A Cyclic-Executive-Based QoS Guarantee over USB

Chih-Yuan Huang,Li-Pin Chang, and Tei-Wei Kuo
Department of Computer Science and Information Engineering
National Taiwan University, Taipei, Taiwan 106, ROC
{d89002,d6526009,ktw}@csie.ntu.edu.tw
Fax: +886-223628167

Abstract

Universal Serial Bus (USB) is a popular standard for PC peripheral devices because of its versatile peripheral interconnection speci£cations. USB not only provides simpli£ed hardware connectors but also supports for various bus traf£cs, such as isochronous and bulk transfer activities. Although the USB speci£cations provide a way for users to specify the upper bound on the number of bytes for each data transfer in a 1ms time frame, little work is done to provide QoS guarantees for devices (e.g., the lower bound on the bytes for each device type in a 1ms time frame) and a mechanism in enforcing the guarantees. In this paper, we propose a cyclic-executive-based bandwidth reservation and scheduling method to support QoS guarantees over USB, especially for those isochronous bus activities. The proposed bandwidth reservation and scheduling method could reserve USB bandwidth for devices in an on-demand fashion. The capability of the proposed scheme was shown by the implementation and demonstration of a USB-based surveillance system prototype which adopted the proposed scheme.

1 Introduction

With the advance of hardware and software technology, computer systems are now very modularized. Various subsystems are interconnected with proper interface de£nitions. Consider I/O subsystems as an example. Various I/O devices are manufactured and connected to few common interfaces, such as SCSI, USB, and IEEE-1394. The needs for resource allocation no longer remains at the so-called kernel part. Instead, I/O subsystems (and of course many other subsystems) now become one of the major players in providing a quality-of-service (QoS) guarantee for applications. The playing of a video stream from a disk serves as a good example in exploring the major parties involved in the resource allocation of system resources.

Real-time scheduling algorithms [5] were often adopted by researchers in enforcing resource reservation. In the past decades, researchers and the industry have developed techniques in providing the QoS guarantees over traditional operating systems. RTLinux [7] is a real-time extension of Linux to deliver hard real-time capabilities and regular Linux services to applications at the same time. Modi£cations to the Linux operating system were also made [2, 6] to provide real-time and QoS scheduling for various resources. In addition to the QoS reservation for CPU time, researchers started exploring the resource allocation problems and their QoS issues over various buses. In particular, the control area network (CAN), which is a serial communication protocol for distributed real-time system, was studied by Hon and Kim [3] in proposing a bandwidth reservation algorithm. Kettler, et al. [4] developed formal scheduling models for several types of commonly used buses, e.g., PCI and MCA, in PC's and workstations.

The objective of this research is to explore the QoS issues of USB subsystem. USB is designed to support many types of devices, such as human interface devices (e.g., keyboard and mouse), block devices (e.g., disks), communication transceivers, stereo speakers, video cameras, etc. The data transfer modes on USB could be classi£ed into four categories: isochronous transfer, interrupt transfer, control transfer, and bulk transfer. Isochronous transfer and interrupt transfer are periodic data transfers, while control transfer and bulk transfer introduce aperiodic bus activities. Different types of devices require different bandwidths and ways to interact with USB host controllers, where a host controller manages the bandwidth of a USB bus. For example, a human interface device (HID) demands a periodic but light bus workload. A storage device requires a best-effort service from the corresponding host controller. On the other hand, an operating system must manage a number of devices through USB simultaneously, and devices would compete for a limited bus bandwidth. How to properly manage USB bandwidth is of paramount importance if reasonable QoS requirements are, indeed, considered.

Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)
1080-1812/03 $17.00 © 2003 IEEE

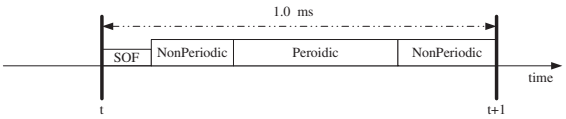

Figure 1. The bandwidth reservation of the USB 1ms time frame.

Although the original design of USB does have an interface de£nition for QoS (in terms of the percentage of the bandwidth for each type of devices and the upper bound on the number of bytes per data transfer for a device type in a 1ms time frame), there is little work being done for the QoS guarantee of each device (e.g., the lower bound on the number of bytes per device in a 1ms time frame). According to the USB speci£cations, isochronous and interrupt transfers could be allocated up to 90% of the total bandwidth of a USB bus. Control transfers could have up to 10% of the total USB bandwidth. Bulk transfers are serviced in a best-effort fashion to utilize the remaining USB bandwidth. The USB bandwidth reservation scheme in each 1ms time frame is described in Figure 1, where SOF stands for the starting of the frame, and time frames of USB 1.1 are all in 1ms. Each USB host controller exercises a series of data structures which are constructed by the operating system to manage devices, as shown in Figure 2. Those data structures are called *Transfer Descriptors*. The technical issue is how to set up proper in-core data structures so that a host controller could behave and communicate with devices as users request and meet their QoS requirements. The research objective of this paper is to propose a proper bandwidth reservation and scheduling method to support the speci£ed QoS requirements. We shall also design a real-time scheduling layer inside the USB driver to let the entire system complying with the USB speci£cation.

The rest of this paper is organized as follows: In Section 2, we present the system architecture of a typical USB subsystem. We then propose a cyclic-executive-based approach to partition resources among devices which need different QoS requirements. Section 3 illustrates the implementation issues and provides system analysis. The experimental results based on a surveillance system prototype are provided in Section 4. Section 5 is the conclusion.

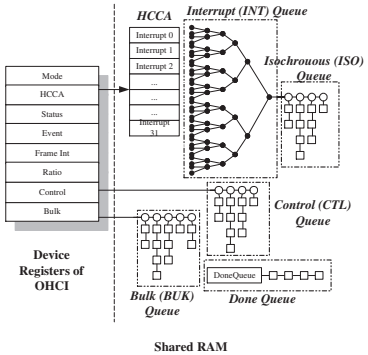

Figure 2. The USB communication channel.

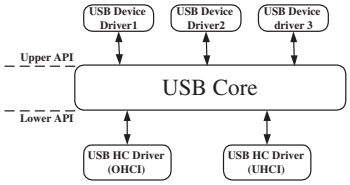

Figure 3. The device driver hierarchy of the USB subsystem.

2 A Real-Time USB System Architecture

In this section, we provide an overview of a USB subsystem on Linux. We propose a real-time USB driver architecture, in which a bus bandwidth reservation and scheduling mechanism is proposed to reserve and allocate USB bandwidth according to the QoS requirements.

USB subsystems on Linux is a layered structure, as shown in Figure 3. The hierarchical architecture provides a better abstraction of USB device drivers so that the kernel could have a transparent view over devices. As shown in Figure 3, there is a core component in the USB subsystem. The USB Core provides a set of consistent and abstracted API's to the upper-level client drivers. The client drivers are hardware-speci£c, and their corresponding devices are accessed by calling the API's provided by the USB Core. Inside the USB Core, the USB control protocol is implemented and proper USB HC (host controller) drivers would be called so that the client drivers do not need to manage how data packets are sent to or received from the devices.

Between the USB Core and the host controller driver, the USB Core translates the requests issued by the USB device drivers into host-controller-awared data structures. The host controller simply handles a memory-resident table called the *USB schedule*. The USB schedule consists of a number of queues of Transfer Descriptors(TD's), as shown in Figure 4. The queues are hooked on the proper device registers of a USB host controller, as shown in Figure 2, where the basic rule in hooking queues on the schedule is on the types of transfers. The actual layout of the USB schedule is host-controller-dependent[1]. The USB schedule is brie¤y illustrated as follows:

[1]There are two major types of host controller: Open Host Controller Interface (OHCI) [10] and Universal Host Controller Interface (UHCI) [9]. We shall focus our discussions on OHCI in this paper, where the basic concept and the operation of OHCI and UHCI are similar

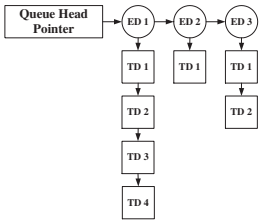

Figure 4. The queue list structure of the OHCI USB schedule.

As shown in Figure 4, an Endpoint Descriptor is at the head of the Transfer-Descriptor list for each device to record the transfer information regarding the device, e.g., the device number for the host controller. For bulk and control data-transfers, the host controller maintains a pointer to the head of the Transfer-Descriptor list for each type, as shown in Figure 2. For interrupt data-transfers, a USB subsystem maintains 32 interrupt entries, which forms a tree structure. Each (internal or leaf) node could be hooked up with a list of Transfer-Descriptors for a device of the interrupt type. The list of Transfer-Descriptors for all isochronous devices is hooked up at the root of the tree structure, as shown in Figure 2. Each interrupt entry will be visited in every 32ms. When an interrupt entry is visited, all devices with a Transfer-Descriptor list on the path between the interrupt entry and the root will be serviced within 1ms. In other words, the isochronous queue will be serviced in every 1ms, provided that time is left after servicing other devices on the path (ahead of the root). While the host controller services a device, it processes all Transfer-Descriptors on its list. When a Transfer-Descriptor is serviced, it is updated to re¤ect the completion of the corresponding work, and the Transfer-Descriptor will be removed from the list and inserted into the DoneQueue (a speci£cal queue for the Transfer Descriptors which are done). The host controller driver will investigate the DoneQueue to return the results to the system.

For the USB client drivers, the control and data transfer are accomplished by interacting with the USB Core through a data structure called the USB request block (URB). Requests are £lled up in URB's and handled by the USB Core asynchronously: A con£gured callback function will be called on the completion of each URB. With the needs of the quality of service support, information regarding timing constraints should be included in URB's as parameters. In our implementation (in Section 3), modi£ed URB's are issued by USB client drivers and then handled by the USB Core. We propose a cyclic-executive-based implementation for the request scheduling in the USB Core layer so that the bandwidth reservations for devices could be accomplished by the scheduling of requests. Furthermore, since the cyclic-executive-based scheduling mechanism proposed in this paper (in the USB Core) is independent of the host controller driver, the resource management extension could be applied to both the UHCI [9] implementation and the OHCI [10] implementation.

3 A Real-Time USB Device Driver

3.1 Overview - A Cyclic-Executive-Based Method

The purpose of this section is to propose a real-time USB device driver and a set of approximation and scheduling methods to allocate USB bandwidth for USB devices. The cyclic executive is a frame-based approach [1]. It is a control structure of programs for interleaving the execution of the period process. We propose a cyclic-executive-based implementation for the USB bandwidth reservation methods in this section. The rationale behind this approach is to install a virtually £xed tree structure for Endpoint-Descriptor queues to have low run-time overheads. The tree structure is only revised when a new device is admitted or open under the admission control method to be proposed later, or when a device is closed. Service requests of devices are intelligently hooked up on the tree structure based on their QoS requirements.

Although the original bandwidth reservation policy for USB [8] only handles isochronous transfers with polling periods being exactly equal to $1ms$, this section considers a general case in which the polling frequencies of isochronous transfers could be like those of interrupt transfers. Note that the polling frequencies of interrupt transfers can be between 1ms to 32ms. The assumption is true under the condition that a layer of our real-time scheduler is built inside the USB drivers, as shown in Figure 5. Note that the original bandwidth reservation policy of isochronous transfers only hooks their Endpoint Descriptors at the root of a polling binary tree, as shown in Figure 2. The real-time scheduler, the period modi£cation policy and its admission control (Section 3.2), and the insertion policy of Endpoint and Transfer Descriptors (Section 3.3) would let the Transfer-Descriptors of isochronous transfers and interrupt transfers be able to be hooked up at virtually any nodes in a polling binary tree.

In order to guarantee the QoS requirements of different devices and to better manage the USB bandwidth, two essential issues must be addressed: (1) admission control policy (Section 3.2) (2) the scheduling of Endpoint and Transfer Descriptors (Section 3.3). We will £rst propose a period modi£cation policy and the corresponding admission control policy in Section 3.2 to determine whether the system can satisfy the QoS requirements of a new request and the existing requests. We then resolve the second issue in Section 3.3. The technical question we face is where to insert Endpoint Descriptors (and their Transfer Descriptors) into queues such that the USB protocol overheads are minimized.

3.2 A Period Modi£cation Policy and Admission Control

This section £rst proposes a period modi£cation policy to move and modify all Endpoint Descriptors (at any node) to the root node and then presents the corresponding admission control policy. We will then present a methodology in the next section to move and modify Endpoint Descriptors back to proper nodes in the tree structure.

For the purpose of USB bandwidth reservation, we include additional information regarding deadlines, service frequencies, and USB bandwidth requirements inside URB, where URB is a data structure used for the interchanging of control and data information between the USB Core and device drivers. In Linux, the USB Core is named as the USB driver (USBD). An abstracted real-time scheduling layer is proposed in the driver hierarchy, as shown in the Figure 5.(b). The newly proposed layer is to communicate between the device driver layer (e.g., USB camera deriver in Figure 5.(b)) and the queues in USBD (e.g., ISO Queue in the £gure). The new layer, i.e., the real-time scheduler, is on the top of USBD such that device drivers would call (or send messages to) the new layer, instead of following the original procedure. The real-time scheduler determines how Transfer Descriptors of a request are inserted into the Endpoint Descriptor of the device to meet the QoS requirements of devices. Note that the real-time scheduler might reject requests for the QoS requirements of devices.

USBD collects the upper-level driver's requests (in terms of URB, as illustrated in the previous sections) and allocates USB bandwidth for each request. Different drivers might need different polling periods to transfer request messages between the peripherals and the corresponding device drivers. The Linux USB OHCI driver restricts periodic service requests (for interrupt and isochronous devices) to a set of £xed periods (e.g., 1, 2, 4, .., and 32ms). Note that the tree structure of the Transfer-Descriptor lists in Figure 2 (in Section 2) determines the frequencies of the traversing of nodes on pathes to the root. If a device has a corresponding node on a path to the root, then the device is serviced in

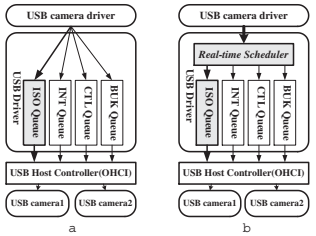

Figure 5. Comparison of two USB driver architecture. (a)the original USB driver architecture. (b)the real-time USB driver architecture.

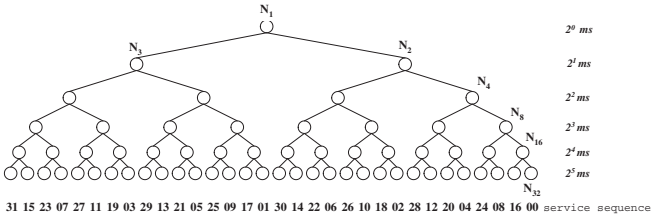

Figure 6. A polling binary tree for the USB scheduler of a USB host controller.

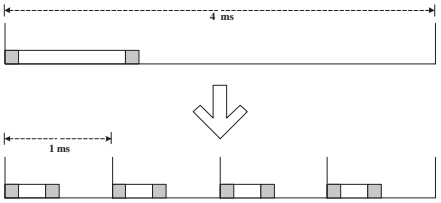

Figure 7. The service of an Endpoint Descriptor for 32 bytes per 4ms is transformed into one for 8 bytes per 1ms. The white and shadowed boxes denote data and the protocol overheads of a payload, respectively.

the 1ms corresponding the path. Because there are 32 leaf nodes, as shown in Figure 2, the tree has 63 internal/leaf nodes. In other words, the structure provides the Host Controller Driver 63 different nodes to hook up Endpoint Descriptors.

A polling binary tree of the 63 nodes provides an abstraction in the handling of isochronous and interrupt transfers. There are £ve levels in the polling binary tree and 32 pathes from leaf nodes to the root node, as shown in Figure 6. The visiting order to each leaf node is marked as the service sequence under the tree in Figure 6, where a visiting occurs for every 1ms time frame. An OHCI host controller travels through each path from a leaf node to the root node of the corresponding to the host controller polling binary tree. A node could have a list of Endpoint Descriptors, and an Endpoint Descriptor could have a list of Transfer Descriptors, as shown in Figure 4. An OHCI host controller processes pending Transfer Descriptors on its way from a leaf node to the root node. The handling of pending Transfer Descriptors on each path should not be more than 1ms; otherwise, the host controller will abort the path and then continue its work to the next path according to the service sequence. Any aborted path is considered as an $overflowed$ path. As a result, any nodes at the i_{th} level (counted from the root node) are for services at every 2^i ms.

Consider a workload with n periodic requests $\{(b_1, p_1), (b_2, p_2), (b_3, p_3), ..., (b_n, p_n)\}$ for devices,

where b_i denotes the number of bytes to be transferred for every p_i ms. Before we introduce the admission control policy, we £rst introduce a period modi£cation policy to move and modify the Endpoint Descriptor of a request at a node to the corresponding Endpoint Descriptor at the root node: When the Endpoint Descriptor of a request is moved from a node to the root node, extra protocol overheads will occur, and the Endpoint Descriptor should be revised to £t the polling periods of the former node and the root node. Figure 7 shows how an Endpoint Descriptor which transfers 32 bytes per 4ms is transformed into one which transfers 8 bytes per 1ms. Such a transformation is applicable to many USB isochronous devices since the transfer rate remains. However, beside the protocol overheads, when the number b_i of bytes per transfer could not divide the original period p_i, then another overheads occurs, i.e., an additional bandwidth requirement of $((\lceil \frac{b_i}{p_i} \rceil)p_i - b_i)$ bytes per p_i time units.

The admission control policy is as follows: Because the period modi£cation policy moves all Endpoint Descriptors to the root node, and the root node is serviced for every 1ms, all requests are schedulable if the services of all Endpoint Descriptors could be done within 1ms. Because only 1500 bytes could be transferred within every 1ms under USB1.1, the total number of bytes for the services of all Endpoint Descriptors should be no more than 1500. That is,

$$\sum_{i=1}^{n} \left((\lceil \frac{b_i}{p_i} \rceil + O) \right) \leq 1500 \qquad (1)$$

In Formula (1), O denotes the protocol overheads of an USB payload. The overheads include packet preambles, packet headers, and some other necessary £elds, as shown in Figure 7. Speci£cally, the overheads are 9 bytes for isochronous transfers and 13 bytes for interrupt transfers. The term $\lceil \frac{b_i}{p_i} \rceil$ denotes the payload data size after the period modi£cation. For example, consider an Endpoint Descriptor of an isochronous transfer which represents 50-byte transfer per 16ms. After the period modi£cation policy, $(\lceil \frac{50}{16} \rceil + 9)$ bytes must be transferred for corresponding the Endpoint Descriptor within 1ms.

3.3 Insertions of Endpoint and Transfer Descriptors

The purpose of this section is to choose proper nodes in the tree structure for the inserting of proper Endpoint and Transfer Descriptors. We start with the results of the period modi£cation policy by reinserting Endpoint and Transfer Descriptors back to proper nodes in the tree structure. The objective is to move (and modify) Endpoint and Transfer Descriptors to nodes of larger heights to minimize the protocol overheads and the additional bandwidth requirements due to the period modi£cation policy, i.e., the difference between $(\lceil \frac{b_i}{p_i} \rceil)$ and $\frac{b_i}{p_i}$ in Formula (1).

Given a collection of admitted requests (in terms of their Endpoint and Transfer Descriptors) $T = \{(b_1, p_1), (b_2, p_2), \cdots, (b_n, p_n)\}$, we shall try to push Endpoint Descriptors (ED's) and their Transfer Descriptors (TD's) at the root node down to nodes of larger heights as far as possible. The further we push the Endpoint and Transfer Descriptors away from the root node, the lower the protocol overheads would be. There are two basic guidelines for this *ED/TD reinsertion policy* (which will be presented later): The £rst guideline is that the destination node of an original ED (and its TD's) should not have a polling period longer than the period p_i of the corresponding request. For example, let a video camera demand a 4096-byte transfer per 8ms. If the data on the camera is not transferred to the host controller within a 8ms window, the data in the memory buffer of the camera might be overwritten by new data. The second guideline is that the ED/TD reinsertion policy should meet the 1ms time frame for each path of a tree structure, as shown in Figure 6. Since only 1500 bytes could be transferred within every 1ms, the total number of bytes for the services of all Endpoint Descriptors on a path should be no more than 1500.

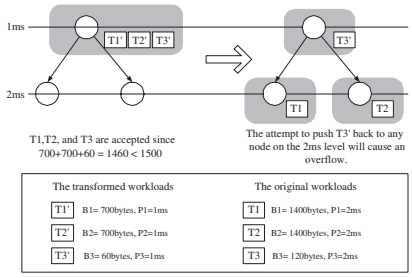

Figure 8. A combinatorial problem occurred in the minimization process. Note that the protocol overheads are not included in this example.

We shall illustrate the idea of the ED/TD reinsertion policy and the guidelines by the following example: To simplify the presentation, the protocol overheads are not shown in this example. Suppose a workload, as shown in Figure 8, is under considerations: $T = \{T_1 = (1400bytes, 2ms)$, $T_2 = (1400bytes, 2ms)$, and $T_3 = (120bytes, 2ms)\}$. With the period modi£cation policy presented in Section 3.2, the workload is £rst revised as follows: $T' = \{T_1' = (700bytes, 1ms)$, $T_2' = (700bytes, 1ms)$, and $T_3' = (60bytes, 1ms)\}$. The workload passes the admission control because $700 + 700 + 60 \leq 1500$ (Please see Formula (1)). In order to minimize the protocol overheads and the additional bandwidth requirements, we move (and modify) Endpoint and Transfer Descriptors to nodes of larger heights: Suppose that we £rst revise the ED's and the TD's of T_1' and T_2' and move them down to nodes of the sec-

ond level. When we want to push T_3' down to any one of the nodes at the second level, we would notice that the new payload size will be 120 bytes because nodes at the second level have a 2ms polling period. It is not feasible for the pushing of T_3' because of the violation of the second guideline (although the £rst guideline is still satis£ed), where $120 + 1400 > 1500$. As a result, the ED and its TD's of T_3' must stay at the root node. As astute readers may notice, the ED/TD reinsertion problem is a combinatorial problem. It is apparently not an easy problem.

For the rest of this section, we present a greedy algorithm for ED/TD reinsertion: The main idea is to £rst sort all ED's in the increasing order of their original periods, where each ED corresponds to a request. Within each iteration, the algorithm tries to push the ED with the largest original period which has not been considered so far down to a node of a larger height. Whenever an ED is considered to move down by one level, there are two choices: right child node n_R or left child node n_L. Let $S(n)$ denote the sum of the numbers of the bytes (plus the protocols overheads) for all of the ED's (and their TD's) hooked up at the node n and all of the nodes in the subtree with n as the root. That is, $S(n) = S(n_L) + S(n_R) + l_n$, where l_n denotes the sum of the numbers of the bytes (plus the protocols overheads) for all of the ED's (and their TD's) hooked up at the node n. The heuristics of the algorithm is to prefer the child node with a smaller value of the function $S(n)$.

4 Performance Evaluation

4.1 Experimental Environments

An USB-based surveillance system was implemented based on the proposed cyclic-executive-based bandwidth reservation method. In this section, we shall £rst introduce the surveillance system as our experimental platform. The system con£guration and performance metrics are presented. The effectiveness of the proposed bandwidth management method was evaluated in terms of the effectiveness of the admission control policy, the protocol overheads, and the actual service time of the cameras.

The surveillance system was divided into the server side and the client side. At the server side, the server managed several USB digital video cameras through an OHCI host controller to provide multiple video streams for monitoring purposes. At the client side, we had several PDA's which ran WinCE. Video streams were delivered to the clients over a 802.11b wireless network. The clients could request available video streams from the server with speci£ed QoS parameters. User could selectively improve the video quality (e.g., video resolution and/or frame rate) of any interested video stream to have a better quality. The experiments consisted of two parts. In the £rst part of experiments, we evaluated the effectiveness of the proposed USB bandwidth reservation method in terms of the number of accepted workloads and the bandwidth utilizations after the protocol overheads were minimized. In the second part of experiments, we evaluated different workload con£gurations over the cyclic-executive scheduler to observe the behaviors of USB devices. This part of experiments was to verify whether the QoS requirements were correctly guaranteed by the cyclic-executive-based scheduler or not.

4.2 Overheads Due to Bandwidth Reservation

This section provides a performance evaluation over the proposed USB bandwidth reservation method in terms of the number of workloads accepted and the actual bandwidth usages.

4.2.1 Request Sets and Metrics

A number of workloads were generated to evaluate the performance of the proposed admission control policy. The number of requests per workload was 20. The periods and the payload sizes of requests were generated based on the following parameters: Supposed that an USB device demanded a periodic request to receive b bytes per p ms. We de£ned the "bandwidth utilization" of the request as $\frac{b/1500}{p}$. Note that 1500 denotes the maximum number of bytes transferred over USB in a 1ms time frame. The USB bus was stressed by a very high utilization, i.e., 93%, excluding any protocol overheads. The bandwidth utilizations of requests were randomly distributed among the requests. The periods of requests were selected among the following ranges [1ms, 8ms], [8ms, 32ms], and [1ms, 32ms]. The lengths of all periods were set as a power of 2. The payload size of a request was determined based on its utilization and period. $1,000$ workloads were evaluated for each period range, and their results were averaged. The performance of an exhaustive-search-based optimal algorithm was also evaluated to compare with our proposed method based on the metric "acceptance ratio" which was the ratio of the number of accepted workloads to the total number of workloads. The exhaustive-search-based optimal algorithm tried all possible combinations exhaustively to insert ED/TD's (generated by the requests) into the polling binary tree.

The performance of the ED/TD reinsertion algorithm was evaluated to see how much protocol overheads could be reduced in the process of reinsertion. We started from the point where ED's and TD's of all requests were hooked up at the root and then reinsert them back to proper nodes in the polling binary tree. We called the USB bandwidth utilization assigned to a request as the "root-resident bandwidth utilization" because it was with the protocol overheads at the root. Note that when the ED and TD's of a request was moved away from the root, then USB bandwidth utilization

would be reduced because of less protocol overheads. The total root-resident bandwidth utilizations of workloads were generated with a bound ranging from 10% to 100%. Each workload consisted of 20 requests when a workload was generated, and the entire root-resident bandwidth utilization was distributed randomly among the requests. After the root-resident bandwidth utilization of each request was determined, the periods of a request was then selected among the ranges [1ms, 8ms], [16ms, 32ms], and [1ms, 32ms]. The payload sizes of requests were calculated according to their assigned root-resident bandwidth utilizations and periods.

4.2.2 Experimental Results

Figure 9 shows the acceptance ratios of the proposed admission control policy and the exhaustive-search-based optimal algorithm. The X-axis denotes the range of the request periods of the workloads, and the Y-axis denotes their acceptance ratios. As shown in Figure 9, the acceptance ratio of the proposed admission control policy was close to that of the exhaustive-search-based optimal algorithm when the period range was selected among ranges [1ms, 8ms] and [1ms, 32ms]. When the periods of requests were chosen in the range [16ms, 32ms], the proposed admission control policy would reject more workloads than the exhaustive-search-based optimal algorithm. It was because of the protocol overheads due to the period transformation of the admission control policy (Please see Figure 7).

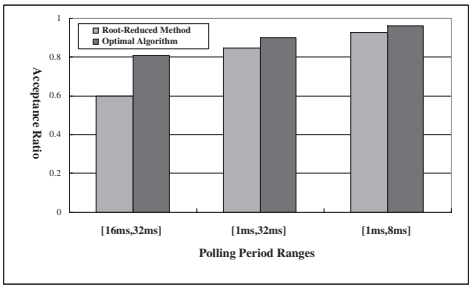

Figure 9. Acceptance ratios under different period ranges and different admission control policies.

Figure 10 showed the results of evaluating the saving of protocol overheads by the ED/TD reinsertion algorithm. The X-axis denotes the "root-resident bandwidth utilizations" of the workloads, and the Y-axis denotes the bandwidth utilizations after the reinsertions of requests were done. The distance between the ceiling (i.e., the line for the bound of bandwidth utilization) and the top of the corresponding bar of a set of workloads (for each selected period range) represented the amount of protocol overheads saving due to the ED/TD reinsertion algorithm. The farther the distance, the better the saving was. Figure 10 showed that

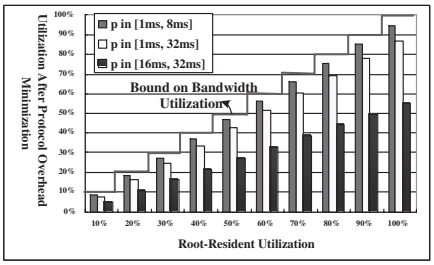

Figure 10. USB bandwidth utilizations v.s. requests periods distribution

the saving was best achieved when the periods of requests were relatively long, i.e., in the range [16ms, 32ms]. It was because of a higher possibility in inserting the ED's/TD's of requests with long periods into nodes with lower levels of the polling tree.

4.3 Bandwidth Reservation and Workload Distribution

4.3.1 Request Sets and Metrics

	Workload 1		Workload 2	
	Camera 1	Camera 2	Camera 1	Camera 2
Payload Size (bytes)	620	310	310	620
Polling Period (ms)	1	2	2	4
Run Duration(sec)	600	600	600	600
Video Resolution(pixels)	176*144	176*144	176*144	176*144
Bandwidth Utilization	41.3%	10.3%	10.3%	10.3%

Table 1. Experimental Workloads. Note that protocol overheads were not included.

In this part of experiments, we evaluated the cyclic-executive-based scheduler under two different workloads to see if the scheduler could satisfy the QoS requirements of the cameras. The two workloads had different polling periods and payload sizes for requests. The workloads were summarized in Table 1. We tried two different ratios of bandwidth utilization for the two cameras: 4:1 and 1:1. We adopt two metrics. The £rst metric is to collect the service time points of USB devices, where a service time point is the time when the corresponding camera was serviced by an USB host-controller. The service time points were collected in the interrupt service routine of the USB host-controller driver. The distribution of the service time points of the two cameras under the two workloads could show whether the cameras were serviced at the proper frequencies. The second metric was the number of bytes received from each camera. The metric was presented as the percentages of the number of bytes received from a camera to the total number

of bytes received from the two cameras in one second. The duration of each run of the experiments was ten minutes (600 seconds).

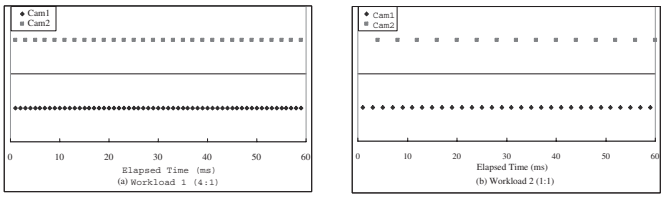

Figure 11. The distributions of service time points of each camera over time. (a) under a 4:1 reservation ratio. (b)under a 1:1 reservation ratio.

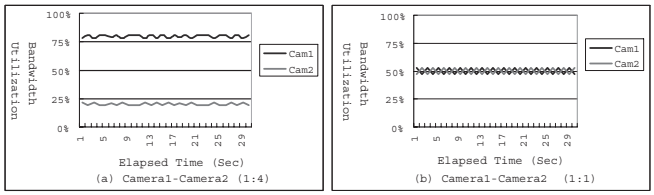

Figure 12. The number of bytes received from a camera to the total number of bytes received from the two cameras. (a) under a 4:1 reservation ratio. (b) under a 1:1 reservation ratio.

4.3.2 Experimental Results

Figure 11 showed the distributions of service time points for the two cameras. The X-axis denotes the time elapsed for an experiment in milli-second, and there were two sets of service time points which corresponded to camera 1 and camera 2, respectively. As shown in Figure 11, the distributions of the service time points of the two cameras could be compared with the polling periods of the cameras in Table 1. Figure 11 shows the results of the £rst 60ms in the experiments because the rest of the results for the 10 minutes were similar to those of the £rst 60ms. Figure 12 showed the number of bytes received from each camera in terms of their percentages of the total USB bandwidth. The distribution of the service time points of the two cameras under the two workloads showed whether the cameras were serviced at the given frequencies. The X-axis denotes the time elapsed so far in seconds, and the Y-axis denotes the percentages. The results in Figure 12 showed that the bandwidth utilizations of the cameras met the parameters speci£ed in Table 1 such that the QoS requirements were guaranteed.

5 Conclusion

Modern operating systems are often pretty modularized. The delivering of QoS guarantees to applications is the responsibility of not only the kernel but also all of the subsystems involved in the servicing of the applications. In this paper, we proposed a cyclic-executive-based methodology to guarantee the QoS requirements of USB devices, where a £xed polling binary tree is adopted for USB bandwidth scheduling. A bandwidth reservation algorithm is proposed to partition available USB bandwidth among devices which need different attentions from the system. Our work results in a more precise way in guaranteeing the QoS requirements of devices and better USB bandwidth utilization could be achieved. For the future research, we shall further explore the USB bandwidth reservation for USB 2.0 which supports transfer rate up to 480 bits per second. The combination of USB 1.1 and 2.0 bandwidth reservation could be a very challenging issue.

References

[1] T.P. Baker and A. Shaw, “The cyclic executive model and Ada,” Real-Time Systems Symposium, 1988.

[2] S. Childs and D. Ingram, ”The Linux-SRT Integrated Multimedia Operating Systems: Bring QoS to the Desktop,” IEEE 2001 Real-Time Technology and Applications Symposium, Taipei, Taiwan, ROC.

[3] S.H. Hong and W.-H Kim, “Bandwidth allocation in CAN protocol,” Proceedings of the IEEE Control Theory and Applications, Jan 2000.

[4] K. A. Kettler, J.P. Lehoczky, and J. K. Strosnider, “Modeling bus scheduling policies for real-time systems,” IEEE Real-Time Systems Symposium, Dec, 1995.

[5] C. L. Liu and J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment,” *JACM*, Vol. 20, No. 1, January 1973.

[6] Y.-C. Wang and K.J. Lin, ”Implementing a General Purpose Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel,” IEEE Real-Time Systems Symposium, Arizona, USA, 1999.

[7] V. Yodaiken, “The RT-Linux approach to hard real-time,” Department of Computer Science Socorro NM 87801.

[8] Compaq, Intel, Microsoft, and NEC, “Universal Serial Bus Speci£cation,” 1998.

[9] Intel, “Universal Host Controller Interface (UHCI) Design Guide, Revision 1.1,” 1996.

[10] Compaq, Microsoft, and National Semiconductor, “Open Host Controller Interface Speci£cation for USB,” 1996.