia system is to provide timely, guaranteed and protected access to system resources [24]. However, schedulers in real-time systems normally focus on a single resource at any given time. For example, a significant amount of research has focused on processor scheduling alone and similarly for network scheduling alone. Disk bandwidth scheduling has been studied to a smaller extent. The problem that has not received sufficient attention is the one of using these different resources, such as the processor, network and disk bandwidth, *simultaneously* within a single node. Sophisticated multimedia applications including video-on-demand and live video-conferencing may access high volume data from a disk, process the data, and transmit it across the network. All these stages must complete by a deadline. This resource management problem is complicated for two reasons. First, it is known that obtaining simultaneous and timely access to multiple resources is known to be an NP-complete problem [10]. Two, each of these resources may be scheduled by a different scheduling policy so that resulting scheduling mismatches have to be resolved.

The most straightforward way of approaching the multiple resource co-scheduling problem is to decouple the use of different resources [24]. This resource decoupling solution is effective only if resources are independent of one another. For example, consider an application which is computation-intensive. Even though its processor guarantee is granted by the processor admission control, the application could miss its deadline because of page faults. The memory resource guarantee can be managed independently from the processor by binding the application code and data into main memory during initialization [28]. Therefore, the application can just request the processor guarantee and the memory guarantee separately. Unfortunately, in practice, some combinations of resources cannot be totally scheduled independently such as the disk bandwidth-processor pair and the network bandwidth-processor pair. Many resource couplings are in fact tied up with the processor for the simple reason that it serves two purposes: for computation by applications and for control of other I/O peripherals.

We now provide a brief overview of RT-Mach and its Real-Time Filesystem (`RTFS`) to provide context and background for the CSS design and implementation.

1.1. A Brief Overview of RT-Mach

RT-Mach [24] is a microkernel operating system which employs separately scheduled servers to provide various system services [7]. It supports an integrated framework that encompasses task scheduling, virtual memory management, synchronization primitives, real-time inter-process communications, real-time disk scheduling and real-time network protocol processing. More recently, RT-Mach has adopted a resource-centric approach based on resource reservations and strict enforcement to provide timely and guaranteed access to resources. This core subsystem that delivers timely access to resources is called a *resource kernel* [24]. An application using the "resource kernel" can specify multiple resource reservations simultaneously and independent of the scheduling policy in the kernel (rate-monotonic policy, deadline-monotonic policy or earliest deadline first policy). The reservation specification uses the $\{C, T, D, S, L\}$ model for the reservation of resource C units of time every recurring time interval T before a deadline D. S and L are the starting time and the life time of resource allocation respectively. In addition, RT-Mach tracks an implicit resource parameter, the blocking time B. This blocking time represents the maximum (desirably bounded) time that a reservation instance must wait for lower priority reservations while executing. This implicit parameter B is introduced with the priority inheritance algorithm[22, 26] supported in the RT-Mach kernel to limit the priority inversion problem. RT-Mach uses the resource decoupling technique to provide the admission control mechanism of each resource separately before granting a resource reservation to the application. It also enforces the usage of resources such that the abuse of resources (intended or not) by one application does not hurt other guaranteed applications. However, as we stated earlier, the resource decoupling technique is feasible only if resources can be independently managed.

1.2. An Overview of Real-Time Filesystems on RT-Mach

Real-Time Filesystem Server (`RTFS`) [2] is a real-time file server running on top of the RT-Mach resource kernel. It has an admission control policy for disk scheduling using the same concept of rate-monotonic analysis for processor scheduling proposed by Liu and Layland [18]. The application can again request a disk bandwidth guarantee using the $\{C, T, D\}$ parameters of the resource kernel. C stands for the number of bytes the application wants to read in a duration of period T before a deadline D. `RTFS` has multiple worker threads which receive and process file access requests from real-time clients. Each worker thread is responsible for storing an incoming request into a common disk request queue. The disk request queue is divided into two different queues: one for reserved requests and the other for unreserved requests and depleted requests[1]. The disk scheduler dispatches reserved requests first using an earliest deadline first (EDF) scheduling policy and dispatches unreserved requests when no reserved request is left in the queue. `RTFS` provides disk access guarantees under disk access competition workloads but only with a light CPU workload.

1.3. Comparison with Related Work

In this paper, we implement a filesystem cooperative scheduling server (`FS-CSS`) by modifying the real-time filesystem server (`RTFS`) [2]. `RTFS` focuses on how the disk scheduler handles the disk request from applications to guarantee time constraints under light CPU workloads. However, during CPU overload conditions, `RTFS` can become unstable because the disk management activities are not properly scheduled by the kernel.

The co-scheduling problem between the processor and the disk is discussed in the continuous media file system (CMFS) in [8]. CMFS is designed and implemented to support real-time storage and retrieval of continuous media data. It uses a buffering technique to reduce the level of synchronization between the application and the disk server. The response time of disk accesses is improved by a disk layout adjustment in order to support real-time behavior. CMFS addresses throughput issues but without hard guarantees.

Resource decoupling and a system (or intermediate) reservation technique is proposed in [24]. This approach tries to decouple resources into separately scheduled entities. The "system reservation" is created for any device scheduler that needs the synchronization between its controlled resource management and the controlling processor cycles. With the system reservation, the disk scheduler gets a guarantee that the required disk service will be dispatched immediately after the disk access completes. Even though this approach can provide disk access guarantees, the exact processor admission control policy is not clear. There is no mechanism to make sure that the system reservation is sufficiently sized for the disk scheduler, or to manage the system reservation among multiple resources such as network bandwidth and disk bandwidth.

Our work is also closely related to that of Jeffay et al.[11] and Lee et al.[3] in the scheduling of OS services. The former studied the problem of scheduling the communication protocol stack processing activities inside a monolithic operating system. A fixed slack-sharing scheme was implemented in the kernel to ensure that guaranteed network bandwidth is available to applications. [3] considered the problem of scheduling protocol stack processing activities in a micro-kernel environment. Their solution is to have a very efficient packet filter that routes packets to clients who process the communication protocol stack within their own address spaces using user-level threads. With this scheme, the use of system resources for system activities is "charged" to user threads. Hence, it forces applications to request and reserve sufficient processing resources to satisfy even their "system activities". Our `CSS` approach can also be viewed as a generic solution for scheduling OS activities, and it can be applied in a micro-kernel environment or a monolithic kernel. The `CSS` implementation discussed in this paper is built in a micro-kernel environment but the server concept is also applicable within a monolithic kernel.

We now summarize and compare our approach with the above approaches. Our Cooperative Scheduling strategy uses a dedicated server to separate the resource management between the disk (or the network) and the processor. Our Filesystem-Cooperative Scheduling Server (`FS-CSS`) is an enhanced version of `RTFS` to provide a guarantee of disk access under heavy CPU

[1] Depleted requests are reserved requests that have already consumed all of their disk bandwidth reservations.

and disk workloads. We use the same scheme in the disk scheduler as the RTFS but add the cooperative scheduling module to make sure that the disk scheduler (in say a device driver) gets necessary cycles on the CPU in timely fashion. This is accomplished by having the processor admission control module take into account the exact processor needs of the FS−CSS. With this approach, both network bandwidth and disk bandwidth guarantees can co-exist in the system since the processor scheduler can individually track how much of the processor cycles the FS−CSS and NT−CSS (NeTwork-Cooperative Scheduling Server) need and account for those into the admission control. In addition, a conjunctive admission control used inside our CSS module considers both the waiting time for the disk head to complete disk access and the waiting time for the service to be dispatched to make sure both components are coordinated to meet the deadline of the application. This enables the FS−CSS to provide hard real-time service guarantees under heavy CPU *and* disk workloads.

1.4. Organization of the Paper

The rest of this paper is organized as follows. In Section 2, we present the requirements of a good multi-resource co-scheduling strategy. We describe our use of the Cooperative Scheduling Server (CSS), a dedicated server to manage each resource (disk bandwidth, network bandwidth, memory, etc.) in conjunction with a processor resource. In Section 3, we detail our implementation of the CSS approach for Disk bandwidth management on the RT-Mach microkernel-based system running a real-time server (a simple OS personality). Section 4 focuses on the performance evaluation of this approach. Finally, we present our concluding remarks outlining our research contributions and future work in Section 5.

2. Design Issues

In order to guarantee all controlled resource accesses against deadline misses, a co-scheduling strategy needs to make sure that the controlled resource (disk bandwidth or network bandwidth server) gets the proper sharing of the controlling resource, namely processor cycles, on a timely basis. This section discusses important design issues for such a co-scheduling strategy. Then we propose the *Cooperative Scheduling Server* (CSS) concept to synchronize the controlled resource scheduler and the controlling resource scheduler and provide a conjunctive admission control for both resources.

2.1. Important Issues for Co-Scheduling Design

We now list some important considerations that influence the design of a co-scheduling strategy.

Scheduling Mismatch due to heterogeneity of resource scheduling policies: The lack of explicit co-scheduling among resources can lead to a scheduling mismatch. For instance, in the case of disk bandwidth guarantees, the controlled resource is the disk bandwidth. Its controlling resource is the processor. The processor scheduler can assign the CPU thread priority according to the rate-monotonic scheduling policy. The disk scheduler can assign the disk access priority according to the earliest deadline first policy. This priority conflict poses the following dilemma. A CPU service must execute to initiate and complete disk transfers. From the disk perspective, one must ensure that these disk-related activities are not unduly delayed by other higher priority activities on the CPU. From the processor perspective, native CPU applications must not miss their timing constraints due to disk-related activities on the CPU.

Conjunctive Admission Control: The admission control of each controlled resource has to take not only its own resource access into account but also the availability of the controlling resource. Hence, to guarantee real-time service, the admission control of co-scheduling strategy needs to account for both the response time for the disk driver to access data from the disk *and* the response time of the processor scheduler to dispatch the driver process.

Resource Synchronization: Disk access commands, once issued to the disk, can proceed in parallel with processor computations on behalf of applications. Good synchronization between the disk and the processor will allow both resources to make progress in parallel as much as possible.

Efficient Resource Utilization: The main goal of real-time scheduling is to achieve high utilization and still guarantee deadlines for applications. Therefore, in addition to guaranteeing the deadline of multiple resource accesses, the system should provide acceptably high overall system utilization of all system resources.

2.2. The Cooperative Scheduling Server Concept

The Cooperative Scheduling Server (CSS) is a dedicated server for the management of controlled resources. In this paper, we focus on the co-scheduling between the disk and the processor but the same approach can be adapted to network bandwidth and the processor as well. We now present a high-level overview of our CSS approach.

A dedicated server, CSS, is created on a controlling resource to be responsible for *all* accesses to one particular controlled resource. Suppose that the CPU is the controlling resource, and disk bandwidth is the controlled resource. This "filesystem CSS", called the FS−CSS, reserves a sufficient amount of capacity on the CPU as needed to fulfil the obligations it makes for accessing disk bandwidth. When an application requests guaranteed and timely access to disk bandwidth, an explicit demand on the disk bandwidth and an implicit demand on the CPU are imposed. FS−CSS, therefore, performs admission control on both the CPU and the disk bandwidth to ensure that both demands can be satisfied. We refer to this as *conjunctive admission control*. From the CSS point of view, it needs to correctly evaluate the imposed demands on the CPU and the disk for an incoming request. In addition, since it must satisfy the timing needs of multiple applications, its own parameters (period and aggregate capacity) must be determined.

The design of the CSS is tightly tied to the pattern of usage of the controlled resource (such as disk bandwidth). In general, an application needs to consume both CPU capacity and disk bandwidth within a certain deadline. Hence, an executing application may alternate between requests for disk accesses and CPU processing. To illustrate our concepts, without loss of generality, we adopt a simple but reasonable programming model where an application performs some CPU processing, initiates disk transfers and continues with CPU processing after the disk transfer is complete. We impose timing constraints (deadlines) on each of these three stages, the sum of which will equal the overall timing constraint of the application.

The assignment of deadlines to these "pipeline stages" is not unlike the "end-to-end scheduling" problem where end-to-end timing constraints can be satisfied by partitioning deadlines across stages of the end-to-end path. The primary differences in our case are two-fold. One, the timing constraints on each of our stages are much tighter. Two, the disk transfers corresponding to multiple applications are handled by one CSS server, an aspect that does not have a direct counterpart in end-to-end scheduling. The implication of this latter observation is that the CSS server parameters have to be carefully chosen and analyzed. We shall discuss this issue in more detail in Section 3.3 and Section 4.3.

2.3. The Detailed Design of CSS

A dedicated Cooperative Scheduling Server (CSS) provides applications timing guarantee using $\{C, T, D\}$ parameters where C stands for the amount of resource the application needs in a recurring period of T before a deadline D. The unit of C depends on the type of each resource. For example, C is the number of bytes (and hence the disk bandwidth consumption time) in the case of disk bandwidth, and the processor computation time needed in the case of the processor.

In co-scheduling the disk and the CPU, when the disk head has completed its reading, the disk scheduler may not get a bounded response time from the processor scheduler. This can cause an application to miss its deadline. Our Cooperative Scheduling Server will request a guarantee from the processor scheduler to make sure that it is properly scheduled when the disk head is idle and ready to read more data. The CSS concept allows the server to request processor sharing with a small server period (deadline) to get high priority in rate-monotonic scheduling (deadline-monotonic scheduling) by the processor scheduler. The CSS computes how much of the processor is needed in each server period to fulfill *all* guaranteed services it grants to applications. It maps all of its processor requirements to a $\{C_s, T_s, D_s\}$ model. C_s stands for the processor computation needed in each server period T_s before the server deadline D_s. Selecting an optimal T_s is quite difficult. A small T_s is desirable for a wide range of timing guarantees because the CSS can guarantee only those requests that have periods and deadlines greater than the server period T_s. However, a small T_s increases overhead and can significantly reduce the granularity of the processor share that the CSS can request from the processor scheduler.

Even though the server can compute the amount of the processor needed to service applications, the time when the CPU is available for the disk scheduler is critical since the disk scheduler cannot use the available CPU when the disk drive is still busy. The CSS therefore considers the response time of the disk driver to read the data from the disk as "blocking time". The processor scheduler takes this blocking time as an implicit parameter into the admission control test to ignore all available CPU when the disk is busy. This guarantees that the the disk scheduler has sufficient CPU available to run when the disk is not busy.

When the CSS receives a grant of its processor request, it receives a priority in the processor scheduler based on the server period (deadline). We refer to this priority as CSS priority. This CSS priority will be used for all application threads that request disk guarantees during the disk access. Therefore, whatever the application priority, the disk scheduler would use the CSS priority to get appropriately high priority scheduling on the processor when it needs disk services. CSS thus eliminates any priority mismatches between application and disk access priorities. In other words, the processor scheduler views all applications that need the disk access as having the same priority, and has full control over all application threads that access the disk.

To use the CSS model, we have to make sure that the CPU consumption of the CSS is periodic and satisfied with the $\{C_s, T_s, D_s\}$ requirement given to the processor scheduler. The CSS uses a token control algorithm to confine its processor consumption to be periodic with period T_s and not larger than the amount of the processor it has reserved. We will show in Section 3 how to compute the amount of processor needed by the CSS.

One last consideration needed is regarding the bounded response time for the communication between an application and the CSS. An application request must not be received at the CSS too late. Similarly, the time for the CSS to send the disk data back to the application must also be bounded. We refer to the time of communication from the application to the server as the *invocation time*, and that from the server to the application as the *return time*. The deadline for both invocation and return times must necessarily be correlated to the deadline of the server to complete the service. For example, if we let the deadline of the invocation time and the return time to be small, the server will have more slack time to complete the service and vice versa.

The execution pattern we consider is as shown in Figure 1. In the pattern shown, C_A represents not only the invocation time but also any computation needed before accessing the disk at the application level. Similarly, C_B models not only the return time but also any computation needed after accessing the disk at the application level. C_R is the time needed by the CSS to access disk data on behalf of the application. If S is the overall slack time available then $S = D - (C_A + C_B + C_R)$. There are many possible ways to share slack S among S_A, S_B and S_R where S_A, S_B and S_R are the slack times for processor computation of C_A, C_B and C_R respectively. Fixed slack-sharing among these three stages is a simple scheme to share the resource. We will discuss slack-sharing in detail again in Section 3.

In summary, our CSS can not only provide hard real-time guarantees but also allow the system to obtain high resource utilization. It must be noted that the Cooperative Scheduling module

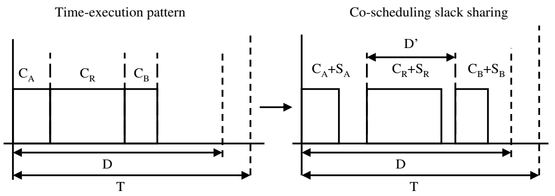


Figure 1. The slack-sharing method for general processor consumption pattern of threads

also allows a controlled resource scheduler to use any scheduling policy. Therefore, if a new scheduling policy that can guarantee real-time service and improve the disk utilization becomes available, the `CSS` allows the update of the scheduling policy and thereby benefit from the increased disk performance.

2.4. The Programming Model with `CSS`

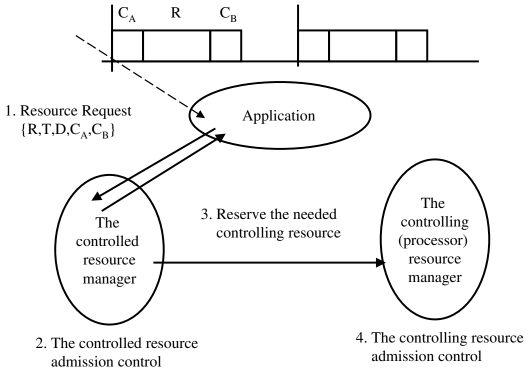


Figure 2. The Cooperative Scheduling Server (`CSS`) architecture

We now describe the programming model to be used by applications with the Cooperative Scheduling Server. The `CSS` uses cooperative scheduling to synchronize the controlled and controlling schedulers which may have different scheduling policies. It applies a conjunctive admission control to make sure that both the controlled resource (the disk) and the controlling resource (the processor) needed are available. The `CSS` allows an application to specify its resource requirement in an $\{C, T, D, C_A, C_B\}$ format. C stands for the amount of resource the application needs in every period T before a deadline D. C_A and C_B are the processor computation time needed at the application level before and after accessing the controlled resource R according to the execution pattern shown in Figure 1.

The architecture of `CSS` is shown in Figure 2. First the `CSS` computes C_R and then shares slack time among the CPU needed for C_A, C_B, and C_R. `CSS` does the admission control to determine if the access of the controlled resource can be completed before a deadline D' where D' is the adjusted deadline of the controlled scheduler to complete the application request after sharing slack to both C_A and C_B (see Figure 1). The `CSS` computes the processor time-share needed in order to satisfy timing constraints of all guaranteed services and maps the requirement to a $\{C_s, T_s, D_s\}$ format where C_s is the processor computation time needed every server period T_s with server deadline D_s.

The `CSS` is also responsible for requesting processor guarantees for C_A and C_B. After successfully requesting the guarantee of its own resource and all the processing needs of C_A, C_B and itself, the `CSS` returns a grant to the application. If admission control fails, intelligent adaptation of the application-level performance and hence its resource needs may become necessary.[2]

3. A `CSS` for Disk Bandwidth Guarantees

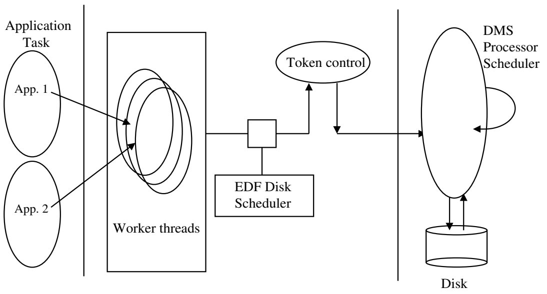


Figure 3. The `CSS` design for disk bandwidth management

This section shows how to adapt the `CSS` approach to obtain guaranteed and timely access to disk bandwidth. Figure 3 illustrates the architecture of `FS-CSS`, a modified real-time filesystem to achieve effective disk bandwidth and processor co-scheduling. In our system, the disk scheduler uses the EDF scheduling policy and the processor scheduler uses the deadline-monotonic scheduling policy. The main goal of `FS-CSS` is to give coordinated service guarantees for both disk bandwidth and processor usage. Since most of the processor reservations created by `FS-CSS` have deadlines less than their periods, to get higher system utilization, the kernel processor scheduler inside RT-Mach is set to use deadline-monotonic scheduling and the processor *reservation* scheduling policy to have an enforcement of processor usage [24].

We now present a brief overview of `FS-CSS`. The application requests a disk bandwidth guarantee by specifying C as the number (bytes) of data it wants to access every recurring period T before a deadline D together with the CPU computation needed before and after accessing the disk. The `FS-CSS` computes C_R, the slack available S and shares S among C_A, C_B and C_R. The admission control in the disk scheduler ensures that the raw speed and bandwidth of the disk drive are sufficient for all guaranteed disk access services. Three processor reservations, BF_{RES}, AF_{RES} and SER_{RES}, are created to make sure that there is enough CPU sharing for the invocation C_A of the disk request, the return time C_B of the disk request and the service time C_R of the disk scheduler respectively.

In the following subsection, we detail the system architecture inside `FS-CSS`, the slack-sharing method, the `FS-CSS` capacity

[2] Q-RAM (QoS-based Resource Allocation Model) [23] is an analytical model that allows applications to operate at different levels of quality of service (QoS) based on the resources available, and can be used here.

computation[3], the admission control for the controlled resource (disk) and the admission control of the controlling resource (the processor). The parameters of all reservations, BF_{RES}, AF_{RES} and SER_{RES}, are also derived.

3.1. Disk BW Reservation Specification Notation

Let the set of n reservations requiring disk bandwidth guarantees be denoted as $\tau_1, \tau_2, \ldots, \tau_n$. Each reservation τ_i needs to read C_i bytes of data every T_i units of time. In addition, the data C_i bytes must be available at or before deadline D_i in each periodic interval separated by T_i. Let τ_i have higher priority than $\tau_{i+1}, \ldots, \tau_n$. This means that $D_1 \leq D_2 \leq \ldots \leq D_n$ in the case of the deadline-monotonic scheduling policy. The reservation τ_i is also needed to provide C_{A_i} and C_{B_i} which are the processor cycles needed at the application level before and after disk access. The disk bandwidth request is therefore in the form of $\{C_i, T_i, D_i, C_{A_i}, C_{B_i}\}$.

3.2. Disk Scheduler with Periodic Token Control

A real-time file server (RTFS) running on top of the RT-Mach microkernel manages the reserved real-time file system. RTFS has multiple worker threads, which receive and process file system requests from real-time clients. A worker thread is assigned to an incoming request until the request is completed. Each disk request is fragmented into blocks (8KB per block in the current implementation). When a worker thread gets a request, it stores the request into a common request queue and blocks for a signal (condition variable shown in Figure 3) from the scheduler. The disk scheduler signals the worker thread, which is responsible for the next dispatched block access using the earliest deadline scheduling policy. A token control scheme is added to confine the processor computation of RTFS to be periodic and satisfy the $\{C_s, T_s, D_s\}$ parameter. A certain amount of token is replenished in every server period T_s. The disk scheduler will allow the access of the next block if and only if there is a token available. Figure 3 shows the architecture of RTFS after modification.

3.3. Fixed Slack-Sharing in FS-CSS

Before sharing slack between the processor scheduler and the disk scheduler, the FS-CSS has to compute the time needed to get a disk access C_i bytes under no CPU competition to find the available slack for the request. Let

- C_{disk_i} be the (least) time needed to get access to C_i bytes without CPU competition.
- T_{seek} and T_{rot} be the maximum seek time and rotational time of the disk head respectively.
- T_{fs} and T_{io} be the disk scheduling overhead and the I/O time for the disk to read (or write) one file system block respectively.
- S_i be the slack available for reservation τ_i.
- S_{A_i} and S_{B_i} be the slack time for C_{A_i} and C_{B_i} respectively.
- S_{disk_i} be the slack time for the disk bandwidth server.
- BL_i be the number of blocks reserved by τ_i.
- K be the block size (bytes) in FS-CSS.
- T_{inv} and T_{ret} be the overhead to invoke the request and return the data between the application and the FS-CSS respectively.

Then, we have

$$\begin{aligned} BL_i &= \lceil \frac{C_i}{K} \rceil \\ C_{disk_i} &= 2T_{seek} + 2T_{rot} + (T_{fs} + T_{io}) * BL_i \\ S_i &= D_i - (C_{A_i} + C_{B_i} + T_{inv} + T_{ret} + C_{disk_i}) \end{aligned}$$

In the implementation of FS-CSS, fixed slack-sharing is used due to the simplicity of the algorithm. Let D_i' be the deadline of the disk server to service the disk access of reservation τ_i after slack sharing then $D_i' = D_i - C_{A_i} - C_{B_i} - S_{A_i} - S_{B_i}$.

From Theorem 1 to be presented in the next section, the best choice of D_i' is to make it an integral multiple of the server period T_s. Let X be the (approximate) proportion of available slack assigned to the disk scheduler. Then,

$$S_{disk_i} = \lfloor \frac{C_{disk_i} + (X * S_i)}{T_s} \rfloor * T_s - C_{disk_i}$$

Of the remaining slack, in order to balance the processor utilization needed between the pre-processing activity and the post-processing activity stages, we set the slack sharing for the pre-processing activitiy S_{A_i} and the slack sharing for the post-processing activity S_{B_i} to be porportional to the needed processor computation time of both activities. Then

$$\begin{aligned} S_{A_i} + S_{B_i} &= S_i - S_{disk_i} \\ \frac{S_{A_i}}{S_{B_i}} &= \frac{C_{A_i} + T_{inv}}{C_{B_i} + T_{ret}} \end{aligned}$$

Picking the right value for X is not an easy matter. Different values for X have different effects on the admission control aspects of the processor scheduler. In our experiment, we picked $X = 75\%$. We tried to get X as high as possible to reduce the over-estimation of FS-CSS Capacity Computation which will be described later. However, a higher X causes shorter deadlines for C_{A_i} and C_{B_i} which can lead to rejects from the processor admission control policy.

3.4. FS-CSS Capacity Computation

Using a recurrence relation [12, 29], FS-CSS computes the number of data blocks the disk scheduler needs to access in every system period T_s to fulfil all guaranteed services. In this section, the computation is shown together with the proof.

THEOREM 1. For n disk bandwidth reservations $\tau_1, \tau_2, \ldots, \tau_n$, the number of blocks *BS* the disk scheduler has to read in every server period T_s to fulfill all threads' deadlines D_i' where D_i' is the deadline of an reservation τ_i from the disk server perspective is given by

$$BS = Max(BS_1, BS_2, \ldots, BS_n)$$

[3] This computation finds the number of data blocks the FS-CSS needs to read in every server period T_s to guarantee all FS-CSS services. Then it can computes how much of the processor cycles needed to provide disk service and the blocking time needed to wait the disk scheduler to access the data from the disk.

where BS_i is the number of blocks the scheduler has to read to guarantee the reservation τ_i

$$BS_i = \lceil \frac{\sum_{j=1}^{i} (BL_j * \lceil \frac{D'_i}{T_j} \rceil)}{D'_i} \rceil * T_s$$

PROOF. τ_i is schedulable if the number of blocks read by the server during its deadline D'_i is greater than the total number of blocks needed for all higher priority threads and itself. Considering the worst case, the number of cycles each higher priority thread exists in duration of D'_i is counted by using the ceiling function as follows:

$$\begin{aligned}
BS_i * \lfloor \frac{D'_i}{T_s} \rfloor &\geq BL_i + \sum_{j<i} BL_j * \lceil \frac{D'_i}{T_j} \rceil \\
BS_i &\geq \lceil \frac{(BL_i + \sum_{j<i} BL_j * \lceil \frac{D'_i}{T_j} \rceil}{D'_i} * T_s \rceil \\
BS_i &\geq \lceil \frac{(BL_i * \lceil \frac{D'_i}{T_i} \rceil + \sum_{j<i} BL_j * \lceil \frac{D'_i}{T_j} \rceil}{D'_i} * T_s \rceil \\
BS_i &\geq \lceil \frac{\sum_{j=1}^{i} BL_j * \lceil \frac{D'_i}{T_j} \rceil}{D'_i} * T_s \rceil
\end{aligned}$$

Note: From the floor function above, the shortest of the set of D'_i values that can serve the same BS_i is when D'_i is an integral multiple of T_s.

The ceiling function above is for adjusting BS_i to be an integral number of blocks needed to be read. To guarantee all reservations in the system, the number of blocks the server has to read per server period is

$$BS = max(BS_1, BS_2, \ldots, BS_n)$$

3.5. Conjunctive Admission Control

FS-CSS has two levels of admission control, one for disk bandwidth access guarantee and the other for processor guarantee. The admission control of disk access under the EDF scheduling policy for n disk bandwidth reservations under the assumption of no CPU competition is as follows:

$$\sum_{i=1}^{n} [\frac{T_{setup} + BL_i * (T_{fs} + T_{io})}{T_i} + BU_{max}] \leq 1$$

where T_{setup} is the file system overhead including the communication time between an application and CSS together with the disk head setup time:

$$T_{setup} = T_{inv} + 2T_{seek} + 2T_{rot} + T_{ret}$$

$$BU_{max} = max(\frac{B_1}{T_1}, \frac{B_2}{T_2}, \frac{B_3}{T_3}, \ldots, \frac{B_n}{T_n})$$

B_i is the priority inversion duration encountered by reservation τ_i given by

$$B_i = (T_{fs} + T_{io}) + T_i - D_i$$

After FS-CSS checks the admission control of the disk access, it has to make sure that the processor cycles needed for C_{A_i}, C_{B_i} and itself are available. All processor reservations are needed to use $\{C, T, D, S, L\}$ specification with implicit parameter B. C stands for the processor computation time needed in every period T before deadline D. S and L are the starting time and the life time of resource allocations respectively. B is the blocking time the application has to wait for any lower priority thread (or the disk head in the filesystem case). There are two separate processor reservations created for every new incoming disk reservation request, BF_{RES} and AF_{RES}. One global reservation created by the disk scheduler, SER_{RES}, is updated for every new incoming disk reservation request.

BF_{RES}**:** This is the reservation for the application to invoke the disk request to the CSS and the computation needed at the application level *before* disk access. This reservation has the same period as the application period. The computation time is reserved in the sum of C_A and the needed invocation time from the application to the server. The deadline of the reservation is computed by adding the computation time with the slack time got from slack-sharing. In the implementation, one assumption of having no blocking time during executing C_A is made. Therefore, the parameter $\{C, T, D, S, L\}$[4] and B for BF_{RES_i} are as follows:

$$C = C_{A_i} + T_{inv}, T = T_i, D = C_{A_i} + T_{inv} + S_{A_i}, B_i = 0$$

AF_{RES}**:** This is the reservation for the FS-CSS to return the disk request to the application and the computation needed at the application level *after* disk access. Similar to BF_{RES}, the parameter $\{C, T, D, S, L\}$ and implicit parameter B for AF_{RES_i} are as follows:

$$C = C_{B_i} + T_{ret}, T = T_i, D = C_{B_i} + T_{ret} + S_{B_i}, B_i = 0$$

SER_{RES}**:** This is the global processor reservation for the computation needed inside the disk scheduler. The disk scheduler has to reserve all processor computation needed for reading BS blocks of data every system period T_s. The computation time needed in each server period is equal to the disk scheduler overhead to read (or write) one file system block times BS. The I/O time of disk device to read (or write) one file block times BS is considered as blocking time. This is because, during this time, the disk scheduler has to wait for the device to finish reading (or writing) the data block before scheduling the next block access. The deadline of the reservation is equal to the server period. All parameters needed for SER_{RES} are as follows:

$$C = BS * T_{fs}, T = T_s, D = T_s, B = BS * T_{io}$$

The processor admission control will determine if the new reservation request of BF_{RES} and AF_{RES}, and the updated reservation request of SER_{RES} are feasible with the current processor utilization using admission control based on traditional deadline-monotonic scheduling.

[4] S and L can be set to be void without any effect on the reservation.

4. Performance Evaluation

In this section, we evaluate FS−CSS using a series of experiments to study the efficiency of the co-scheduling between disk bandwidth and processor for timeliness guarantees. We ran different disk bandwidth reservations under both heavy processor and disk competition workload to test the effectiveness of the conjunctive admission control. Finally, we measured the average processor usage in the server compared with the processor we have reserved.

Table 1. The thread parameters used in the Experiment of Section 4.

Th	No.	Reserve	C(*ms/KB*)	T (*ms*)	D (*ms*)
CPU_0	2	CPU	2	30	30
CPU_1	2	CPU	6	100	100
CPU_2	2	CPU	50	325	325
FS_1	1	Disk BW	18 KB	200	200
FS_2	1	No Reserve	80 KB	280	280
FS_3	1	Disk BW	100 KB	320	320
FS_4	1	Disk BW	270 KB	521	521

4.1. The Effectiveness of Disk Bandwidth Guarantees

A sample set of scenarios is described below. Qualitatively similar results were obtained with serveral other workloads as well. We tested RTFS using the FS−CSS model on a Pentium II 300 MHz workstation. The disk we used is a 6 GB drive with a maximum head seek latency of 24 *ms* and a maximum rotational latency of 14 *ms*.

In our experiments, we pick the period of the FS−CSS to be 50 *ms*. Larger the value of T_s, higher is the granularity of the FS−CSS processor capacity. However, the hard real-time deadlines supported by FS−CSS are in multiples of T_s. This results in an "internal fragmentation" loss equal to $\frac{T_s}{D'}$, where D' is the deadline of the application. Due to the relative slowness of the disks, we expect that applications requiring real-time disk accesses will have deadlines equal to or greater than 200 *ms*. We then accept a practical internal fragmentation loss of up to 25%, and set T_s = 50 *ms*. Naturally, T_s should be changed accordingly if either the shortest application deadline or the acceptable fragmentation loss changes.

We first measured the following parameters: $T_{inv} = 2.3ms$, $T_{ret} = 4.1ms$, $T_{fs} = 500\mu s$, $K = 8$KB and $T_{io} = 2.3ms$.

Four periodic disk-access threads are spawned in several configurations, with and without disk bandwidth reservations. In all experiments, we have a number of processor competing threads with processor reservations to consume any available processor cycles. We run the experiment in a window time-span of 100 seconds measuring the completion time compared with the deadline requested by applications. All the threads in these experiments are configured as shown in Table 1.

Note. The C_A and C_B parameters of all FS threads are set to be 1 *ms* in our experiment.

In the first phase of this experiment, all disk-access threads are spawned *without* disk bandwidth reservation. Figure 4 shows the completion times of all four disk-access threads compared with their deadlines. The result shows that all disk-access threads miss their deadlines and their completion times are very varied. This is because the disk scheduler dispatches unreserved disk accesses without considering any thread deadlines.[5] Another reason is that the processor competing threads grab the processor needed by the disk scheduler.

In the second phase of this experiment, FS_1, FS_3 and FS_4 request disk bandwidth reservations. Figure 5 shows the completion time of all four disk-access threads with their deadlines. The result shows all three threads with disk bandwidth reservations meet their deadlines. FS_2, which is a disk access thread *without* disk bandwidth reservation, misses deadlines 136 times out of 350 times. Compared with the variation of the completion time of the unreserved disk access thread, the completion times of the reserved disk access thread are much more stable. This is because after the disk scheduler signals a worker thread to read a data block from disk, it allows the worker thread to continuously read data unless there is a new thread with higher priority coming in. In other words, the disk scheduler is properly scheduled by the processor scheduler and hence gets a predictable response time.

In the third phase of this experiment, an aperiodic thread is spawned to continuously consume *all* available CPU without processor reservations. This thread consumes all available CPU. The goal of this experiment is to show the effect of unreserved processor thread on disk access threads. Figure 6 shows that the existence of an aperiodic CPU-consuming thread has an effect on only FS_2, the disk access thread *without* disk bandwidth reservation.

4.2. The Effectiveness of Processor Guarantees

We now re-emphasize the sustained effectiveness of the processor guarantees in the system while using the CSS model. In the previous experiment, we have run several threads with disk bandwidth guarantees simultanouesly with several threads with processor computation guarantees (CPU_0, CPU_1 and CPU_2 as listed in Table 1). As seen in Figures 4 through 6, the system can guarantee the deadlines of the disk bandwidth reserved threads. At the same time, all reserved-CPU threads *also* meet their deadlines in all experiments. The processor usage obtained by these three CPU threads in the worst-case configuration used in Figure 6 is plotted in Figure 7 and corresponds exactly to the requested reservations.[6] This reflects the effectiveness of both processor and disk bandwidth guarantees obtained with the CSS approach.

[5] The disk performance for unreserved service can be improved by using SCAN algorithm or other scheduling policies.

[6] However, we have found that when the applications run for a relatively long time, a very small number of deadlines eventually do get missed. We suspect that some assumptions are being occasionally violated. Two likely candidates are non-contiguous layout of disk blocks read by an application on the disk side, and non-consideration of system overhead on the processor reservation side.

Table 2. Cpu over-estimation in Periodic Conversion

Reserved	U_{res}	U_{used}	Over-estimation (percent)
BF_{RES}	0.0281	0.0276	1.78
AF_{RES}	0.0462	0.0460	0.43
SER_{RES}	0.0900	0.0250	72.22
Overall	0.1643	0.0986	39.99

4.3. Accuracy of the FS−CSS Processor Capacity

The FS−CSS reserves some capacity on the controlling CPU resource as computed in Section 3.4. We conducted an experiment to determine whether this capacity was being under-used. The same set of threads as in the previous experiment was re-run to measure the average processor usage of FS−CSS. Table 2 compares all processor reservations created with the average processor usage of FS−CSS. As can be seen, our capacity computation is rather high relative to what is actually being consumed by FS−CSS. This results in wasted processor allocation to FS−CSS.

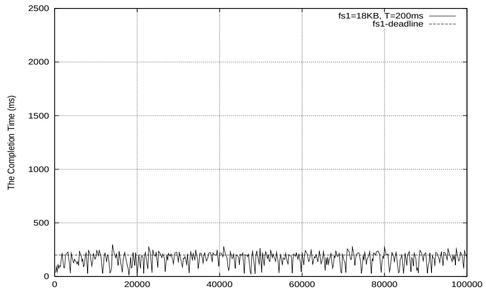
The source of this over-estimation lies in the conservative computation of the worst-case data block formula (using a ceiling function) given in Section 3.4. This processor over-estimation can be reduced in two ways.

1. Optimize the data block formula by reducing the effect of the ceiling function.

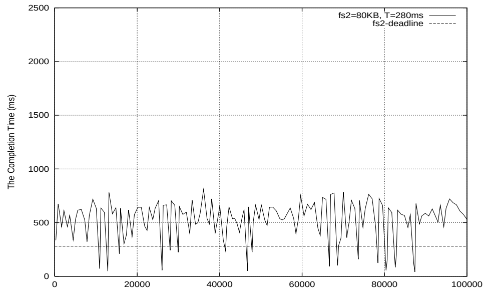
2. From the FS−CSS processor capacity formula, the higher the deadline D'_i of the reservation τ_i, the less is the number of average blocks the server has to read. Therefore, if the slack time to the disk scheduler increases, the number of blocks needed decreases. This reduces the processor reservation SER_{RES}. Consequently, the processor over-estimation is shrunk. Two approaches to increase D'_i are:

Increase the level of acceptable latency at the application level. The disk access (middle) stage of the programming model can be pipelined such that the application can use buffering to allow an additional period of disk access latency. This approach increases the slack available and, therefore, increases the slack time available to the disk scheduler. This is analogous to the *pipelining* stages in an end-to-end path of distributed real-time systems. The cost is increased data latency at the application level but with better schedulability.

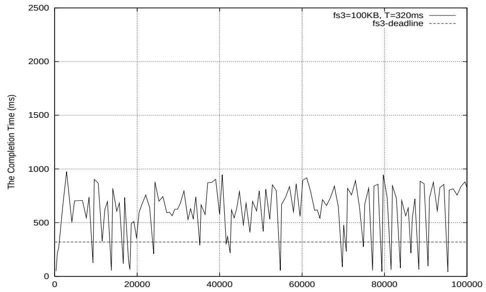
Increase the percent of slack-sharing for the disk scheduler server from BF_{RES} and AF_{RES}. In the experiment described earlier, fixed slack-sharing has been used. The percent of slack-sharing to the disk server can be increased to reduce the problem of processor over-estimation. However, when the percent of slack-sharing to the disk server is increased, the slack available to BF_{RES} and AF_{RES} are decreased. This leads to shorter deadlines for BF_{RES} and AF_{RES}. From the experiment, the deadlines of BF_{RES} and AF_{RES} after slack adjustment are approximately the same. A reject from the admission control policy for BF_{RES} and AF_{RES} requests is found if there are many BF_{RES} and AF_{RES} reservations in the system. This happens because there is no management of the relationship among the various BF_{RES}s and AF_{RES}s. A *biased slack-sharing* scheme



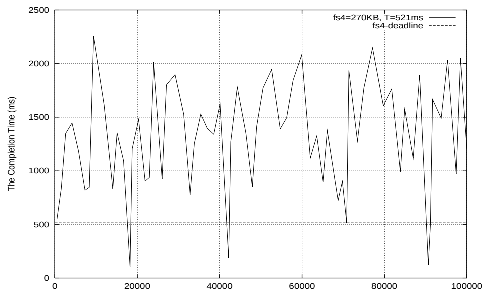
(a) FS_1-No reserve



(b) FS_2-No reserve

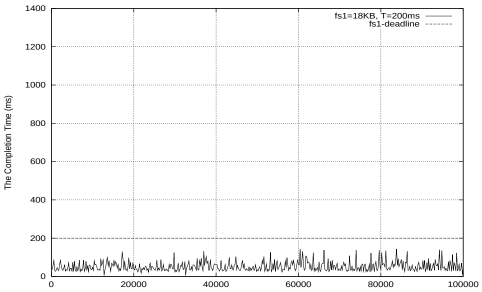


(c) FS_3-No reserve

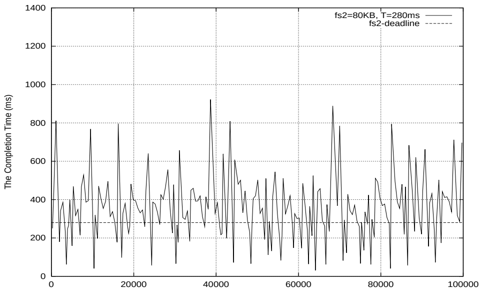


(d) FS_4-No reserve

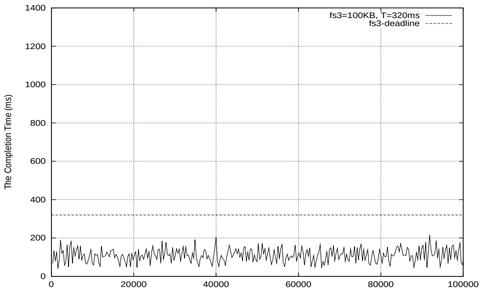
Figure 4. The completion of disk access threads without disk bandwidth reservation. At the same time, all the three reserved CPU tasks meet *all* their deadlines.



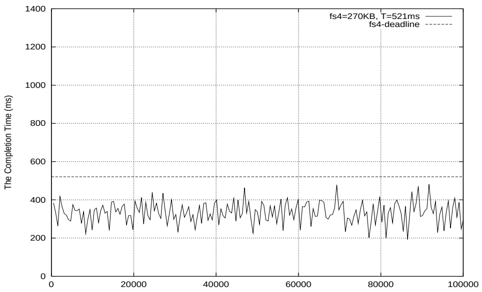
(a) FS_1-With reserve



(b) FS_2-No reserve

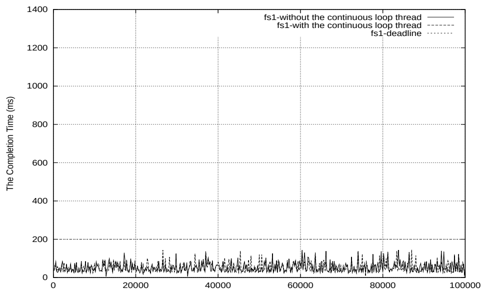


(c) FS_3-With reserve

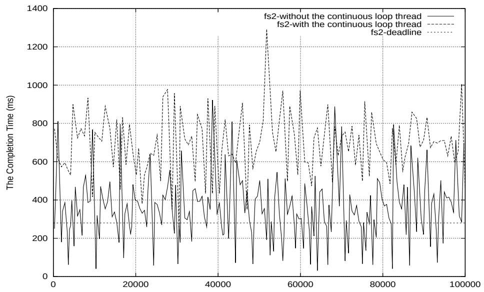


(d) FS_4-With reserve

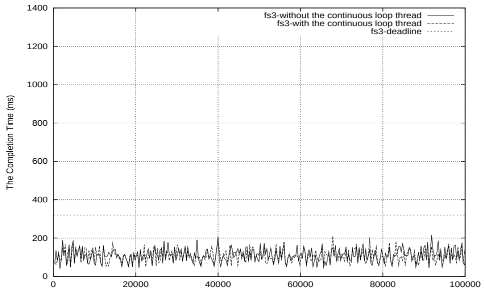
Figure 5. The completion of disk access threads with/without disk bandwidth reservation. At the same time, all the three reserved CPU tasks meet *all* their deadlines.



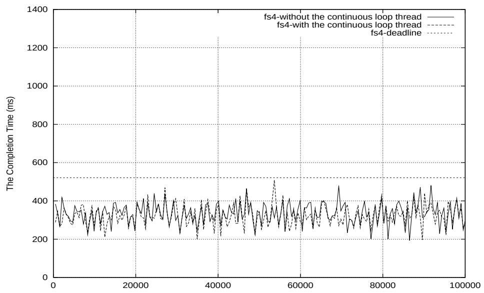
(a) FS_1-With reserve



(b) FS_2-No reserve



(c) FS_3-With reserve



(d) FS_4-With reserve

Figure 6. The behavior of reserved disk access threads with an unreserved thread *continuously* consuming all available CPU. At the same time, all the three reserved CPU tasks meet *all* their deadlines.

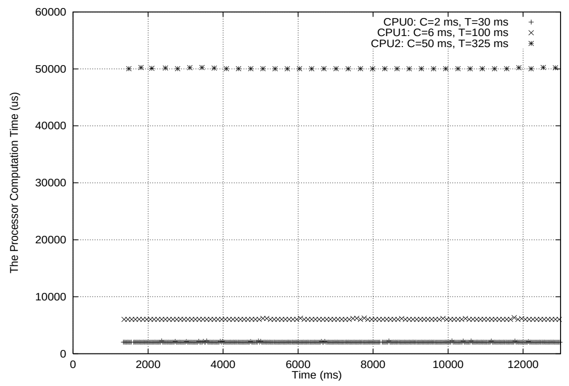


Figure 7. The processor usage of the reserved CPU threads with the disk ccess threads and an unreserved thread *continuously* consuming all available CPU. Both CPU and disk bandwidth reservations are being satisfied simultaneously.

which favors the disk resource is a better approach than fixed slack-sharing for increasing the slack to the scheduler server. It simultaneously reduces the problem of rejects from the deadline-monotonic admission control to BF_{RES}s and AF_{RES}s. This biased slack-sharing scheme is as follows:

Let there be n disk bandwidth reservations in the system as $\tau_1, \tau_2, \ldots, \tau_n$ where $D_1 \leq D_2 \leq \ldots \leq D_n$. S_i, S_{A_i}, S_{B_i} are the slack available in SER_{RES}, BF_{RES} and AF_{RES} of τ_i respectively, then

$$S_{B_i} = C_{A_i} + (d_{unit} * (2i - 1))$$

$$S_{A_i} = C_{B_i} + (d_{unit} * (2i))$$

In our case, we picked $d_{unit} = 5ms$. This approach reduces the processor competition among BF_{RES}s and AF_{RES}s because every time one more pair of BF_{RES} and AF_{RES} is added into the system, all slacks of BF_{RES}s and AF_{RES}s are increased and adjusted in order of the deadlines of the applications they are responsible for. Therefore, the processor utilization, which is seen by any threads that have deadlines falling between the deadlines of BF_{RES}s, AF_{RES}s and file-access threads, for all BF_{RES}s and AF_{RES}s is increased by smaller values if there are more reservations in the system. This is different from the linear increase of processor utilization in the case of fixed slack-sharing. This approach is effective under the assumption that only a small number of application threads in the system have deadlines between d_{unit} and T_s. If the number of reserved disk access threads are increased until the deadline of BF_{RES} and AF_{RES} are beyond T_s, other application threads will be penalized.

5. Concluding Remarks

Timely access to multiple resources is known to be an NP-complete problem in the general case. In this paper, we have considered the problem of co-scheduling accesses to multiple resources which are controlled from a single controlling resource. This problem is common and arises when a processor must be used to control and manage peripheral devices to obtain disk and/or network bandwidth. We have proposed the use of a Co-operative Scheduling Server, `CSS`, which serves as a dedicated resource management server that can use a controlling resource and a controlled resource. A conjunctive scheme that requires successful admission on both resources is therefore used. We have applied this scheme successfully to obtain guarantees in the use of both disk bandwidth *and* processor cycles. Our experiments show the effectiveness of the disk bandwidth guarantees under both heavy processor and competing disk workloads. The programming model we assume led us to a fixed slack-sharing scheme. However, the processor capacity needed by this filesystem `CSS` (named `FS-CSS`) is found to be rather pessimistic and conservative. This aspect can be effectively addressed by the use of a biased slack-sharing scheme that favors the bottleneck resource or the introduction of pipelining with additional end-to-end latency at the application level.

Future work on the topic can proceed along several fronts. The performance-sensitive parameters in our `CSS` model need to be studied at greater depth. The choice of the server period T_s impacts not only the effectiveness of the `CSS` in meeting application-level deadlines, but also the granularity of the processor capacity usage. In addition, our admission control of disk access is very conservative. We use the maximum seek and rotational time of the disk head, while in practice, the actual seek and rotational delays would be much smaller and variant. Optimized disk layout schemes and exploitation of the statistical behavior of the disk can enhance the overall performance of the `CSS` model. Finally, an implementation of `NT-CSS` (the network server using `CSS` model) is needed to co-schedule multiple controlled resources inside a single node for simultaneous application usage of network bandwidth, disk bandwidth and processor capacity. The randomness of incoming network packets can also have an adverse effect on the `NT-CSS` model. This effect can be reduced by (a) minimizing the processing time of packet classifiers inside the protocol stack, (b) adding a buffer in the kernel so the protocol stack merely adds incoming packets into the buffer, and (c) by having the `NT-CSS` periodically read the buffer and dispatching to the application based on the processor reservation of `NT-CSS`.

References

[1] A. Mehra, A. Indiresan and K. Shin. Resource Management for Real-Time Communication: Making Theory Meet Practice. *Technical Report. CSE-TR-281-96. Computer Science and Engineering Division, The University of Michigan.* January 1996.

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

[3] C. Lee, K. Yoshida, C. Mercer and R. Rajkumar. Predictable communication Protocol Processing in Real-Time Mach. *Proceedings of IEEE Real-Time Technology and Applications Symposium.* June 1996.

[4] C. W. Mercer, R. Rajkumar and J. Zelenka. Temporal Protection in Real-Time Operating Systems. *Proceedings of the* 11_{th} *IEEE Workshop on Real-Time Operating Systems and Software,* May 1994, pp. 79-83.

[5] C. W. Mercer, S. Savage and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. *Proceedings of the IEEE international Conference on Multimedia Computing and Systems.* May 1994.

[6] C. W. Mercer and H. Tokuda. Preemptibility in Real-Time Operating Systems *Proceedings of the* 13_{th} *IEEE Real-Time Systems Symposium.* December 1992.

[7] D. Golub, R. W. Dean, A. Forin and R. F. Rashid. Unix as an Application Program. *Proceedings of Summer 1990 USENIX Conference.* June 1990.

[8] D. P. Anderson, Y. Osawa and R. Govindan. Real-Time Disk Storage and Retrieval of Digital Audio/Video Data. *Technical Report No. UCB/CSD 91/646. Computer Science Division (EECS), University of California Berkeley.* August 8, 1991.

[9] H. Tokuda, C. W. Mercer, Y. Ishikawa and T. E. Marchok. Priority Inversion in Real-Time Communication. *Proceedings of* 10_{th} *IEEE Real-Time System Symposium.* December 1989.

[10] J. Blazewicz, W. Cellary, R. Slowinski and J. Weglarz. Scheduling under resource constraints – Deterministic Models. *Annals of Operations Research. Volumn 7.* Baltzer Science Publishers, 1986.

[11] K. Jeffay, F. D. Smith, A. Moorthy and J. Anderson. *Proportional Share Scheduling of Operating System Services for R eal-Time Applications*. IEEE Real-Time Systems Symposium, Madrid, Spain, December 1998.

[12] Joseph, M. and Pandya. Finding Response Times in a Real-Time System. *The Computer Journal (British Computing Society)* 29(5):390-395, October, 1986.

[13] K. Jeffay. Scheduling Sproadic Tasks with Shared Resources in Hard Real-Time Systems. *Proceedings of the* 13_{th} *IEEE Real-Time Systems Symposium, pp. 89-99.* IEEE December, 1992.

[14] Lehoczky, J. P., Sha, L., Strosnider, J. K. and Tokuda, H. Fixed Priority Scheduling Theory for Hard Real-Time Systems. *Technical Report, Department of Statistic, Carnegie Mellon Univeristy.* 1991.

[15] Lehoczky, J. P., Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. *Proceedings of the IEEE Real-Time Systems Symposium.* December 1990.

[16] Lehoczky, J. P., Sha, L. and Ding Y. The rate Monotonic Scheduling Algorithm – Exact characterization and Average-Case Behaviour. *IEEE Real-Time System Symposium.* December 1989.

[17] Leung, J. Y. and Whitehead, J. On the complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation 2, 4* (Dec, 1982) 237-250.

[18] Liu, C. L. and Layland, J. W. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *JACM 2-(1): 46-61.* 1973.

[19] Locke, C. D., Vogel, D. R. and Lucus, L. Generic Avionics Software Specification. *Technical Report, Software Engineering Institue, Carnegie Mellon University.* 1990.

[20] Manacher. G. K. Production and stabilization of real-time task schedulers. *J. ACM 14'3 (July 1967),* 439-465.

[21] Nakajima, T., Kitayama, T., Arakawa, H. and Tokuda, H. Integrated Management of Priority Inversion in RT-Mach. *Proceedings of the IEEE Real-Time Systems Symposium.* December 1993.

[22] R. Rajkumar. Synchronization in Real-Time Systems: A Priority Inheritance Approach, *Kluwer Academic Publishers, 1991.*

[23] R. Rajkumar, C. Lee, J. Lehoczky and D. Siewiorek. A Resource Allocation Model for QoS Management. *Proceedings of the IEEE Real-Time Systems Symposium.* December 1998.

[24] R. Rajkumar, K. Juvva, A. Molano and S. Oikawa. Resource Kernels: A Resource-Centric Approach to Real-Time System. *Proceedings of the SPIE Conference on Multimedia Computing and Networking,* Jan 1998.

[25] Sha, L. and Goodenough, J. B. Real-Time Scheduling Theory and Ada. *Computer.* May 1990.

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

[27] Sprunt, H. M. B., Sha, L. and Lehoczky, J. P. Aperiodic Task Scheduling on Hard Real-Time Systems. *The Real-Time System Journal.* June 1989.

[28] T. Nakajima and H. Tezuka Virtual Memory Management for Interactive Cotinuous Media Application *IEEE International Conference on Multimedia Computing and Systems. (ICMCS'97)*

[29] Tindell, K. An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks. *Tech. Rept. YCS189, Department of Computer Science, University of York.* December 1992.