# USB Interrupt Differentiated Service for Bandwidth and Delay-Constrained Input/Output

Zhiyuan Ruan, Anton Njavro, Richard West
*Department of Computer Science*
*Boston University, Boston, MA*
{zruan, njavro, richwest}@bu.edu

***Abstract*—Device interrupts have long been a problem in real-time systems. Handling an interrupt in the context of a critical task potentially leads to a missed deadline. While modern systems have proposed techniques to schedule interrupts, the challenge is to determine the correct priority based on the task that triggers their occurrence. One approach is to assign an interrupt with the highest priority among all tasks waiting on a corresponding device. However, this does not fully eliminate priority inversion, as the interrupt may be handled at a much higher priority than the task with which it is associated.**

**To solve this problem, we present a solution for devices connected to a Universal Serial Bus (USB). USB has properties that make it potentially suitable for tackling high bandwidth sensor data processing and low-latency input/output (I/O) control. We describe the implementation of an xHCI (USB 3.x) driver in the Quest RTOS, which guarantees throughput and delay requirements for USB devices and their I/O requests using time-budgeted interrupt handling servers. Our paper is the first to introduce differentiated USB interrupt servicing. Experiments show how our approach outperforms a Linux xHCI driver running on a `PREEMPT_RT`-patched system with `SCHED_DEADLINE` tasks. Differentiated USB interrupt handling is shown to improve the performance of Quest's default xHCI driver, which is purposely designed to provide real-time I/O guarantees.**

***Index Terms*—Universal Serial Bus (USB), Message Signaled Interrupts (MSIs), Differentiated Service, Real-Time I/O**

## I. Introduction

Embedded single board computers connect with input/output (I/O) devices using relatively low-bandwidth interfaces based on RS232 [1], Serial Peripheral Interface (SPI) [2], $I^2C$ [3], or Controller Area Network (CAN) [4] bus protocols, among others. However, new classes of real-time and embedded systems are emerging with the need to process high-bandwidth data from sensors such as cameras, LIDARs and RADARs. Examples include those in the automotive domain, where an array of cameras might work together with a LIDAR or RADAR unit to provide sensory inputs for a semi- or full-autonomous advanced driving assistance system (ADAS). Various standards are being developed to support host connectivity of high-bandwidth devices, including GSML [5] and Ethernet MAC-PHY chips [6]. However, none of these are as prevalent as Universal Serial Bus (USB), which is increasingly popular on many industrial grade board computers.

USB allows multiple different devices to be attached to a host via a shared bus instance managed by a host controller. The host controller generates interrupts on completion of device transfers at a moderated rate. USB has the bandwidth to handle modern sensor devices such as cameras, while meeting the latency requirements of many control buses. For example, USB 3.2 is capable of reaching bus rates of 20 Gbps, with USB 4.0 rising to 40 Gbps, and USB/Thunderbolt 5.0 expected to reach as high as 120 Gbps. At the same time, transfer requests are schedulable in microframes of 125 microseconds, making USB potentially suitable for latency-sensitive transfers typically associated with CAN buses.

While USB appears to be a viable bus technology for handling I/O requests for both low- and high-bandwidth devices, with relatively low latency bounds, it poses several challenges. First, the host controller driver needs to be able to schedule I/O requests according to real-time guarantees, but evidence suggests that most such drivers are not sufficiently predictable to meet real-time requirements in timing-critical domains [7]–[9]. Second, many devices now connect to hosts with different service-level guarantees, dictated by task criticality levels.

To understand the importance of mixed-criticality systems [10] and real-time I/O, consider the direction being taken by automotive systems. These systems are tending towards centralized or zonal architectures [11], which replace separate electronic control units (ECUs) with either a centralized main computer or several domain controllers managing multiple vehicle functions. These functions include chassis, body, powertrain, infotainment and ADAS services, and increasingly are being consolidated onto one or several computers where they run as software tasks. Consequently, we are now seeing the emergence of *software-defined vehicles* (SDVs) [12]–[16] comprising multiple functions of different criticality levels on the same host machine. These host machines feature powerful multicore CPUs, capable of handling hundreds of software tasks, each interacting with the physical environment through a network of sensors and actuators. It becomes critical to ensure real-time information exchange between sensors and actuators connected to the same host machine.

The problem addressed by this paper is to consider how USB is able to manage the real-time exchange of information between host tasks and devices such as those connected to traditional CAN bus networks or high-bandwidth sensors. In tackling this problem we consider the importance of meeting throughput and delay constraints associated with tasks of different criticality levels.

While prior work has investigated the use of USB for real-time I/O [7]–[9], [17], [18], we are unaware of any reports

of how to ensure *differentiated* service guarantees for I/O requests, beyond the basic endpoint service-level agreements supported by a device. While a USB request might select a particular transfer requirement based on the endpoint capabilities (e.g., number of bytes to transfer over a window of time), no prior work has looked at how to differentiate the handling of interrupts from USB devices according to their service requirements.

Significantly, USB has the ability to associate specific interrupts with different service requests, by including an *interrupter* value in the request. This is similar to *early demultiplexing* [19], [20], which associates each interrupt with the task that led to its occurrence [21]–[23]. However, most approaches either assign a highest priority among a set of waiting tasks to the handling of interrupts, or they require software assistance to determine the match of an interrupt with the task that caused it to occur. Here, we investigate USB capabilities to support differentiated services.

In this paper, we present the following contributions: (1) we describe the design and implementation of a USB differentiated services framework, which uses multiple USB interrupters to associate different I/O requests with specific interrupt handlers; (2) we show how to guarantee service-level constraints on I/O requests by scheduling USB interrupts along with tasks that depend on them; (3) we compare our approach in the Quest RTOS [24] against a Linux system that features one USB interrupter for all I/O requests. This work is the first, to our knowledge, to implement multiple USB interrupters into a real-time scheduling framework. Experimental results show how our approach is able to guarantee differentiated services support while prior solutions, including Linux, are not.

The next section provides background to the problem addressed in this work. It includes pertinent details about USB, interrupters and how interrupts are typically handled in modern operating systems. This is followed by technical details behind our USB differentiated services approach in Section III . Section IV describes the evaluation of USB differentiated services in comparison to prior approaches, including Linux. Related work is discussed in Section V, followed by conclusions and future work in Section VI .

## II. Background

One of the main challenges for real-time USB transfers is the timely handling of interrupts when such transfers are completed. Simply preempting the currently executing task to handle an interrupt is often unacceptable. Systems such as Linux charge the preempted task for time spent handling interrupts, even though the task is not actually making progress during that time, potentially causing it to miss an important deadline. What is needed is an approach that correctly accounts and charges interrupt processing time to an entity associated with the task that initiated the USB request.

In Linux, the interrupt handling is divided into a top and bottom half. The top half acknowledges the interrupt as soon as it is triggered, and then defers the bulk of the handler to a bottom half. Normally, the bottom half is executed as soon as the top half of a handler finishes, with interrupts enabled. If a bottom half is preempted by interrupts and restarted *MAX_SOFTIRQ_RESTART* times, it defers execution to a per-core *ksoftirqd* thread. *MAX_SOFTIRQ_RESTART* defaults to 10 to achieve a trade-off between low-latency interrupt handling and CPU time needed by preempted tasks. Each ksoftirqd thread is scheduled using the SCHED_FIFO class, to handle deferred interrupts.

As others have noted [21], [22], the Linux interrupt-handling approach leads to a mismatch between the priority of the task requesting I/O and the priority associated with the interrupt handler on I/O completion. To begin, interrupt bottom halves take highest priority until they have interfered with task processing too many times, which leads to their demotion to an entirely different priority. It makes sense in a real-time system to manage interrupts at the priority of the entity, either a kernel or application task, that generated the request.

In our case, we wish to differentiate between USB requests, so that host controller interrupts are matched to the correct bottom half processing priority. In the rest of the section, we will discuss how eXtensible Host Controller Interface (xHCI) interrupters and Message Signaled Interrupts (MSIs) provide the basis for correct prioritization of interrupts resulting from USB requests.

### A. USB & xHCI

Universal Serial Bus (USB) is a master-slave protocol that allows multiple devices to be connected to a host computer. All USB transfers are initiated by the host acting as the master. Different bus speeds include low, full, high, superspeed, and superspeed+ versions 3.1 and 3.2, supporting device throughputs of 1.5Mbps, 12Mbps, 480Mbps, 5Gbps, 10Gbps and 20Gbps, respectively.

A USB device is defined by a set of descriptors that are readable by the host. The descriptors correspond to the *device*, its *configurations*, *interfaces* and supported *endpoints*. A device has one device descriptor, with at least one configuration. A configuration contains at least one interface, which defines zero or more endpoints.

A device descriptor contains general information, such as the USB version, vendor and product IDs, the device class, and the number of supported configurations. A configuration descriptor includes information such as the maximum bus power consumed by the device, and the number of interfaces supported by the current configuration. Only one configuration is active at a time. An interface descriptor serves as the header for a defined number of endpoints, used for communication with the host. Each endpoint descriptor then defines the type and direction of transfer, polling interval, and maximum packet size of the endpoint. There are four endpoint transfer types: (1) *Control*, for lossless transmission of device configuration data, (2) *Bulk*, for lossless transmission of non-real-time data, (3) *Interrupt*, for lossless real-time data, and (4) *Isochronous*, for loss-tolerant real-time data.

All USB transfers are managed by the host controller, which has undergone four standards, at the time of writing: the Open

Host Controller Interface (OHCI), Universal Host Controller Interface (UHCI), Enhanced Host Controller Interface (EHCI), and Extensible Host Controller Interface(xHCI). These specifications are intended to work with USB 1.1, USB 1.x, USB 2.0, and USB 3.x respectively. All specifications are backward compatible with earlier standards.

xHCI defines three types of ring buffer, for communication between USB devices, host and host controller: *Command Ring*, *Event Ring* and *Transfer Ring*. A single Command Ring per extensible host controller (xHC) instance is used to pass commands to the xHC. An xHC supports up to 1024 Event Rings. Each Event Ring is used by the xHC to pass various event notifications to the host, such as transfer completion. Event rings are managed by *interrupters* (discussed in Section II-B). One Transfer Ring per USB endpoint is used by the host to exchange data with a device.

### B. Interrupters & PCI Interrupts

Traditionally, a device asserts a signal on a physical pin to trigger an interrupt to the host. Legacy x86-based platforms use up to two cascaded 8259 Programmable Interrupt Controller (PIC) chips, which support up to 15 device interrupts, each with fixed priority.

The 8259 PIC's limited number of interrupts and lack of support for symmetric multiprocessing (SMP) led to it being replaced by the Advanced Programmable Interrupt Controller (APIC). The APIC consists of one Local APIC (LAPIC) per core and one optional IO-APIC that directs external interrupts to specific cores. An IO-APIC supports up to 240 interrupt lines, with each line capable of being assigned an independent interrupt vector. However, most IO-APICs such as Intel's 82093AA [25] only support 24 interrupt lines. When an interrupt line is asserted, the IO-APIC writes the interrupt vector associated with the line into the LAPIC. Then, the LAPIC will deliver the interrupt to its processor core.

The Peripheral Component Interconnect (PCI) bus is commonly found in personal computers (PCs), and provides a means to connect hardware devices to the host computer. PCI is used to deliver interrupts through an IO-APIC. When using an IO-APIC, all functions on a PCI device share four interrupt lines: INTA#, INTB#, INTC# and INTD#. Each interrupt is then mapped to an interrupt number on the CPU by the BIOS. Due to limited interrupt lines provided by the IO-APIC, it is often necessary for a PCI device to share the same interrupt number with other PCI devices. Therefore, the host must query the PCI device to determine the function that caused an interrupt.

PCI revision 2.2 introduced Message Signaled Interrupts (MSIs) to allow devices to bypass an IO-APIC and write directly to a target LAPIC. MSIs allow each device to have up to 32 interrupts without needing a physical interrupt pin. Intel's tests with Linux [26] show that an IO-APIC reduces interrupt delivery latency by a factor of around three compared to a PIC, while MSIs reduce the latency by a factor of around seven relative to a PIC. MSI-X was introduced in PCI 3.0, to allow up to 2048 interrupts per device. Unlike MSI, where all interrupts from one device are directed to the same LAPIC, MSI-X allows each interrupt to target different LAPICs. In this paper, we focus on MSI, as our test platform's xHC does not support MSI-X capabilities.

The xHC must support MSI if it features more than one interrupter. An xHCI supports up to 1024 interrupters, with each interrupter managing events and their notification to the host. Each interrupter consists of an Interrupter Management Register, an Interrupter Moderation Register and an Event Ring. The Interrupter Management Register allows the host to enable and disable individual interrupters. The Interrupter Moderation Register allows the host to moderate the frequency of interrupts generated by an interrupter. Each interrupter will generate an interrupt to the host if it is enabled and there is something in its event ring that requires interrupt handling.

The MSI capability in the xHC's PCI configuration space must be programmed, to establish interrupt vectors and handlers with each interrupter. The MSI capability allows the host to set the total number of interrupts to be enabled. The host sets the vector for interrupter zero, with subsequent vectors being automatically assigned to higher numbered interrupters. For example, if the starting vector is set to be 32, then the first interrupter will generate vector 32, the second interrupter will generate vector 33, and so on. Therefore, with MSI, it is possible to associate a unique interrupt (and corresponding interrupter) with each separate USB device (and corresponding xHC Event Ring).

## III. Technical Details

To support the correct prioritization of interrupts resulting from USB requests, we use the Quest real-time operating system (RTOS) [27] in this work. In Quest, interrupt handling is divided into top and bottom halves, as with Linux. However, a top half will acknowledge an interrupt and schedule a bottom half on Quest's Virtual CPUs (VCPUs) [24]. Each VCPU is given a budget of C, and period of T, time units. There are two types of VCPUs in Quest: Main and I/O VCPUs. Application and system tasks are assigned to Main VCPUs, which are implemented as Sporadic Servers [28]. Interrupt bottom halves are executed on I/O VCPUs, which operate as bandwidth preserving servers with a utilization factor, $U_{IO}$. A task executing on a Main VCPU may issue a blocking I/O request that completes interrupt processing on a specific I/O VCPU, before the task is unblocked and rescheduled on its Main VCPU.

The budget and period of an I/O VCPU are dynamically calculated as a function of $U_{IO}$ and the period of a specific Main VCPU for a task awaiting I/O completion, as described shortly. Significantly, an I/O VCPU has a single replenishment, which is available at a future *eligibility* time. This guarantees its bandwidth utilization never exceeds $U_{IO}$ on the underlying physical CPU (or core). This approach is effective for short-lived interrupt handlers, as it avoids frequent reprogramming of replenishment timers needed to accurately manage budget usage of a Sporadic Server [29].

We establish a baseline system configuration that uses one I/O VCPU for all interrupt bottom half processing associated with Quest's xHCI driver. In the current version of Quest, the top half xHCI interrupt handler sets the period, $T_{IO}$, of its I/O VCPU to be the same as the *smallest period*, $T_{Main}$, of all Main VCPUs currently waiting for USB requests to be completed. As Quest defaults to using Rate-Monotonic Scheduling (RMS) [30] for all VCPUs, a smaller period implies a higher priority. The budget of the I/O VCPU is then set to $U_{IO} \times T_{Main}$, thus preserving bandwidth $U_{IO}$ over the Main VCPU's *period*.

### A. USB Interrupt Differentiated Service

While the above approach leads to an I/O VCPU inheriting the priority of one of the Main VCPUs it serves, it does not entirely eliminate priority inversion. It is still possible for a USB request associated with a low-priority Main VCPU to be handled at the highest priority among all waiting Main VCPUs. To eliminate this problem, we extend Quest's xHCI driver to use one I/O VCPU per USB interrupter, with our test machine having an xHC capability of up to 8 interrupters. Using different interrupters for USB requests ensures the correct I/O VCPU is selected when handling a bottom half. This, in turn, leads to improved differentiated service support for USB requests of different priorities.

To illustrate the importance of differentiated service, let us consider a uni-processor system with three threads, $\tau_H, \tau_M$ and $\tau_L$. Each thread is assigned high, medium and low priority, respectively. Suppose $\tau_H$ issues an I/O request and then suspends itself until an interrupt fires. Suppose also that $\tau_M$ and $\tau_L$ are CPU-bound tasks. Let us assume the system is running Linux with the `PREEMPT_RT` patch. $\tau_H, \tau_M$ and $\tau_L$ are scheduled under `SCHED_DEADLINE`. However, the interrupt bottom half is executed under `SCHED_FIFO` with a priority of 50, which is a lower priority class than the `SCHED_DEADLINE` tasks.

Consider $\tau_H, \tau_M$ and $\tau_L$ have total execution times of two, three and one time units, respectively, with top and bottom half interrupt handling taking one time unit in each case, per I/O request. As shown in Figure 1, with Linux, $\tau_H$ misses its deadline (at t=6) because the bottom half for $\tau_H$ is delayed by $\tau_M$ and $\tau_L$. The same setup with Quest, shown in Figure 2, shows no deadline misses as the bottom half inherits the priority of $\tau_H$.

Now let us reconsider the setup with the same number of threads. This time, both $\tau_H$ and $\tau_L$ issue I/O requests after 1 time unit, while $\tau_M$ remains a CPU-bound task. As shown in Figure 3 with adjusted ready times, Linux still leads to deadline misses. Quest also leads to deadline misses as shown in Figure 4, because the high priority bottom half handles the low priority task's request and thus delays $\tau_M$ and $\tau_H$.

With our USB differentiated service framework, individual bottom halves will handle requests with matching priorities. As shown in Figure 5, two bottom halves are created – one for $\tau_H$ and another one for $\tau_L$. Because bottom halves execute with correct priority, there are no deadline misses using Quest.

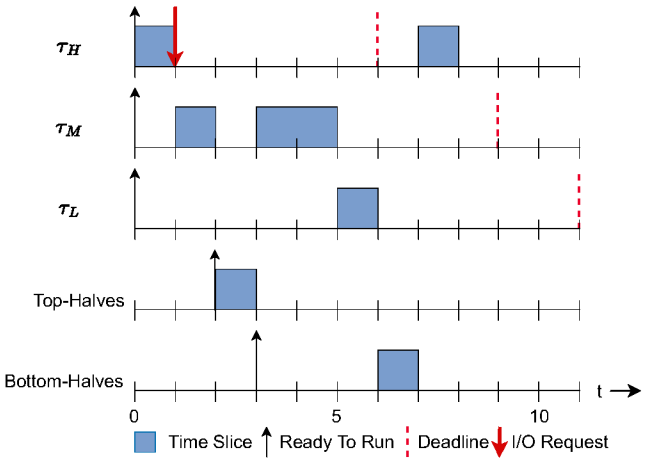



Fig. 1: Linux: $T_H$ Misses Deadline

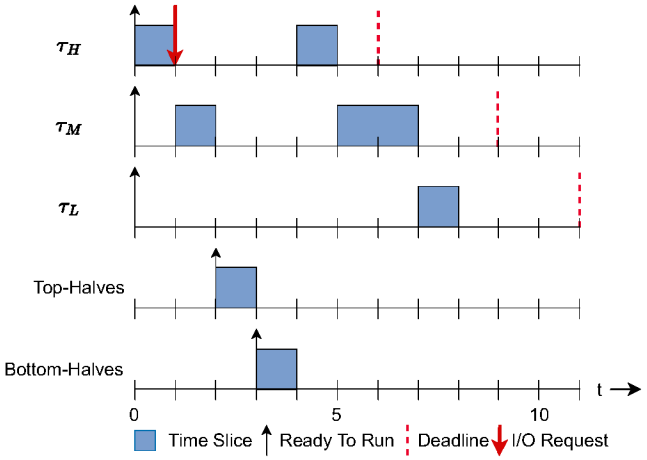



Fig. 2: Quest (No Differentiated Service): No Deadline Misses

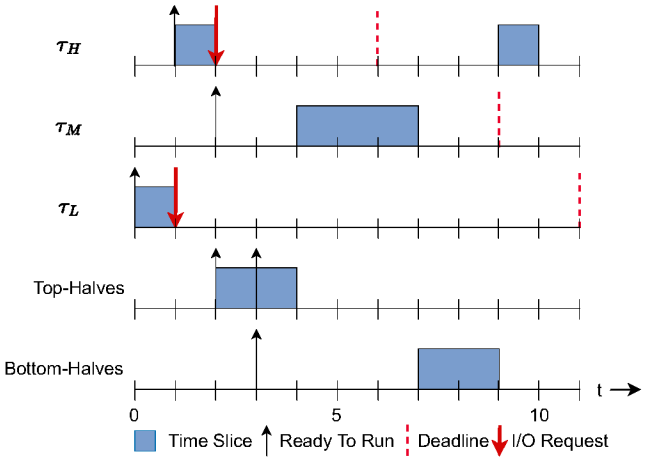



Fig. 3: Linux: $T_H$ Misses Deadline with More I/O Requests

### B. USB Interrupt Differentiated Service API

Our interrupt differentiated service API allows user-space programs to bind USB device interrupts to I/O VCPUs. This is done by a call to `usb_set_iovcpu`, which includes the file

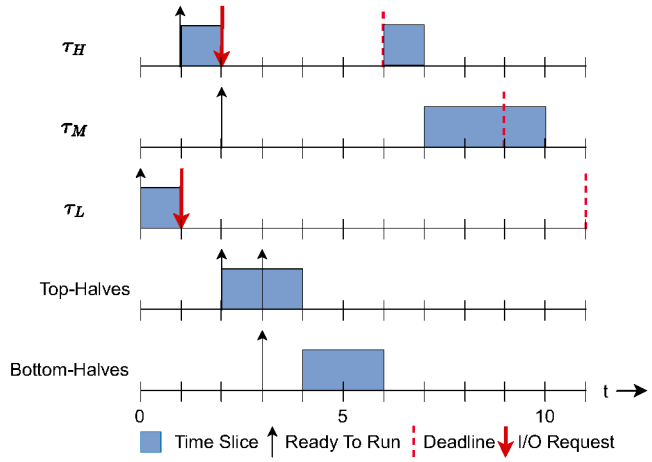



Fig. 4: Quest (No Differentiated Service): Deadline Misses with More I/O Requests

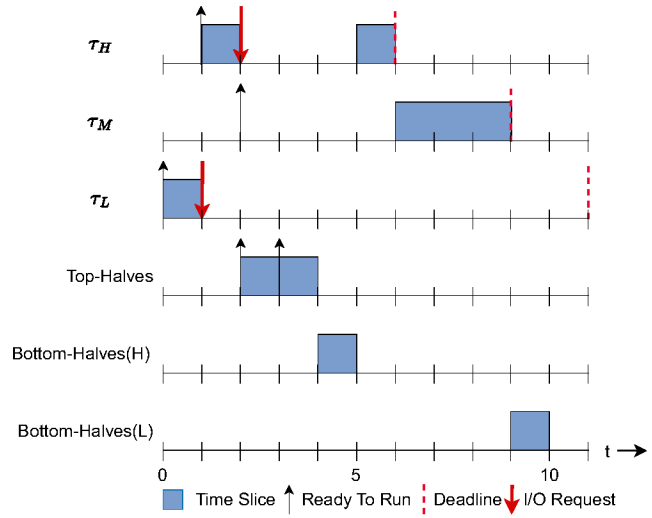



Fig. 5: Quest (Differentiated Service): No Deadline Misses with More I/O Requests

descriptor of the USB device and `user_iovcpu_class`:

```
static inline int usb_set_iovcpu(int fd,
    user_iovcpu_class iovcpu_class);
```

On success, `usb_set_iovcpu` returns 0, otherwise it returns -1. A successful function call results in all interrupts generated from I/O requests using the file descriptor `fd` to be handled with the selected I/O VCPU scheduling parameters.

The number of I/O VCPU classes is configurable, up to the total number of interrupters. Each of the values in the type `user_iovcpu_class` is one-to-one mapped to an I/O VCPU class in the kernel. For this paper, we define 8 classes as shown in `init_params`. These each have their own configurable utilization, $U = C/T$ (ranging from 1% to 12% in this example), to accommodate different I/O throughput and latency constraints. Not all I/O VCPUs need to be in use, only those that are selected for the specific throughput and delay requirements. If the system has too many active I/O VCPUs, it is possible to encounter infeasible schedules due to the additive utilization requirements they need to handle interrupts [24].

```
typedef enum {
    USER_IOVCPU_CLASS_USB0, USER_IOVCPU_CLASS_USB1,
    USER_IOVCPU_CLASS_USB2, USER_IOVCPU_CLASS_USB3,
    USER_IOVCPU_CLASS_USB4, USER_IOVCPU_CLASS_USB5,
    USER_IOVCPU_CLASS_USB6, USER_IOVCPU_CLASS_USB7
} user_iovcpu_class;

static struct sched_param init_params[] = {
    { .type = IO_VCPU, .C = 1, .T = 100,
      .io_class = IOVCPU_CLASS_USB0 },
    { .type = IO_VCPU, .C = 2, .T = 100,
      .io_class = IOVCPU_CLASS_USB1 },
    { .type = IO_VCPU, .C = 3, .T = 100,
      .io_class = IOVCPU_CLASS_USB2 },
    { .type = IO_VCPU, .C = 4, .T = 100,
      .io_class = IOVCPU_CLASS_USB3 },
    { .type = IO_VCPU, .C = 5, .T = 100,
      .io_class = IOVCPU_CLASS_USB4 }.
    { .type = IO_VCPU, .C = 6, .T = 100,
      .io_class = IOVCPU_CLASS_USB5 }
    { .type = IO_VCPU, .C = 8, .T = 100,
      .io_class = IOVCPU_CLASS_USB6 },
    { .type = IO_VCPU, .C = 12, .T = 100,
      .io_class = IOVCPU_CLASS_USB7 }
};
```

### C. API Implementation

The USB interrupt differentiated service framework dynamically binds a file descriptor to an I/O VCPU, which in turn is statically bound to an interrupter. The relationship is illustrated in Figure 6.

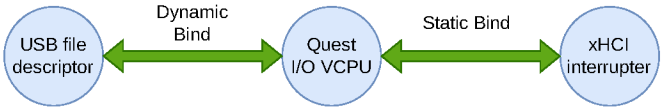



Fig. 6: Mapping of USB `fds` to I/O VCPUs and Interrupters

The mapping between I/O VCPUs and interrupters is defined by `xHCI_interrupter_to_IOVCPU_map` in Quest's USB host controller driver `xhci_hcd.c` as follows:

```
iovcpu_class xHCI_interrupter_to_IOVCPU_map[] = {
    IOVCPU_CLASS_USB0, IOVCPU_CLASS_USB1,
    IOVCPU_CLASS_USB2, IOVCPU_CLASS_USB3,
    IOVCPU_CLASS_USB4, IOVCPU_CLASS_USB5,
    IOVCPU_CLASS_USB6, IOVCPU_CLASS_USB7, 0
};
```

`xHCI_interrupter_to_IOVCPU_map` defines a mapping from interrupters to I/O VCPUs, which may be surjective or bijective. `xHCI_interrupter_to_IOVCPU_map` requires that all its fields are assigned to a corresponding I/O VCPU class. The number of entries in the map is one more than the maximum number of interrupters implemented for the target platform's USB host controller. The last entry is always 0, to mark the end of the structure. There are nine entries in the above `xHCI_interrupter_to_IOVCPU_map` because our xHCI hardware supports up to eight interrupters.

As shown in Figure 7, a thread is created for each interrupter during xHCI initialization. These threads are bound to I/O VCPUs according to the `xHCI_interrupter_to_IOVCPU_map`. The interrupt handler function in the

host controller driver (`xhci_hcd.c`) is able to tell which interrupter generated a device interrupt by checking the MSI vector number. The corresponding interrupter thread is woken up to run with the shortest period among all tasks currently waiting on the interrupter's event.

USB request blocks (URBs) capture all information necessary to perform USB transactions. An URB is a system abstraction that is converted to a *transfer request block* (TRB) when service requests are submitted to the host controller. A field in each URB contains the index of the interrupter associated with an I/O request. The correct interrupter index is determined by querying the `xHCI_interrupter_to_IOVCPU_map` with the I/O VCPU associated with the requesting task's file descriptor. The binding between a file descriptor and I/O VCPU is recorded in the caller's task structure field, `fd_iovcpu_bind`.

The host controller driver is not aware of the file descriptor used by the current task to submit USB requests. However, for each USB request, a corresponding task structure field, called `current_iovcpu`, is updated with the value of the `fd_iovcpu_bind` member. The host controller driver uses this entry in the task structure to determine the I/O VCPU to handle the interrupt on completion of the I/O request. An overview of how a file descriptor binds to an I/O VCPU is shown in Figure 8 for a `usb_read` request. A similar approach is taken for `usb_write` calls.

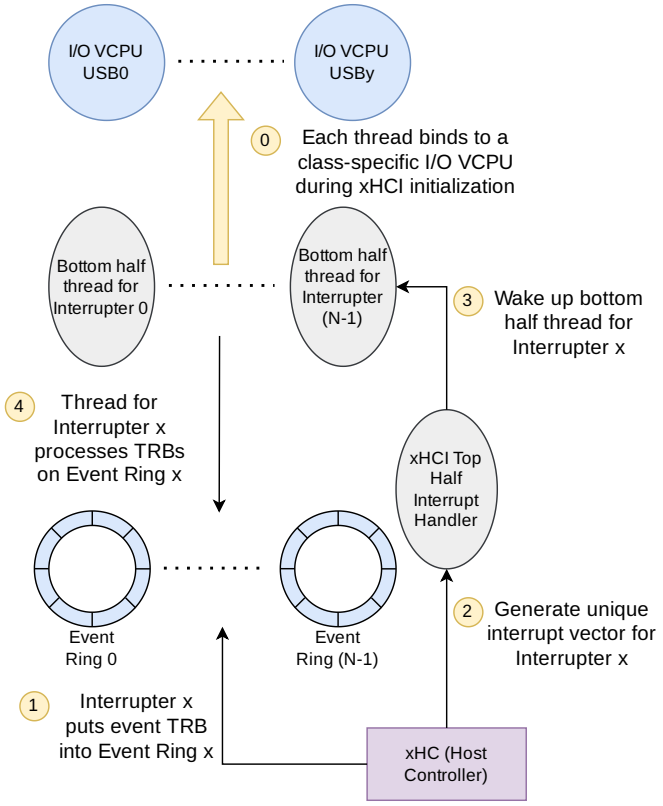



Fig. 7: Thread to I/O VCPU and Interrupter Binding

### D. Throughput & Latency Model

The throughput of a task is defined as the number of bytes transferred per unit time. The latency, as defined in this paper, is the time it takes for a USB request to complete and then return to user-space. We make the following assumptions in order to model the expected throughput and latency for a given task with our USB differentiated service framework in Quest:

- A task always issues read and write requests to the USB device directly, instead of using an intermediate buffer.
- A USB device's throughput is never less than the task's request rate; whenever a request is made to the device, data is immediately available for transfer between the USB device and host.
- The system's USB software stack has a bounded worst-case overhead for each USB request.
- Each task is mapped to an unique I/O VCPU and corresponding interrupter.

We assume a task's throughput and latency are affected by:

1) its Main VCPU's budget, $C_{Main}$, and period, $T_{Main}$,
2) the chosen I/O VCPU's utilization factor, $U_{IO}$,
3) the number of bytes, $\beta$, in each USB read/write request,
4) the USB software overhead, $\Omega_{Main}$, which consumes Main VCPU budget and which is assumed to have a bounded worst-case value,
5) the USB software overhead, $\Omega_{IO}$, which consumes I/O VCPU budget and is also assumed to have a bounded worst-case value.

Given a real-time task in Quest, its I/O VCPU's budget $C_{IO}$ is calculated as $T_{Main} \times U_{IO}$. The task consumes Main

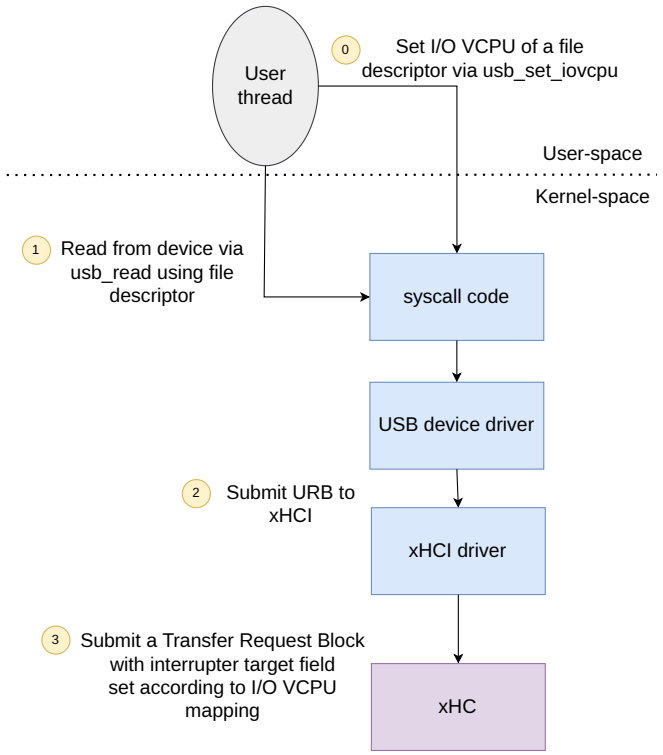



Fig. 8: File Descriptor to I/O VCPU Mapping

VCPU budget for all its execution time except bottom half interrupt handling, which is charged to the I/O VCPU. For I/O requests, the Main VCPU is typically used to allocate and free USB TRBs. The throughput and latency of I/O requests are dictated by the I/O VCPU processing delays, followed by the Main VCPU processing delays when the task is awoken on completion of the bottom half interrupt handler.

There are two cases to consider that affect the latency: one where $\Omega_{Main} \leq C_{Main}$ and the other where $\Omega_{Main} > C_{Main}$. Ideally, a task's Main VCPU budget, $C_{Main}$, should always be set greater than $\Omega_{Main}$, but for proper analysis we consider the situation where this is not the case.

Case 1 ($\Omega_{Main} \leq C_{Main}$):

$$latency \leq \frac{\Omega_{IO}}{U_{IO}} + T_{Main} \tag{1}$$

Case 2 ($\Omega_{Main} > C_{Main}$):

$$latency \leq \frac{\Omega_{IO}}{U_{IO}} + \lceil \frac{\Omega_{Main}}{C_{Main}} \rceil T_{Main} \tag{2}$$

Either case:

$$throughput = \frac{\beta}{latency} \tag{3}$$

Equations 1 and 2 are based on the fact that each I/O VCPU is a bandwidth preserving server. This accounts for the overhead $\frac{\Omega_{IO}}{U_{IO}}$. When I/O VCPU bottom half processing completes, the Main VCPU is awoken to finish the I/O request. In Case 1, the task might not get to complete execution of its budget until the end of its Main VCPU period. Hence, the latency in Equation 1 is extended by a worst-case value $T_{Main}$. In Case 2, multiple budgets and, hence, periods of the Main VCPU are needed to complete I/O processing after the bottom half is handled. In either case, the throughput is shown in Equation 3, from the calculation of latency. A task should set its Main VCPU's period to a frequency that matches the data generation rate of the device. Then, the above equations should be used to choose an appropriate $U_{IO}$ for its I/O VCPU to meet the task's latency and throughput requirements.

## IV. Experimental Evaluation

**Test Setup.** In this section, we evaluate our Quest USB differentiated service framework and compare it with Linux. The host machine used in all experiments is a Cincoze DX1100 embedded PC with a 2.4GHz Intel Core i7-8700T CPU [31]. For Linux, we use Ubuntu 20.04.2 with a `PREEMPT_RT` patched kernel, version 5.4.19 rt.

Teensy 4.1 controller boards, shown in Figure 9, connect to the DX1100 via one or more USB serial interfaces. These boards provide a means for the host machine to connect to an array of different bus protocols and pins, to support general-purpose I/O (GPIO), CAN, LIN, I2C, SPI, USART and PWM signaling, among others.

Each Teensy board features an NXP iMX RT1062 System-on-Chip (SoC) with an ARM Cortex-M7 600 MHz processor, and USB 2.0 connectivity with speeds up to 480 Mbps. We use Teensy controllers as programmable USB Communication Device Class, Abstract Control Model (CDC-ACM) devices. Each Teensy is configurable to operate in single, dual or triple serial mode, presenting one, two or three USB interfaces, respectively, to the host machine. This allows us to emulate up to three USB devices with one controller. Unless stated otherwise, all USB read and write requests are 512 bytes at a time.

**Quest and Linux CDC-ACM Drivers.** We measure the performance of our Quest xHCI differentiated services framework communicating with a CDC-ACM device driver, to exchange data between the DX1100 and the Teensy boards. We created a custom Linux USB CDC-ACM driver, to circumvent several problems with the original version.

The original Linux USB CDC-ACM driver submits a burst of 16 USB read requests to a device all at once, as soon as a user process opens the device. Whenever a USB read request completes, the data read from device is pushed to an intermediate layer called a line discipline. Then, the driver will submit another request to the device. Consequently, when a user process issues a USB read request, it goes to the line discipline layer instead of the USB CDC-ACM driver. The rationale behind Linux's approach is to increase throughput by reducing user-to-kernel context switching overhead.

Linux's default CDC-ACM driver makes it challenging to accurately measure the end-to-end latency of USB requests. By end-to-end latency, we mean the time difference between when a USB request is submitted and when the data is either read by a user process, or has been sent to a device. Measuring latency from user-space will only capture the delay from the buffered line discipline layer. Likewise, measuring latency inside the CDC-ACM driver will not include the delay to transfer data between user- and kernel-space.

To accurately determine the the end-to-end latency of I/O requests, the device driver must be modified in one of the two ways. Either the line discipline layer is removed, ensuring data is transferred directly between a device and user process, or the driver tags data with specific USB requests. We chose to take the first approach with our custom Linux driver, because it is simpler to remove the line discipline layer. This makes it easier to measure the latency of individual I/O requests without adding extra code that could increase delays to Linux.

While the original Linux driver may increase throughput, it negatively impacts the freshness of data. For example, if data is buffered in the line discipline it may be stale by the time it is read by a user process. Therefore, our custom Linux and Quest drivers both submit USB requests directly to the driver without buffering.

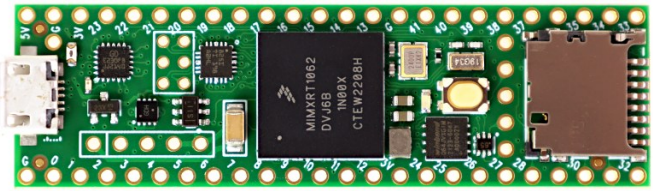


Fig. 9: Teensy 4.1

### A. Read Latency

In the first experiment, we compare the USB read latency in Quest with that of Linux. A Teensy board is connected to a DX1100 USB port. A program sketch is loaded onto the Teensy to output USB traffic as quickly as possible. A corresponding Quest program runs on a Main VCPU with a period of 20,000 microseconds and budget of 2,000 microseconds. The program is associated with an I/O VCPU that has a utilization, $U_{IO}$, of 1% for all USB read and write requests. The I/O VCPU inherits its period from the program's Main VCPU ($T_{IO} = T_{Main}$), while its budget is calculated as the I/O VCPU period multiplied by its utilization, which is 200 microseconds. The Quest program issues one USB read within its period and goes to sleep until its next period. For Linux, a similar program runs the same parameters using the SCHED_DEADLINE policy.

Using the hardware timestamp counter, we measure the time for both Quest and Linux to complete USB read requests. USB read latencies are shown in microseconds in Figure 10. There are four categories – `Quest (No kfree) With IOVCPU`, `Quest (No kfree) Without IOVCPU`, `Linux` and `Quest (kfree) With IOVCPU`. `Quest (kfree) With IOVCPU` is the regular version Quest, which releases kernel memory used for USB requests every time they complete. The `Quest (No kfree)` versions defer memory reclamation and will be discussed later in this section. We compare the overheads of using I/O VCPUs versus handling bottom half processing directly on a Main VCPU (as labeled `Without IOVCPU`). Note that because each program only issues one read per period, we are measuring the lowest possible latency here for each system.

| | Average($\mu$s) | Standard Deviation($\mu$s) | Max($\mu$s) | Min($\mu$s) |
|---|---|---|---|---|
| *Quest (kfree) With IOVCPU* | 213.83 | 7.18 | 237.97 | 187.09 |
| *Linux* | 112.26 | 21.68 | 188.57 | 35.95 |
| *Quest (No kfree) With IOVCPU* | 66.01 | 7.64 | 131.64 | 39.45 |
| *Quest (No kfree) Without IOVCPU* | 61.68 | 6.90 | 123.21 | 36.50 |

TABLE I: USB Read Latency Statistics (in Microseconds)

Table I summarizes the USB read latency statistics. While the average read latency for `Quest (kfree) With IOVCPU` is slightly higher than Linux's, there is less variability. The difference in average latency between `Quest (kfree) With IOVCPU` and `Linux` is due to how each system frees USB resources after a read. In both systems, a USB request consumes memory to submit TRBs to different queue structures, namely Event, Transfer and Command Rings. The USB stack will release those resources once the request has completed.

In Linux, at the end of the USB request completion routine, the release of memory resources is assigned to a tasklet, which will be scheduled to execute at some point in the future. In `Quest (kfree) With IOVCPU`, the USB stack releases the resources as soon as a request is completed. Therefore, there is added latency to perform the memory reclamation.

To confirm the cause of the added latency, we conduct the

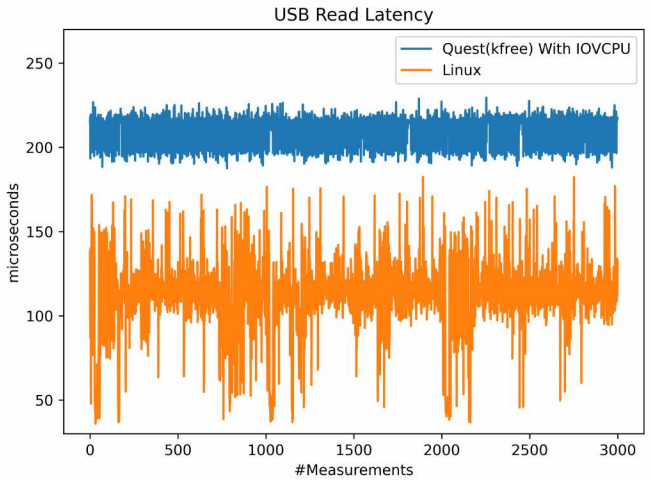



Fig. 10: Quest (kfree) With IOVCPU vs Linux

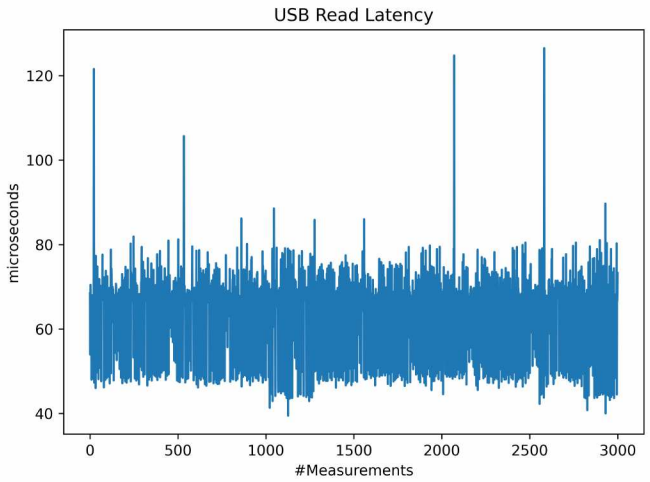



Fig. 11: Quest (No kfree) With IOVCPU

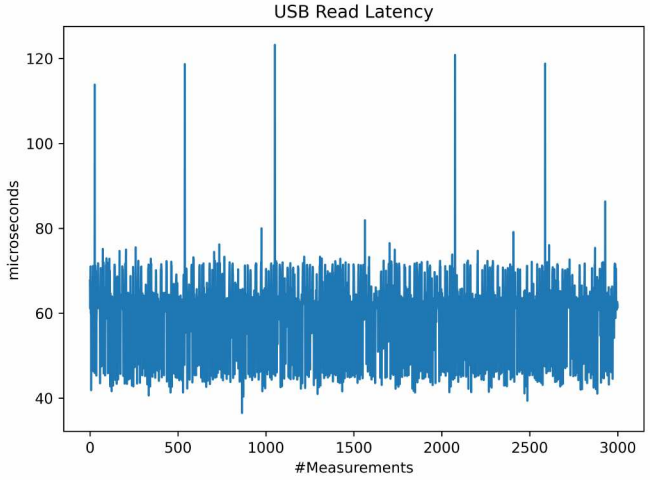



Fig. 12: Quest (No kfree) Without IOVCPU

Quest experiment again with `kfree` removed from the USB completion routine. The results for `Quest (No kfree) With IOVCPU` in Figure 11 and Table I show that Quest's read latency is both lower and less variable than Linux's. We believe the spikes around 120 microseconds in the graph are due to read transactions sometimes failing to complete within a 125 microsecond USB microframe, and are therefore placed in the next microframe by the xHC. While we cannot defer `kfree` indefinitely, as that will lead to a memory leak, it is possible to schedule resource reclamation at a suitable time in the future that does not impact the predictability of I/O requests. We leave the study of when to reclaim memory to future work.

The experiment is conducted once again, with the I/O VCPU disabled to identify whether it is affecting the latency. We change the program to run on a Main VCPU with a period of 400 microseconds and budget of 200 microseconds. The budget matches that of the case when an I/O VCPU is used, and the reduced period of the Main VCPU means its budget is replenished more frequently according to the rules of how a Sporadic Server operates. The result is shown in Figure 12. Compared to Figure 11, shows that the use of an I/O VCPU brings negligible overhead. As will be seen, I/O VCPUs bring about benefits when supporting differentiated service for separate classes of I/O requests.

### *B. I/O Task Service Guarantee*

Our next experiment compares Quest with a single interrupter (the default implementation) against Linux, and also a version of Quest with multiple interrupters. The objective of this experiment is to guarantee service to a high priority I/O task in the presence of low priority I/O tasks.

Two Teensy controllers are connected to the DX1100. Both Teensy boards operate in triple-serial mode, resulting in a total of six CDC-ACM device interfaces. Seven tasks, divided into three separate categories, are created in both Quest and Linux. The three categories encompass low, medium and high priority tasks, with constraints as shown in Table II.

Five tasks are assigned to the low priority class, and each read from a unique CDC-ACM interface until they run out of their $1000$ microsecond budgets. A single medium priority task runs in an infinite loop without using any I/O devices. Additionally, a single high priority I/O task issues a read request from a Teensy device once every $10,000$ microseconds.

The read latency for the high priority task is measured using the hardware timestamp counter. All tasks, regardless of priority, share the same USB interrupter and bottom half handling thread in Linux and in the Quest (single interrupter) system. However, in a Quest system with two USB interrupters, it is possible to differentiate I/O requests for low and high priority tasks – these tasks have their own interrupter and bottom half handling thread dedicated to a separate I/O VCPU, with its own budget and period.

The results for Linux and Quest with a single interrupter are shown in Figures 13a and 13b, respectively. For Linux, the read latency increases dramatically from what is shown in Figure 10. The reason is that all bottom halves will eventually be processed using a deferrable `ksoftirqd` thread, when the interrupt frequency passes a threshold. While all the tasks in Linux are set to run with SCHED_DEADLINE, ksoftirqd is still running with SCHED_FIFO, which by design has a lower priority. Therefore, the medium priority CPU-bound task takes precedence over ksoftirqd, causing substantial delay as shown in Figure 13a.

Even if Linux's ksoftirqd task is altered to operate under SCHED_DEADLINE, there will still be a potential mismatch of its priority and the priority of a task requesting I/O. To confirm this, we conduct the experiment again on Linux. Command line tool `chrt` is used to schedule ksoftirqd under SCHED_DEADLINE with a period of $10,000$ microseconds and budget of $1000$ microseconds. The result, shown in Figure 14, suggests the situation is worse than expected, as bottom half processing still experiences large variability in latency. Similar results occur when attempting to set the period of ksoftirqd to $5,000$ microseconds. We conclude that Linux's bottom half processing needs significant modification, to work correctly at the priority of the task issuing the I/O request. This is an area for future investigation.

| | # Tasks | Period($\mu$s) | Budget($\mu$s) |
|---|---|---|---|
| *Low Priority I/O Task* | 5 | 20000 | 1000 |
| *Medium Priority CPU Task* | 1 | 15000 | 1000 |
| *High Priority I/O Task* | 1 | 10000 | 1000 |

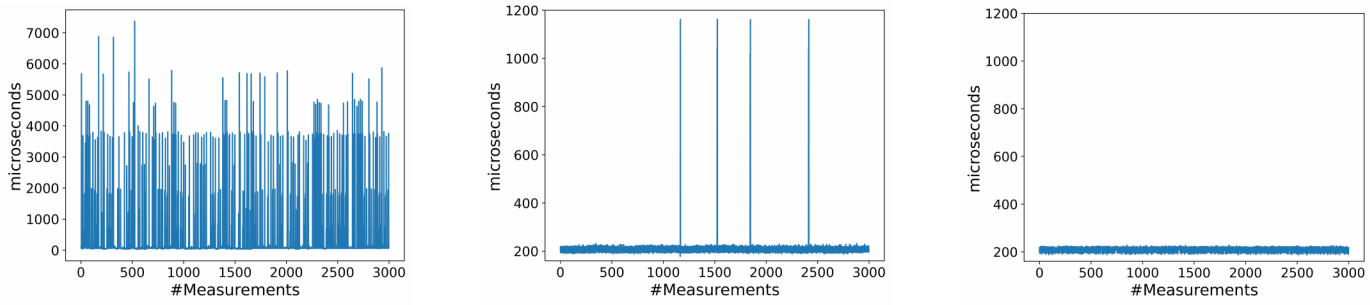
TABLE II: Priority Inversion Experiment Task Parameters

Although the latency variation is generally lower with Quest using one interrupter than with Linux, it still peaks at around 1200 microseconds, as shown in Figure 13b. Here, Quest allows a bottom half to be executed on an I/O VCPU at the (inherited) priority of the highest priority blocked task awaiting I/O completion. Both high and low priority tasks issuing I/O requests will share the same I/O VCPU, meaning a low priority request might delay the servicing of one that is higher priority. Each of the five low priority tasks delays the high priority task by around 200 microseconds, on average, in this experiment. Therefore, in total, the five low priority tasks delay the high priority task by around 1000 microseconds. Combined with the high priority task's own 200 microseconds average read latency, we see several latency spikes close to 1200 microseconds in Figure 13b.

When Quest uses two interrupters, as shown in Figure 13c, the presence of medium and low priority tasks does not affect the latency, previously shown in Figure 10. This shows that correct interrupt priority assignment using hardware interrupters is important to ensure temporal isolation between tasks.

### *C. Differentiated Service Effects on Throughput*

In this experiment, the DX1100 runs a version of Quest with multiple USB interrupters. A task with a Main VCPU period of 20,000 microseconds and budget of 2,000 microseconds reads from a single Teensy device. The task continuously issues 512 byte read requests until it is out of budget. The time it




(a) Linux (ksoftirqd with SCHED_FIFO) (b) Quest (One Interrupter) (c) Quest (Two Interrupters)

Fig. 13: High Priority Task Read Latency for (a) Linux, using ksoftirqd with SCHED_FIFO, (b) Quest with One Interrupter, and (c) Quest with Two Interrupters

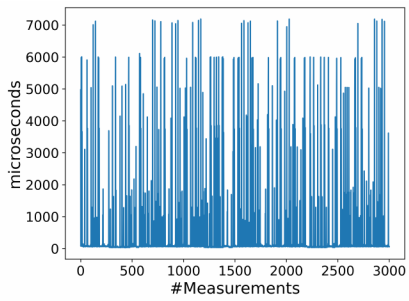



Fig. 14: High Priority Task Read Latency for Linux using ksoftirqd with SCHED_DEADLINE

takes for each USB read to complete is once again measured, using different I/O VCPU utilizations of 1%, 2% and 4%. The throughput for each case is then calculated.

Using the model in Section III-D, we set $\Omega_{Main}$ to 20 microseconds and $\Omega_{IO}$ to 200 microseconds, as determined from empirical data. In this case, the `kfree` operation to reclaim the memory of used TRB data structures is executed in the bottom half, which runs on an I/O VCPU. We compare three different I/O VCPU utilizations of 4%, 2% and 1%. Using Equations 1 and 3, where $\Omega_{Main} \leq C_{Main}$ leads to pessimistically low predicted throughputs, as the worst-case latency assumes $\Omega_{Main}$ will not complete in the context of the Main VCPU until the end of its period. In practice, we see that as $\Omega_{Main}$ is relatively small compared to the Main VCPU budget and period, we assume a more typical delay of $\frac{\Omega_{IO}}{U_{IO}} + \Omega_{Main}$. Applying this to Equation 3, our throughputs using the three different I/O VCPUs are estimated as:

$$throughput_{4\%} = \frac{512}{(\frac{0.0002}{0.04} + 0.00002)} = 101992\ bytes/s \quad (4)$$

$$throughput_{2\%} = \frac{512}{(\frac{0.0002}{0.02} + 0.00002)} = 51097\ bytes/s \quad (5)$$

$$throughput_{1\%} = \frac{512}{(\frac{0.0002}{0.01} + 0.00002)} = 25574\ bytes/s \quad (6)$$

The empirical throughput results are shown in Figure 15. The ratio between each throughput corresponds to the ratio of their I/O VCPU parameters. Additionally, the throughputs with 4%, 2% and 1% I/O VCPU utilizations closely match the expected calculations in Equations 4, 5, and 6. We conclude that our framework is able to provide analyzable differentiated services by using different I/O VCPU parameters.

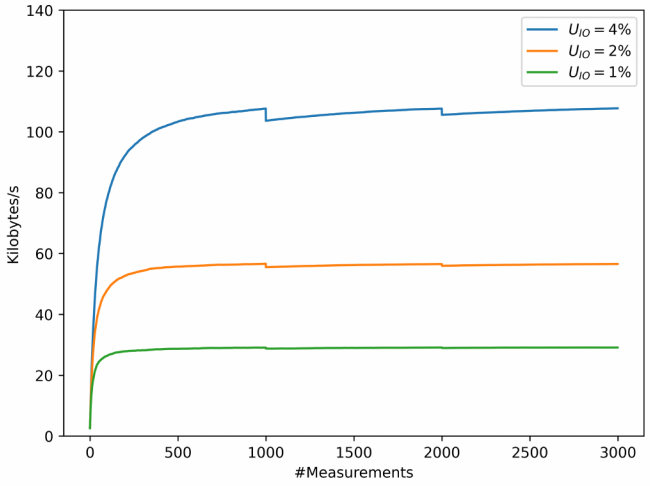



Fig. 15: USB Read Throughput for Different I/O VCPU Utilizations of 1%, 2% and 4%

### D. Round Trip Experiment

A round trip experiment is now described, which uses a single Teensy controller connected to the DX1100. The Teensy board echos back the data read from the host. A pair of reader and writer tasks running on Quest with multiple interrupter support are each assigned to a separate Main VCPU having a budget of 2,000 microseconds and a period of 10,000 microseconds. A separate interrupter is used for the I/O requests from the two tasks. The writer continually writes 512 bytes to a file descriptor connected to the Teensy, which relays the data that is then received by the reader.

Throughput measurements are shown for two experiments with I/O VCPU utilization of 1% and 2% in Figure 16. Varying the I/O VCPU parameters causes the throughput to change in proportion to the ratio between the I/O VCPU utilizations. Additionally, in each case, the write and read rate match.

Therefore, our framework is able to provide differentiated service for both reading and writing under load. In this case, $\Omega_{IO}$ is 20 microseconds, as `kfree` is deferred until the experiment completes. Assuming negligible I/O processing delay using a task's Main VCPU ($\Omega_{Main} = 0$), the expected throughputs with the two I/O VCPUs are shown below:

$$throughput_{2\%} = \frac{512}{(\frac{0.00002}{0.02})} = 512000\ bytes/s \quad (7)$$

$$throughput_{1\%} = \frac{512}{(\frac{0.00002}{0.01})} = 256000\ bytes/s \quad (8)$$

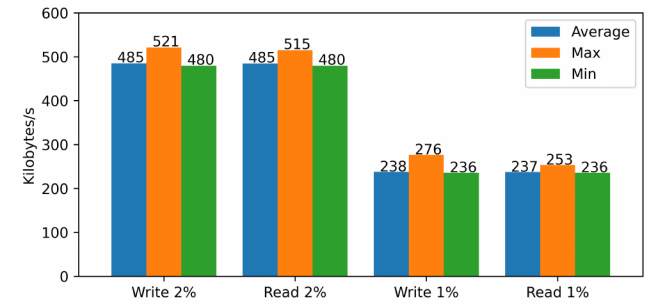



Fig. 16: Average, Maximum and Minimum USB Round Trip Throughputs with I/O VCPU Utilizations of 1% and 2%

These values match the experimental results, confirming the predictable nature of our differentiated services framework.

### *E. High Bandwidth Experiment*

In this experiment, we aim to show that Quest provides differentiated service for high bandwidth traffic, while maintain service guarantees for latency sensitive control traffic. Two Teensy controllers are used. One Teensy is configured with 3 CDC-ACM interfaces, to simulate three high bandwidth camera traffic streams, while the other one is used to simulate low throughput but latency sensitive control traffic (e.g., USB-CAN data that might be processed on a SDV central computer represented by our DX1100). According to the website cctvcalculator [32], a 60 FPS HD camera transmitting data in MJPEG format consumes bandwidth around 12,500,000 bytes per second. We use this as the target bandwidth for our most critical camera.

Four threads are created in Quest: $\tau_H, \tau_M, \tau_L$ and $\tau_C$. Threads $\tau_H, \tau_M, \tau_L$ correspond to high, medium and low bandwidth cameras, respectively, while $\tau_C$ represents a control task. `IOVCPU_CLASS_USB0` to `IOVCPU_CLASS_USB3` are used for each thread, with 3%, 2%, 1% and 1% utilizations, respectively. $\tau_H, \tau_M, \tau_L$ continuously read blocks of 9000 bytes of data from the first Teensy, while $\tau_C$ reads 512 bytes from the second Teensy once every period.

Initially, $\tau_H$ is mapped to `IOVCPU_CLASS_USB0`, $\tau_M$ is mapped to `IOVCPU_CLASS_USB1`, $\tau_L$ is mapped to `IOVCPU_CLASS_USB2` and $\tau_C$ is mapped to `IOVCPU_CLASS_USB3`. After a certain number of read requests, $\tau_H$ will make a mode change so that it is remapped to `IOVCPU_CLASS_USB2`. At the same time, $\tau_L$ is remapped to `IOVCPU_CLASS_USB0`. This mode change simulates the scenario where the criticality of cameras changes over time, thus leading changes to their service requirements. This could be representative of a front-facing vehicular camera used for active cruise control initially being highest criticality, only for a side-facing camera later becoming more important when handling a lane change. A summary of the setup of threads is shown in Table III.

| | Period($\mu$s) | Budget($\mu$s) |
|---|---|---|
| $\tau_H$ | 10000 | 2500 |
| $\tau_M$ | 10000 | 1000 |
| $\tau_L$ | 10000 | 2500 |
| $\tau_C$ | 10000 | 500 |

TABLE III: High Bandwidth Experiment Thread Parameters

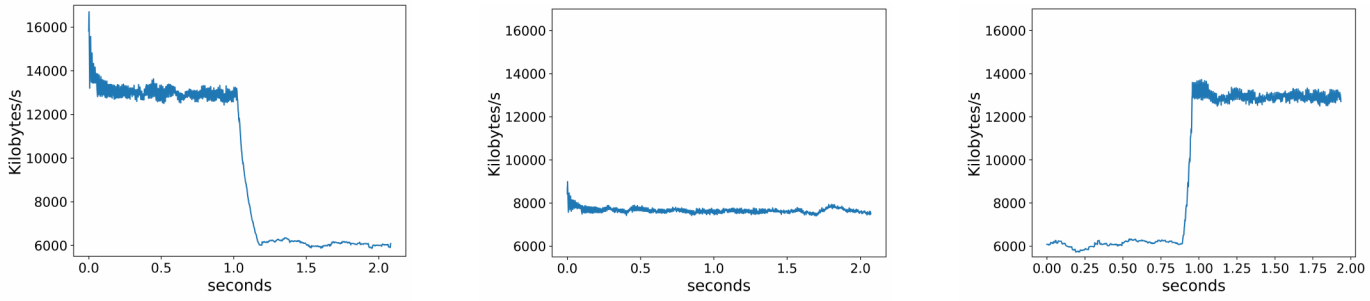
The throughput results for $\tau_H$, $\tau_M$, $\tau_L$ and $\tau_C$ are shown in Figures 17a, 17b, 17c and 18, respectively. The read latency graph for control traffic is shown in Figure 19.

As shown in Figure 17a, the throughput of $\tau_H$ starts around 13,500,000 bytes per second. After the mode switch, $\tau_H$'s throughput quickly drops to around 6,000,000 bytes per second, and the throughput for $\tau_L$ switches the opposite way, as shown in Figure 17c. $\tau_L$'s throughput starts around 6,000,000 bytes per second and quickly reaches 13,500,000 bytes per second after the mode switch. Meanwhile, $\tau_M$ and $\tau_C$'s throughputs are not affected by the mode switch at all, as shown in Figures 17b and 18. Additionally, Figure 19 shows that the read latency of $\tau_C$ is not affected by other traffic. Thus, this experiment demonstrates that Quest is able to provide differentiated service for high bandwidth traffic while maintaining latency service guarantees for critical low bandwidth control traffic.

## V. Related Work

Linux/RK [33] is one of the first systems to support processor capacity reserves [34], as a way to manage CPU time for tasks and interrupts. This is similar to the use of time-budgeted virtual CPUs for task and interrupt scheduling in our USB differentiated services framework. This has similarities to Linux systems supporting fine-grained preemption [35] and `SCHED_DEADLINE` tasks, which reserve CPU bandwidth according to a Constant Bandwidth Server algorithm [36]. However, as with other RTOSs [37]–[39] that attempt to provide temporal isolation between tasks, there is no proper integrated scheduling of interrupts with resource reservations or correct prioritization.

Works such as klmirqd [40] and process-aware interrupt scheduling [22] attempt to address real-time interrupt handling by identifying the process waiting for the I/O device and then letting the interrupt bottom half inherit the priority of the process. In the case where there are multiple processes waiting for a device, the bottom half will inherit the highest priority of all the waiting processes. However, once again, none of those works constrain interrupt handling to CPU reservations. Motivated by process-aware interrupt scheduling [22], and




(a) $\tau_H$ Throughput (b) $\tau_M$ Throughput (c) $\tau_L$ Throughput

Fig. 17: Throughputs for $\tau_H$, $\tau_M$ and $\tau_L$ with Mode Changes for $\tau_H$ and $\tau_L$

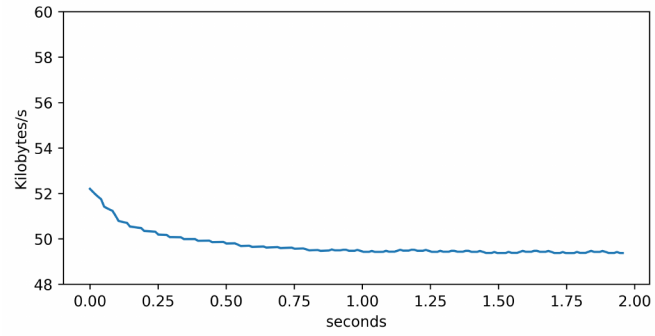



Fig. 18: $\tau_C$ USB Control Data Throughput

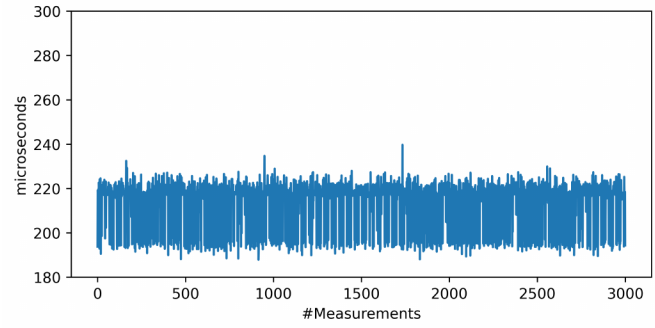



Fig. 19: $\tau_C$ USB Control Data Read Latency

the use of reservations [41], Quest [24] provides unified scheduling and temporal isolation for both tasks and interrupts. It does this using time-budgeted virtual CPUs.

Other related work has addressed real-time Universal Serial Bus scheduling [7]–[9], [17], [18]. However, none of this work has investigated how to provide USB differentiated service. This is because all USB-related interrupt handling is limited to one interrupter for all devices connected to the host. With our USB differentiated service framework, the mapping of a device interrupt to a time-budgeted server is handled by hardware. Using Quest's VCPUs, our framework is able to provide temporal isolation between tasks and interrupt bottom halves as well as fine-grained control of delay and throughput bound for I/O tasks.

While related work has looked at combined scheduling of interrupts and tasks [21]–[23], or the handling of all threads as interrupts [42], this is the first work to our knowledge to investigate differentiated interrupt management for USB devices.

## VI. Conclusions And Future Work

This paper introduces a USB differentiated service framework, to correctly associate interrupt bottom half handling with I/O requests from tasks of different criticalities. We implemented CDC-ACM drivers for the Quest RTOS and Linux, to support Teensy devices connected to a USB host controller. Experiments show the benefits of using multiple USB interrupters in combination with Message Signaled Interrupts to process I/O events. Each I/O event that triggers an interrupt is correctly associated with a time-budgeted server, to guarantee throughput and delay constraints. Bottom half interrupt handlers are therefore scheduled at rates corresponding to the tasks that led to their execution.

Applying USB differentiated services to Quest shows how it is able to avoid added delays to high priority tasks caused by priority inversion. This contrasts with priority inversion experienced by both Quest and Linux using only one USB interrupter. Even when Linux tasks are scheduled using SCHED_DEADLINE, priority inversion still exists, because Linux does not adequately schedule interrupt bottom halves.

As part of future work, we plan to investigate real-time garbage collection techniques to both bound and reduce the delay of our USB software stack. Lessons learned from this work will be used in the context of differentiated services for real-time networks. We plan to compare USB differentiated services with those of protocols related to Time-Sensitive Networking [43], including IEEE 802.1Qbv time-aware traffic shaping [44].

## VII. Acknowledgements

This work is supported in part by the National Science Foundation (NSF) under Grant # 2007707. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF. Special thanks are also given to our colleagues at Drako Motors who have helped with many issues related to USB device I/O and choice of microcontroller boards.

## References


[1] "TIA-232-F: Interface between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange," 1997.

[2] *M68HC11 Reference Manual*, 6th ed., 2007.

[3] *The $I^2C$–Bus Specification*, 2nd ed., January 2000.

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

[5] "Gigabit Multimedia Serial Link: https://www.analog.com/en/applications/technology/gigabit-multimedia-serial-link.html."

[6] "Automotive MAC-PHY devices connect low cost MCUs to Ethernet: https://www.embedded.com/automotive-mac-phy-devices-connect-low-cost-mcus-to-ethernet/," September 2023.

[7] E. Missimer, Y. Li, and R. West, "Real-time USB Communication in the Quest Operating System," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2013, pp. 11–20.

[8] R. West, A. Golchin, and A. Njavro, "Real-time USB Networking and Device I/O," *ACM Transactions on Embedded Computing Systems (ACM TECS)*, vol. 22, no. 4, pp. 1–38, 2023, https://doi.org/10.1145/3604429.

[9] A. Golchin, Z. Cheng, and R. West, "Tuned Pipes: End-to-end Throughput and Delay Guarantees for USB Devices," in *39th IEEE Real-Time Systems Symposium (RTSS)*, 2018.

[10] S. Vestal, "Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance," in *Proceedings of the 28th IEEE Real-Time Systems Symposium (RTSS)*, 2007, pp. 239–243.

[11] G. Niedrist, "Deterministic Architecture and Middleware for Domain Control Units and Simplified Integration Process Applied to ADAS," 2016, https://www.tttech.com/technologies/adas.

[12] S. Aust, "Vehicle Update Management in Software Defined Vehicles," in *2022 IEEE 47th Conference on Local Computer Networks (LCN)*, 2022, pp. 261–263.

[13] C. Bodei, M. D. Vincenzi, and I. Matteucci, "From Hardware-Functional to Software-Defined Vehicles and their Security Issues," in *2023 IEEE 21st International Conference on Industrial Informatics (INDIN)*, 2023, pp. 1–10.

[14] F. Pan, J. Lin, M. Rickert, and A. Knoll, "Resource Allocation in Software-Defined Vehicles: ILP Model Formulation and Solver Evaluation," in *2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC)*, 2022, pp. 2577–2584.

[15] "BlackBerry — QNX: Software-Defined Vehicles: https://blackberry.qnx.com/en/ultimate-guides/software-defined-vehicle," checked October 25, 2023.

[16] "ARM Automotive: Software-Defined Vehicles: https://www.arm.com/markets/automotive/software-defined-vehicles," checked October 25, 2023.

[17] 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*, ser. RTAS '03. Washington, DC, USA: IEEE Computer Society, 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=827266.828522

[18] 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*, ser. RTSS '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 395–404. [Online]. Available: http://dx.doi.org/10.1109/REAL.2004.45

[19] D. Tennenhouse, "Layered Multiplexing Considered Harmful," in *Protocols for High-Speed Networks*, North Holland, Amsterdam, 1989, pp. 143–148.

[20] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. ACM, December 1995, pp. 40–53.

[21] L. E. Leyva-del-Foyo, P. Mejia-Alvarez, and D. de Niz, "Predictable Interrupt Management for Real Time Kernels over conventional PC Hardware," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2006.

[22] Y. Zhang and R. West, "Process-Aware Interrupt Scheduling and Accounting," in *27th IEEE Real-Time Systems Symposium (RTSS)*, 2006.

[23] G. Parmer and R. West, "Predictable interrupt management and scheduling in the Composite component-based system," in *Proceedings of the 29th IEEE Real-Time Systems Symposium*, Barcelona, Spain, December 2008.

[24] M. Danish, Y. Li, and R. West, "Virtual-CPU Scheduling in the Quest Operating System," in *17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011, pp. 169–179.

[25] *Intel 82093AA I/O Advanced Programmable Controller(IOAPIC)*, https://pdos.csail.mit.edu/6.828/2018/readings/ia32/ioapic.pdf, Intel corporation, 1996.

[26] J. Coleman, "Reducing interrupt latency through the use of message signaled interrupts," January 2009. [Online]. Available: https://www.techonline.com/wp-content/uploads/2020/09/media-1036817-intel_321070.pdf

[27] "Quest OS: http://www.questos.org."

[28] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time Systems," *Real-Time Systems Journal*, vol. 1, no. 1, pp. 27–60, 1989.

[29] E. Missimer, K. Missimer, and R. West, "Mixed Criticality Scheduling with I/O," in *Proceedings of the 28th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2016.

[30] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *JACM*, 1973.

[31] "Cincoze: www.cincoze.com."

[32] "CCTV Bandwidth Calculator," https://www.cctvcalculator.net/en/calculations/bandwidth-calculator/.

[33] S. Oikawa and R. Rajkumar, "Linux/RK: A Portable Resource Kernel in Linux," in *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium (RTAS)*, Jun. 1998.

[34] C. Mercer, S. Savage, and H. Tokuda, "Processor Capacity Reserves: An Abstraction for Managing Processor Usage," in *Proceedings of the 4th Workshop on Workstation Operating Systems*, 1993, pp. 129–134.

[35] PREEMPT_RT, 2019, https://rt.wiki.kernel.org/index.php/Main_Page.

[36] L. Abeni and G. Buttazzo, "Integrating Multimedia Applications in Hard Real-Time Systems," in *Proceedings of the 19th IEEE Real-time Systems Symposium*, 1998, pp. 4–13.

[37] VxWorks, 2019, https://www.windriver.com/products/vxworks/.

[38] D. Bayer and H. Lycklama, "MERT-A Multi-environment Real-time Operating System," in *ACM SIGOPS Operating Systems Review*, vol. 9, no. 5. ACM, 1975, pp. 33–42.

[39] J. J. Labrosse, *MicroC/OS-II: The Real Time Kernel*. CRC Press, 2002.

[40] G. Elliott and J. Anderson, "Robust Real-time Multiprocessor Interrupt Handling Motivated by GPUs," in *24th Euromicro Technical Committee on Real-Time Systems*, 2012, pp. 267–276.

[41] N. Manica, L. Abeni, and L. Palopoli, "Reservation-Based Interrupt Scheduling," in *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, Stockholm, Sweden, April 12–15 2010.

[42] W. Hofer, D. Lohmann, F. Scheler, and W. Schröder-Preikschat, "SLOTH: Threads as Interrupts," in *Proceedings of the 30th IEEE Real-Time Systems Symposium*, Washington DC, USA, December 01–04 2009.

[43] "IEEE standard for local and metropolitan area network–bridges and bridged networks," *IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014)*, pp. 1–1993, 2018.

[44] "IEEE standard for local and metropolitan area networks – bridges and bridged networks - amendment 25: Enhancements for scheduled traffic," *IEEE Std 802.1Qbv-2015*, 2015.