Real-Time USB Communication in the Quest Operating System

Eric Missimer, Ye Li and Richard West

Computer Science Department
Boston University
Boston, MA 02215
Email: {missimer,liye,richwest}@cs.bu.edu

***Abstract*—This paper describes a real-time USB 2 sub-system for the Quest operating system. Quest is designed for real-time embedded systems. Such systems need to interact with their environment using sensors and actuators. On many embedded platforms today there is support for basic serial, USB 2.0 and 100 Mbps Ethernet. Of these, USB 2.0 supports the highest throughput, while also supporting real-time communication.**

We show how the Quest USB 2.0 sub-system improves upon some of the deficiencies in USB software stacks in systems such as Linux through experimental evaluation. We demonstrate that the Quest USB sub-system is capable of predictable bandwidth allocation and increased overall performance. By dynamically reordering transaction requests, Quest's USB sub-system is able to avoid unnecessary admission control rejections. Additionally, we are able to provide real-time guarantees for asynchronous USB transactions such as bulk transfers, which are typically treated in a best-effort manner. Real-time guarantees for bulk transactions are necessary for any system interacting with devices that implement bulk endpoints such as in a real-time file system. The paper also introduces an algorithm for USB scheduling that accepts more requests and provides bulk transfer guarantees, for cases where Linux fails.

I. Introduction

This paper focuses on the development of a real-time communication framework for an operating system we are developing, called Quest [1]. Quest is intended for use in embedded systems that must interact with their environment via a collection of sensors and actuators. Many embedded systems are built on single board computers, with I/O connectivity limited mostly to low-bandwidth serial links. Communication standards such as RS232 [2] are typically supported on these platforms, along with protocols for a Serial Peripheral Interface (SPI) bus [3], I^2C [4], or Two-Wire Interface (TWI). However, for many emerging real-time applications, sensors require greater bandwidth than that supported by simple serial bus protocols. For example, HD video cameras may generate data streams with bit-rates of several megabits per second. In contrast, many universal asynchronous receiver-transmitter (UART) chips are limited to baud rates that are an order of magnitude less.

This work is supported in part by NSF Grant #1117025.

On popular single board computers such as the Beagleboard [5] from Texas Instruments, or the Raspberry Pi [6], connectivity with other devices is possible using USB 2.0 or 100Mbps Ethernet. To date, higher bit-rate communication technologies such as Thunderbolt [7] and USB 3.0 [8] are not typically available on many low-cost single board computers. While communication technologies such as CAN [9] and Flexray [10] exist, these are either more expensive or are still limited to bandwidths far below that offered by Ethernet or USB. For this reason, we are developing a communication infrastructure in Quest based on the ubiquitous USB 2.0 standard. In particular, we envision the application of real-time communication protocols over USB to be used to not only interact with sensors and actuators, but also to support real-time communication between computer nodes in a distributed embedded system.

One of the limitations with USB is that host-to-host, or *peer* communication is not natively supported. Devices typically act as slaves and connect to computers, or hosts, which act as masters. The masters are responsible for establishing a communication link to peripheral devices, which only need to support simplified signaling logic.

The USB "on-the-go" (OTG) specification supports dual-role devices, allowing them to act as either masters or slaves, as necessary. As an example, a smartphone device might communicate with a host computer to allow access to its storage. In other situations the smartphone may wish to output photographic images to a printer. In this way, the smartphone is able to take on the role of either a master or slave, depending on the entity with which it communicates. Moreover, star, ring and tree-based communication topologies can be established, to allow devices to operate in a distributed embedded system. This has motivated us to consider USB as the bus standard for real-time communication in our system. While we focus on USB 2.0 or, more specifically, the Extended Host Controller Interface (EHCI) [11], we believe that porting a real-time communication infrastructure to USB 3.0 should be relatively straightforward.

In comparison to USB approaches for systems such as Linux, the Quest RT-USB framework is able to:

- guarantee bandwidth allocations when opening connections with communication endpoints,
- dynamically reorder transfer requests to avoid unnecessary rejections, and
- provide real-time guarantees for both asynchronous and periodic transfers.

The Quest RT-USB sub-system guarantees bandwidth allocations for communication paths between endpoints using admission control. When a transfer request is received, existing requests are reordered in their framelist schedules to avoid unnecessary rejections. That is, if a feasible schedule is possible, then attempts are made to dynamically reorder transactions so that the success rate for communication requests is maximized. Dynamic requests can occur for a new device or to change data rates for a currently active device. For example, if the frame rate and/or image quality of a USB camera needs to be changed, this could result in the need to dynamically reorder transactions. For bandwidth guaranteed requests, we focus on USB high-speed devices due to the fact that both OTG and modern sensors such as USB cameras are capable of high-speed throughput.

Quest's USB sub-system temporally isolates periodic requests so that soft real-time guarantees are achievable on asynchronous transfers, such as for bulk data. This enables bulk devices such as those found in flash storage to be used in a real-time system, providing timing guarantees on accesses to a filesystem, for example.

One of our design goals is to create a real-time communication bus that is capable of host-to-device and peer-to-peer communication. Using custom FPGA-based "USB routers" it is possible to create such a communication bus (Figure 1). A single USB router could appear as multiple USB devices to different hosts. This would enable data communication between hosts or other routers. Given the real-time aspects of USB, we aim to create a real-time bus communication protocol built on top of USB that resembles CAN and Flexray. The first step towards this objective is a real-time USB host sub-system, as described in this paper.

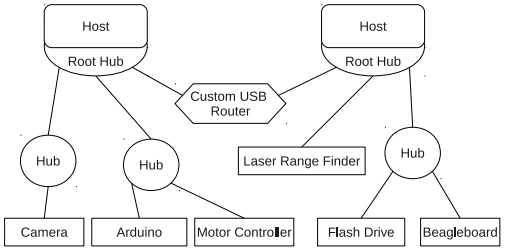

Fig. 1. USB Communication Topology

The next section provides a brief summary of the USB 2.0 and EHCI specifications. This is followed by an introduction to the USB scheduling problem in Section III. In Section IV, we describe the Quest real-time USB architecture in more detail. The experimental evaluation is discussed in Section V, followed by related work in Section VI. Finally, conclusions and future work are discussed in Section VII.

II. USB 2 AND EHCI OVERVIEW

We now provide a brief overview of the USB 2.0 [12] and EHCI [11] specifications.

A. Universal Serial Bus

As stated earlier, Universal Serial Bus (USB) is a master-slave protocol that connects a host computer (the master) to one or more peripheral devices (the slaves). A device operates at one of three possible communication rates: low, full or high speed. These have maximum throughputs of 1.5, 12 and 480Mbps, respectively. Figure 2 depicts the hardware-software structure of both a USB host and device, which communicate over a physical link.

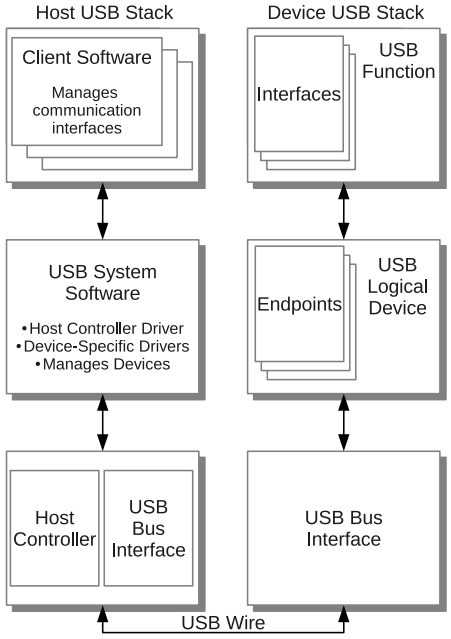

Fig. 2. USB Host and Device Stacks

Each physical device consists of one or more *configurations* that specify how many interfaces it supports, amongst other information. Only one configuration for a given device is active at any time, and it in turn supports one or more *functions*. A multi-function full-speed input device, for example, might have two functions for both a keyboard and a mouse. Each hardware function provides

a collection of *interfaces*, with each interface providing one or more *endpoints* of communication.

Each endpoint specifies the type of transfer mode, whether it is an input or output endpoint, the maximum packet size, how many packets it can receive during a single transaction, and how often transactions should occur in the case of periodic endpoints.

A function can have alternative interfaces to enable or disable certain endpoints and/or change their data rate. For example, USB cameras have different interfaces for a video stream source. Each interface exposes a different version of the endpoint for receiving the video frames and the device driver selects the endpoint that has the minimum data rate required for the camera frame rate and image quality.

The host side of USB communication consists of a hardware *host controller*, and a software stack. The software stack comprises a USB host controller driver, device-specific drivers, and interfaces that allow client applications to communicate with devices.

The USB specification supports four basic *transfer types* for data exchange between a host and a device:

- *Control transfers* - Used for device configuration. No time guarantees are provided by the USB specification. Error detection and checksumming is used to guarantee data delivery. All devices have at least one control endpoint, identified as endpoint zero.
- *Bulk transfers* - Used for devices that do not require time guarantees but do require guaranteed data delivery. An example bulk transfer device would be a USB thumb-drive for mass storage.
- *Isochronous transfers* - Used for devices that require real-time guarantees without retransmissions of erroneous data. The bandwidth and time properties are specified using the packet size, number of packets per transmission, and transmission interval (specified in *micro-frames* for high speed devices and *frames* for low and full speed devices). The interval is always a power of 2 for high speed devices. A camera is an example isochronous device.
- *Interrupt transfers* - Used for devices that require both real-time guarantees and also the correctness of transferred data. If the data is corrupted, it is retransmitted at the next opportunity. The real-time guarantees are specified the same way as for isochronous transfers. A keyboard is an example interrupt device.

USB *transactions* are always initiated by the bus master; peripheral devices only respond to host requests. In USB 1.0/1.1, all transactions occur within a frame set at 1 millisecond. Transactions cannot cross a frame boundary. USB 2.0 supports micro-frames of 125 microseconds. Each frame contains eight micro-frames and transactions cannot cross a micro-frame boundary. The host controller will discard any transactions that span these micro-frame boundaries.

B. Enhanced Host Controller Interface

The Enhanced Host Controller Interface (EHCI) is the most commonly used specification for a USB 2.0 host controller. EHCI splits transaction types into two classes: (1) *periodic*, which is for isochronous and interrupt transfers, and (2) *asynchronous*, which is for bulk and control transfers.

All transactions that the host controller performs are described in transaction descriptors. Periodic transactions are organized into a periodic frame list that the host controller indexes into using its *frame index register*. The index is incremented every micro-frame. Asynchronous transactions are organized into a round-robin circular linked-list.

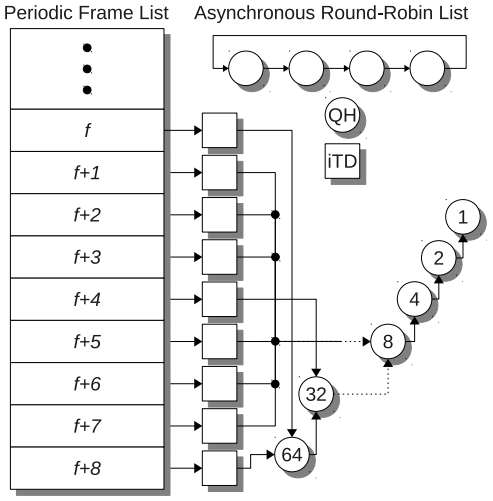

Fig. 3. EHCI Scheduling Overview

Figure 3 shows an overview of the scheduling data structures. The periodic list points either to an *isochronous transaction descriptor* (iTD), for isochronous transactions, or a *queue head* (QH) for interrupt transactions. The value in each QH is the interval of the corresponding endpoint. Both iTDs and QHs contain a pointer that identifies the next transaction to be processed. A single iTD contains the data to perform a maximum of eight transactions (potentially one per micro-frame). The host controller uses the three least significant bits of the frame index register to determine which transaction in the iTD it should execute if the transaction is active. Therefore, multiple iTDs must be

in the periodic list to perform more than eight requests for an isochronous endpoint. However, multiple iTDs for a single endpoint must not be encountered within the same micro-frame.

By comparison, a queue head does not contain all the information necessary to perform a transaction but instead points to a linked-list of *queue element transaction descriptors* (qTDs). A QH and its qTDs represent all transactions that are pending for a given endpoint. Similar to iTDs, QHs in the periodic list contain an 8-bit bitmap that the host controller uses to determine whether it should execute a transaction in the given micro-frame.

Periodic Tree-based Scheduling. In order to meet the requirements of iTDs and QHs, the periodic schedule forms a tree, with the frames of the periodic list acting as leaves. Every micro-frame the host controller picks the appropriate starting leaf and traverses transfer descriptors until it reaches the root. The first elements in the path are iTDs (if any). After all iTDs are traversed the host controller encounters QHs (if any).

The QHs are arranged such that if they are supposed to be reached only every 2^n frames they will be skipped every $2^n - 1$ frames. For example, in Figure 3, frames f and $f + 8$ are leaf nodes for paths that traverse an iTD followed by all the QHs with intervals from 64 to 1 micro-frames. Frame $f + 4$ is a leaf node for a path that includes an iTD followed by all the QHs with intervals from 32 to 1. Likewise, frames $f+1$, $f+2$, $f+3$, $f+5$, $f+6$, and $f+7$ are leaf nodes for paths through all QHs with intervals from 8 to 1. Note that for brevity we have omitted a queue head with interval 16.

At the root of the tree in Figure 3 are all the QHs that have an interval of 8 or less. Such QHs must be reachable from every index of the periodic list. After traversing the periodic list, the host controller uses the remaining time in the micro-frame to process the asynchronous list from its previous position.

Another example periodic schedule with iTDs and QHs for different endpoints is shown in Figure 4. The numbers in the iTDs and QHs are the frame numbers in the periodic list when the corresponding transfer descriptors are accessed by the host controller. Isochronous endpoint A has an interval of eight or less micro-frames and therefore one of the iTDs for it must be visited every frame. Isochronous endpoint B has an interval of sixteen micro-frames, so iTDs for it are visited every other frame. Similarly, interrupt endpoint C has an interval of sixteen micro-frames, so its QH is positioned to only be reached every other frame. Finally, interrupt endpoint D has an interval of eight or less micro-frames, so its QH is reached every frame. The traversal ordering is outlined in Table I.

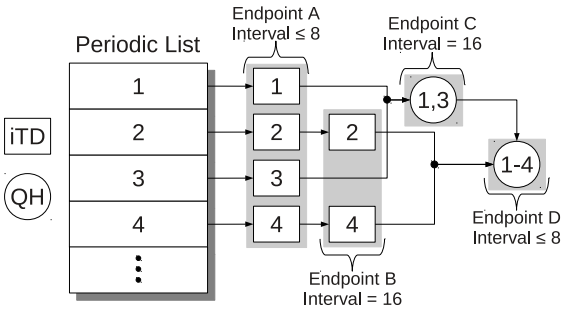

Fig. 4. Periodic Schedule Example

Frame	Endpoints Visited
1	A, C, D
2	A, B, D
3	A, C, D
4	A, B, D

TABLE I
ORDER ENDPOINTS ARE VISITED DURING TRAVERSAL

III. USB SCHEDULING PROBLEM

USB scheduling does not allow transmissions to cross micro-frame boundaries. This is a restriction placed on the USB host controller by the specification [12]. This limitation causes USB scheduling to look similar to the traditional bin-packing problem. In what follows, we define the scheduling problem over a set of tuples S such that:

$$S = \{(w_1, t_1), (w_2, t_2), \ldots, (w_n, t_n)\},$$

where w_i is the time it takes to send transaction i and t_i is the interval of transaction i (either in micro-frames or frames). For high-speed endpoints, the value of t_i is restricted to a power of two in the range $[2^0, 2^{10}]$. Each value, w_i, is measured in nanoseconds and can take any of the following forms [12]:

$$(c + p + (2.083 * \lfloor 3.167 + 56/6 * b \rfloor)) * k$$

- c is the overhead uniquely associated with a given host controller. For our implementation we assume this is five nanoseconds, which is the same in Linux.
- p is the protocol overhead to send a packet. For isochronous transactions, this is 638.232 nanoseconds. For all other transactions this is 916.52 nanoseconds. Most of the difference between the two is the extra overhead to ensure data integrity.
- b is the number of bytes in the packet. While not an explicit requirement of the USB 2.0 specifications this value is typically a power of 2. For high-speed bulk transactions this value must be 512. For interrupt and isochronous endpoints this value can range from 0 to 1024.

- k is either 1, 2, or 3 representing the number of packets in a single transaction.
- The constants 2.083, 3.167 and $56/6$ are given by the USB 2.0 specification [12]. These account for various transmission costs including bit stuffing.

As stated above, USB transactions cannot cross 125μs micro-frame boundaries. Depending on whether we are providing real-time guarantees for bulk transactions we either have 100 or 125 micro-seconds for scheduling real-time transactions.

The interval, t_i, specifies the rate to exchange data between a device and a host. For high-speed devices, intervals are specified in micro-frames[1]. Therefore, a device endpoint that has an interval of one micro-frame has no options for scheduling. A device with an interval of two or more micro-frames has multiple options for when it is scheduled. For example, a device with an interval of two can be scheduled in either odd or even micro-frames. If the device is scheduled in micro-frame f it must also be scheduled in micro-frame $f{+}2i \mid i{\in}\{1, 2, \dots\}$. Consequently, the number of options we have to schedule a single USB device's transactions is equal to the endpoint interval. Additionally, the number of unique scheduling options for a set of n endpoint tuples, $\{(w_1, t_1), (w_2, t_2), \dots, (w_n, t_n)\}$, is $\prod_{i=1}^{n} t_i$.

The USB scheduling problem has bounded execution time, due to the fact there is a maximum number of possible transactions that are schedulable. The maximum transactions is constrained by the product of the number of zero-length packets, z, schedulable in a single micro-frame, and the maximum interval value, 1024. Let the value $M = 1024z$. All scheduling problems that present more than M transactions can be rejected in constant time as not schedulable.

The question remains: "if the size of S is less than or equal to M is there an efficient algorithm that can determine if S is schedulable and, if so, what is the assignment of transactions to micro-frames?" As the set of possible transaction times is limited to a finite number, the problem is similar to restricted bin packing, which has a well known dynamic programming solution [13]. However, due to the added restriction that transactions must be spaced apart by their interval value, the USB scheduling problem does not trivially map to the restricted bin-packing problem.

Our approach to finding an assignment of USB transactions is to use a heuristic that parallels the first-fit decreasing algorithm for bin-packing and rate monotonic scheduling [14]. The algorithm first sorts transactions by interval from smallest to largest, breaking ties by sorting transmission delay from largest to smallest. The transactions are then inserted into the first available micro-frame according to their interval constraints. The algorithm for this approach is the following:

[1]Unless stated otherwise we assume all intervals are in micro-frames. However, they could be in frames for low- and full-speed devices.

Algorithm 1 Quest USB Scheduling Algorithm

```
R ← array of n USB requests
A ← array for the scheduling assignments
T ← 1024 element array initially all zero
// T[f] = time used in micro-frame f
B ← 100000 (in nano-seconds ∼ allows time for bulk transfers)

// Sort transactions by increasing interval,
// breaking ties by largest transmission delay first
R ← SORT(R)
for i = 0 to n − 1 do
   w_i ← TRANSMISSION_DELAY(R[i])
   t_i ← INTERVAL(R[i])
   A[i] ← −1
   j ← 0
   while A[i] = −1 ∧ j < t_i do
      feasible ← TRUE
      f ← j
      while f < 1024 do
         if T[f] + w_i > B then
            feasible ← FALSE
         end if
         f ← f + t_i
      end while
      if feasible then
         A[i] ← j // Request i starts in micro-frame j
         f ← j
         while f < 1024 do
            T[f] ← T[f] + w_i
            f ← f + t_i
         end while
      end if
      j ← j + 1
   end while
   if A[i] = −1 then
      return FALSE
   end if
end for
return (TRUE, A)
```

If successful, the algorithm returns an n-element array A, such that each $A[i]$ holds the starting micro-frame of request i. As stated earlier, transactions with smaller periods (here, intervals) are ordered first. The tie-breaking policy, sorting from largest to smallest transmission delay, is justified with the following logic: the set of positions in a schedule that are acceptable to higher transmission delay requests is a subset of the positions for lower transmission delay requests. By scheduling lower transmission delay requests first, it is possible that they reduce the number of positions in the schedule for higher transmission delay requests. Conversely, scheduling higher transmission delay requests first does not re-

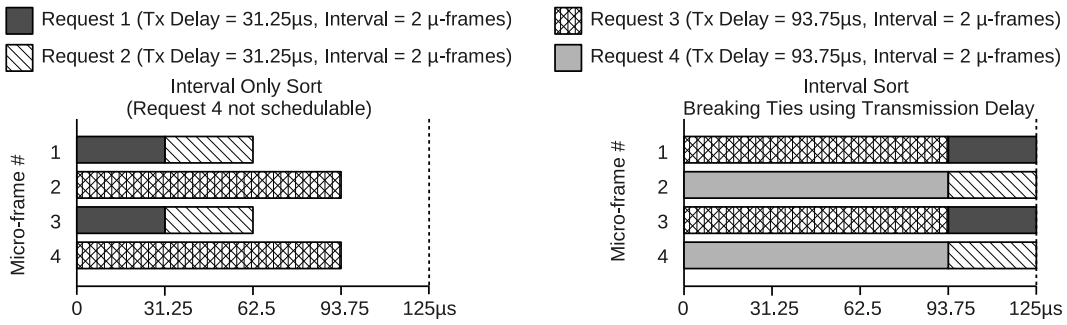

Fig. 5. Interval Only Sort vs. Interval Sort Breaking Ties using Transmission Delay

duce the number of choices in the schedule *only* available to lower transmission delay requests. See Figure 5 for an example.

Figure 6 shows simulation results comparing the Quest USB scheduling method to the Linux first-fit approach, as well as others. Table II outlines each scheduling method used. The simulation iterates over all possible schedulable permutations of five or less USB requests, and reports the success rate of each algorithm. In the simulation we limited possible interval values to 2, 4, 8 or 16, and possible bytes per packet to powers of two in the range $[32, 1024]$. Quest failed to schedule 149,600 of the 62,287,898,048 possible USB requests combinations. By comparison, Linux failed to schedule 95,364,176 USB request combinations, which is a factor of 637 worse.

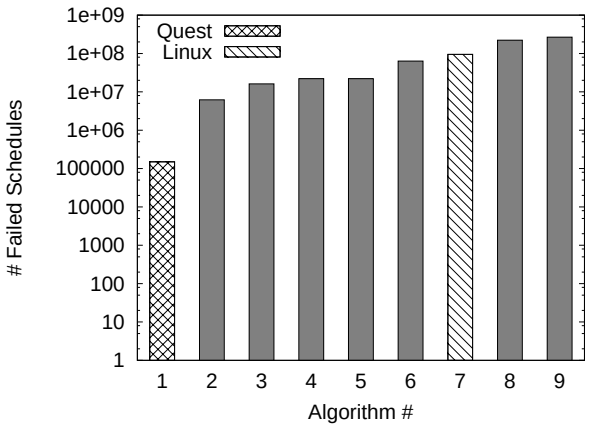

Fig. 6. Scheduling simulation results

Algorithm #	Description
1	Sort by increasing interval breaking ties with a sort by decreasing transmission delay then first-fit (Quest)
2	Sort by increasing interval then first-fit
3	Sort by increasing interval breaking ties with a sort by increasing transmission delay then first-fit
4	Sort by increasing transmission delay $\times$ interval then first-fit
5	Sort by decreasing transmission delay $\times$ interval then first-fit
6	Sort by decreasing transmission delay then first-fit
7	first-fit (Linux)
8	Sort by increasing transmission delay then first-fit
9	Sort by decreasing interval then first-fit

TABLE II
A COMPARISON OF HEURISTIC SCHEDULING METHODS.

IV. REAL-TIME USB ARCHITECTURE

The Quest USB sub-system has been implemented from scratch, in accordance with the USB 2.0 and EHCI specifications. It allows isochronous and interrupt endpoints to specify their throughput requirements which, if accepted, are guaranteed for the lifetime of the request. This differs from the approach in Linux where bandwidth is only guaranteed up to the transfer length of a USB request block (URB). In Linux, a device driver must explicitly reschedule the URB when the request is finished. This is typically done in a completion callback function provided by the URB, which runs in the context of an interrupt handler. The Linux approach allows race conditions to occur that will create gaps in the periodic list schedule. Also, if a request is not resubmitted immediately after it has completed the bandwidth can be allocated to another endpoint. In Quest, bandwidth is pre-allocated for iTDs and QHs throughout the entire periodic list schedule, respecting their interval requirements, thereby avoiding race conditions.

In addition to providing bandwidth guarantees to devices, the Quest USB sub-system also reorders active transactions to increase overall bandwidth usage. That is, transactions are reordered while maintaining endpoint constraints, to allow the admission of new requests that would otherwise be rejected, as described earlier.

Transaction reordering does not alter the position of

QHs in the schedule. Only isochronous transactions are reordered if possible. First, a tentative schedule is formed according to the sorting criteria explained in Algorithm 1. If a new transaction can be accommodated in the tentative schedule, the schedule is committed as active. Only iTDs requiring a different frame assignment are moved. This causes each reassigned isochronous device to violate its interval value for at most one transaction. For example, Figure 7 shows three requests scheduled in micro-frames 1-4. Request 4 arrives between micro-frames 4 and 5, causing Request 2 to be reassigned to even micro-frames. This leads to the periodic request skipping a micro-frame at time 5, but it resumes its correct interval spacing for subsequent micro-frames (at 6, 8, etc).

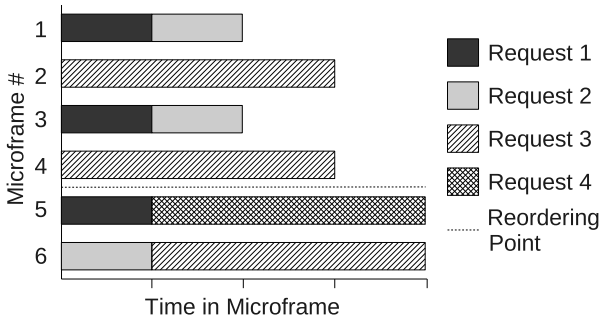

Fig. 7. Schedule Reordering Example

The Quest USB sub-system only reorders isochronous transactions that are far enough ahead of the current frame index register. This is to avoid updates to entries that are being buffered by the host controller itself. It is possible to do this with iTDs but more difficult with QHs, since there is only a single QH for each interrupt transaction. If we were to reorder both iTDs and QHs there would be a brief period of time where the QHs were following the new schedule but the iTDs were following the old schedule.

The USB 2.0 specification [12] states that high-speed periodic transactions can use at most 80% of a micro-frame, leaving at least 20% for asynchronous transactions. However, along with providing real-time guarantees to isochronous and interrupt transfers, the Quest USB sub-system can provide real-time guarantees for bulk transactions. To accomplish this, bulk transactions are associated with an interval value similar to that with isochronous and interrupt endpoints. This interval value is not provided by the endpoint but by the device driver when it submits the real-time request to the USB core.

Bulk transactions are organized in a round-robin list, as described in Section II, and they must occur at the same rate. Therefore, in Quest, the minimum interval among all bulk transactions is used to determine their scheduling order. With this approach, bulk transactions are scheduled in the same way as interrupt and isochronous transactions, using the same scheduling algorithm.

The Quest USB sub-system can run in one of two modes. The first mode provides real-time guarantees for bulk transactions, while still ensuring that periodic transactions do not exceed 80% of any micro-frame. In the second mode, real-time guarantees are not ensured for bulk transactions but collectively they receive at least 20% of every micro-frame. Either way, a mixture of both real-time and best-effort transactions can be supported.

V. Experimental Evaluation

In this section, we evaluate our Quest USB sub-system and compare it to Linux, where appropriate. The host machine for both Quest and Linux was a 3.10 GHz Intel® Core i3-2100 CPU. For Linux, we used Ubuntu 10.04 with kernel version 2.6.32. The USB 2.0 EHCI chip was an Intel® Corporation cougar point USB enhanced host controller (rev 05).

Beagleboards [5] shown in Figure 8 were also used in various experiments. These feature USB “on-the-go” (OTG) device capabilities. The boards themselves ran Ångström Linux [15], kernel version 2.6.34. We used the Linux gadget driver interface to create a custom USB device. This allowed us to measure throughput more effectively as the Beagleboards were programmed to always have data and space available for requests. We wrote a device driver for the Beagleboards in Linux and Quest that performed the equivalent functionality on both operating systems. Each Beagleboard has 9x512-byte input- and output-endpoints that can be programmed to be any endpoint type. In comparisons with Linux, the Quest real-time asynchronous mode was disabled, unless stated otherwise. This is because Linux does not provide this feature.

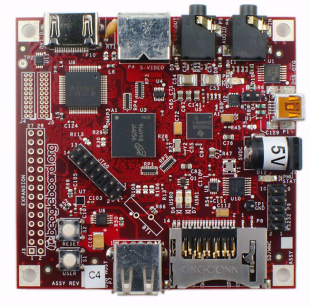
Fig. 8. Beagleboard

A. Maximum Throughput

While the goal of the Quest USB sub-system is to provide real-time guarantees, the design of the system should not be detrimental to performance. To verify this, we tested the maximum throughput of both Quest and Linux. We used three Beagleboards with the same configuration: three isochronous input endpoints, each having an interval of one micro-frame, and three bulk input endpoints, all with a packet size of 512-bytes. Figure 9 shows the aggregated throughput of Quest and Linux. The experiment shows that Quest and Linux are comparable in maximum throughput, when all periodic endpoints have the same interval.

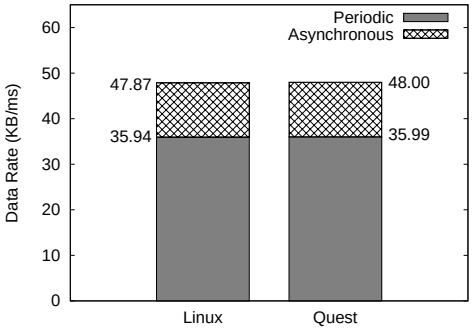

Fig. 9. Maximum Throughput

B. Scheduling Vulnerability

As previously discussed in Section IV, the total throughput of a USB schedule with Linux is dependent on the order of opening (or processing) endpoint requests. Different schedules are possible when periodic devices have an interval greater than one micro-frame. To demonstrate this issue, we opened the same endpoints with different orderings in both Quest and Linux. We used two Beagleboards that had a total of four isochronous input-endpoints. Each endpoint had an interval of two micro-frames. Additionally, there were a total of seven isochronous input-endpoints each with an interval of one micro-frame. The packet size for all the endpoints was again 512-bytes.

In the first ordering, the four isochronous endpoints each with intervals of two micro-frames were opened first, followed by the seven endpoints each with intervals of one micro-frame. In the second ordering, the seven endpoints with an interval of one micro-frame were opened first, followed by the four endpoints with an interval of two micro-frames. Both Linux and Quest allow a maximum of nine periodic transactions of 512-bytes per micro-frame. In the first ordering, Linux assigns all isochronous endpoints with an interval of two to the same micro-frames and therefore cannot open the last two isochronous endpoints with an interval of one. In Quest, the four endpoints with an interval of two were also assigned to the same micro-frames. However, when the sub-system is presented with the tenth endpoint it reorders all the transactions in order to accommodate this request. Linux fails to open the last two isochronous endpoints because it is unable to reorder the four requests each with an interval of two, which have all been assigned to the same micro-frames. This is shown by the missing bar in Figure 10. This is not a problem for either Linux or Quest in the second scheduling experiment.

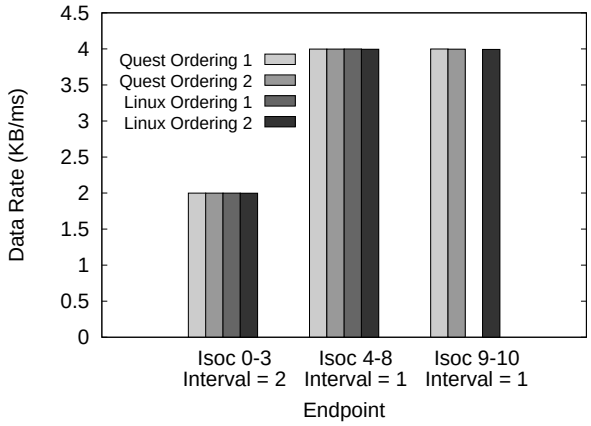

Fig. 10. Scheduling Vulnerability

C. Real-Time Asynchronous Transactions

To demonstrate that Quest is able to provide real-time guarantees for asynchronous transactions, we conducted an experiment involving two Beagleboards. Each beagleboard was configured with seven bulk, four isochronous, and three interrupt endpoints, each with an interval of one micro-frame. All transaction sizes were 512-bytes. The results are shown in Figure 11. As expected, Linux allows all transactions to occur. Quest, however, does not allow the three interrupt transactions to pass the admission control, because they would result in the bulk transactions falling below a data rate of 512-bytes per micro-frame.

D. Bandwidth Preservation

To show how Quest provides bandwidth preservation for all open endpoints, even when the endpoint is not currently in use, we conducted the following experiment: 3 Beagleboards were connected to the host as shown in Figure 12. A token was passed between the two Beagleboards while an attempt to open seven isochronous endpoints on the third Beagleboard took place. As stated earlier in the paper, Linux only provides bandwidth preservation for the output-endpoints when they are

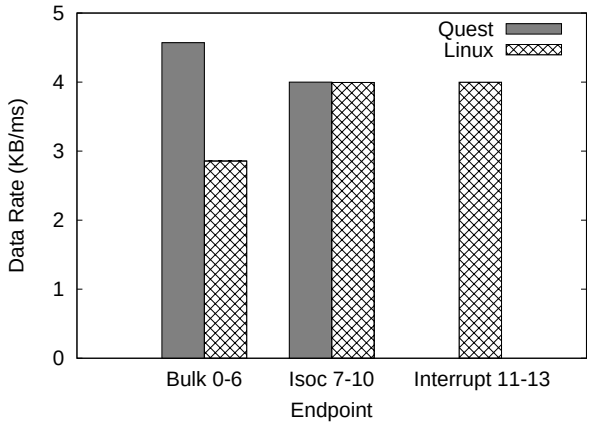

Fig. 11. Real-Time Asynchronous Transactions

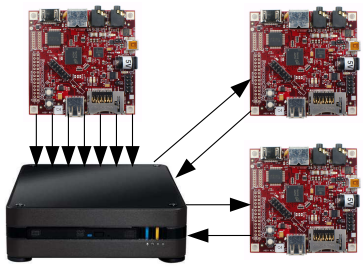

Fig. 12. Bandwidth Preservation Setup

actively used. Since at most one output-endpoint is actively used at any given time, the bandwidth preservation is intermittent. This results in all seven isochronous endpoints of the third Beagleboard being opened and the token being blocked from further transmission by the host. In Quest, because bandwidth is allocated for open endpoints, and remains allocated even if the endpoint is not currently in use, two of the seven isochronous endpoints fail to open. However, the token continues to be passed between Beagleboards. Table III shows the results of the experiment.

VI. Related Work

Real-time bus communication has been studied in both a theoretical framework [16]–[18] and in various physical implementations [19]–[23]. In this section, we provide an overview of both the theoretical aspects of real-time buses along with actual implementations.

OS	Tokens Passed Without Third Beagleboard	Tokens Passed With Third Beagleboard	Third Beagleboard Isoc Endpoints Opened
Quest	1499	1499	5
Linux	1499	300	7

TABLE III
Results of bandwidth preservation experiment

Lehoczky et al [16] modeled real-time bus scheduling as a CPU scheduling problem, with several notable differences relating to preemption, buffering and priority granularity. Our method of ordering periodic requests is similar to rate-monotonic task scheduling, with rules for breaking ties. While Lehoczky et al address the real-time scheduling of periodic messages transmitted on a multi-master bus, this does not apply to USB, which is a master-slave bus protocol.

The Controller Area Network (CAN) [9] bus protocol is a commonly used protocol and has been heavily studied. Tindell et al [23] studied the worst case transmission delay due to blocking of higher priority tasks taking jitter into account in their analysis. More recently, Davis et al [24] provide a revised study of CAN, correcting for earlier analytical flaws. USB differs from CAN because CAN is a multi-master bus protocol where each node is capable of initiating a transmission. CAN differs from protocols such as Ethernet by determining which message has the highest priority during a negotiation phase at the start of each transmission.

Zuberi and Shin [20] address the utilization problem of CAN-bus networks. They introduce the mixed traffic scheduler (MTS), which is a hierarchical scheduler using both Earliest Deadline First (EDF) and Deadline Monotonic Scheduling (DMS). The authors state that pure Earliest Deadline is not reasonable due to the limited number of bits for priority levels in the CAN-bus protocol. Their solution is to discretize time into regions and first use EDF to prioritize tasks with deadlines in different time regions. DMS is then used to prioritize tasks that fall within the same time region.

Davis et al [19] provide a scheduling analysis of CAN assuming that messages within a node are scheduled using a FIFO queue instead of priority queue. While previous scheduling analyses for CAN are based on the assumption that the highest priority message submitted for transmission is the next message a node transmits, in practice many CAN device drivers use a FIFO queue.

Huang et al [25], [26] attempt to provide QoS guarantees for USB 1.1 and 2.0. To do this, they modify endpoint descriptors within the host controller driver. Effectively, all endpoints are treated as an *equivalent* endpoint with the same throughput but with the smallest possible interval given the endpoint speed. For full- and low-speed devices, the smallest interval is one frame. For high-speed devices, the smallest interval is one micro-frame. As an example, Huang et al treat a high-speed endpoint with a packet size of 512-bytes and interval of two micro-frames as an equivalent endpoint with a packet size of 256-bytes and interval one micro-frame.

Admission control is then performed on the modified endpoints. This is to ensure the total utilization is below maximum capacity. To reduce overhead, an attempt is made to reinstate the original endpoint intervals and packet sizes.

This endpoint modification violates the USB specifications as the endpoint interval rate is not respected. Similarly, polling at a higher rate is not guaranteed to work for all devices. Our USB sub-system attempts to guarantee QoS for USB 2.0 transactions without endpoint modification. Our USB sub-system does, however, use similar sorting strategies to those defined by Huang et al. They also provide a probabilistic guarantee for asynchronous transactions, while we provide explicit bandwidth partitioning for asynchronous transactions.

VII. Conclusions and Future Work

This paper describes the real-time USB framework in the Quest operating system. This is instrumental in the development of real-time embedded applications that require interaction with the physical world through sensors and actuators. With USB OTG functionality and with custom FPGA-based USB routers, a real-time USB sub-system is also capable of being the backbone of a real-time communication interconnect between compute nodes. We have demonstrated a host USB sub-system that provides bandwidth partitioning, dynamic reordering of transactions to avoid unnecessary admission control rejections, and real-time guarantees for asynchronous transactions. We showed how our approach to the USB scheduling problem results in a higher data throughput than the approach found in Linux.

Future work will involve the development of a real-time USB OTG sub-system to act as the device counterpart to the real-time USB host sub-system described in this paper. This will allow the creation of star, tree and ring topology USB networks. We will also investigate the creation of custom hardware that is capable of acting as a USB router. Such a device would allow multiple computer hosts to interconnect with each other without the need for USB OTG. It would also allow for more complex topologies than those allowed by USB OTG.

While current embedded single board computers are limited to USB 2.0 it is reasonable to assume that USB 3.0 will eventually begin to appear on such systems. We plan to investigate how we can apply the techniques described in this paper to USB 3.0 host controllers.

References

[1] "Quest." http://www.cs.bu.edu/~richwest/quest.html.

[2] "TIA-232-F: Interface between data terminal equipment and data circuit-terminating equipment employing serial binary data interchange," 1997.

[3] *M68HC11 Reference Manual*, 6.1 ed., 2007.

[4] *The I^2C–Bus Specification*, 2.1 ed., January 2000.

[5] "Beagleboard.org." http://beagleboard.org, October 2012.

[6] "Raspberry Pi." http://www.raspberrypi.org, October 2012.

[7] Intel, "Thunderbolt technology: Technology brief," 2012.

[8] "Universal Serial Bus 3.0 specification," June 2011.

[9] "Road vehicles – Controller area network (CAN)," 2009. ISO 11898.

[10] FlexRay Consortium, *FlexRay Communications System Protocol Specification*, 3.0.1 ed., October 2010.

[11] *Enhanced Host Controller Interface Specification for Universal Serial Bus*, 1.0 ed., March 2002.

[12] *Universal Serial Bus Specification*, 2.0 ed., April 2000.

[13] R. Motwani, "Lecture notes on approximation algorithms - Volume I," Tech. Rep. CS-TR-92-1435, Stanford University, 1992.

[14] L. Sha, R. Rajkumar, and S. Sathaye, "Generalized rate-monotonic scheduling theory: a framework for developing real-time systems," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 68–82, 1994.

[15] "The Ångström distribution." http://www.angstrom-distribution.org, October 2012.

[16] J. P. Lehoczky and L. Sha, "Performance of real-time bus scheduling algorithms," in *Proceedings of the 1986 ACM SIGMETRICS joint international conference on Computer performance modelling, measurement and evaluation*, SIGMETRICS '86/PERFORMANCE '86, (New York, NY, USA), pp. 44–53, ACM, 1986.

[17] K. A. Kettler, J. P. Lehoczky, and J. K. Strosnider, "Modeling bus scheduling policies for real-time systems," in *Proceedings of the 16th IEEE Real-Time Systems Symposium*, RTSS '95, (Washington, DC, USA), IEEE Computer Society, 1995.

[18] C. M. Aras, J. F. Kurose, D. S. Reeves, and H. Schulzrinne, "Real-time communication in packet-switched networks," in *PROC. IEEE*, pp. 122–139, 1994.

[19] R. I. Davis, S. Kollmann, V. Pollex, and F. Slomka, "Controller area network (CAN) schedulability analysis with FIFO queues," in *Proceedings of the 2011 23rd Euromicro Conference on Real-Time Systems*, ECRTS '11, (Washington, DC, USA), pp. 45–56, IEEE Computer Society, 2011.

[20] K. M. Zuberi and K. G. Shin, "Non-preemptive scheduling of messages on controller area network for real-time control applications," in *Proceedings of the Real-Time Technology and Applications Symposium*, RTAS '95, (Washington, DC, USA), IEEE Computer Society, 1995.

[21] H. Kopetz and G. Grünsteidl, "TTP-A protocol for fault-tolerant real-time systems," *Computer*, vol. 27, pp. 14–23, Jan. 1994.

[22] J. Loeser and H. Härtig, "Low-latency hard real-time communication over switched Ethernet," in *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, ECRTS '04, (Washington, DC, USA), pp. 13–22, IEEE Computer Society, 2004.

[23] K. Tindell, H. Hanssmon, and A. J. Wellings, "Analysing real-time communications: Controller area network (CAN)," in *Proceedings of the 15th IEEE Real-Time Systems Symposium (RTSS 94), San Juan, Puerto Rico, December 7-9, 1994*, pp. 259–263, IEEE Computer Society, 1994.

[24] R. Davis, A. Burns, R. Bril, and J. Lukkien, "Controller area network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, 2007.

[25] C. Y. Huang, L. P. Chang, and T. W. Kuo, "A cyclic-executive-based QoS guarantee over USB," in *Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '03, (Washington, DC, USA), IEEE Computer Society, 2003.

[26] C. Y. Huang, T. W. Kuo, and A. C. Pang, "QoS support for USB 2.0 periodic and sporadic device requests," in *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, RTSS '04, (Washington, DC, USA), pp. 395–404, IEEE Computer Society, 2004.