Real-Time USB Networking and Device I/O

RICHARD WEST, AHMAD GOLCHIN, and ANTON NJAVRO, Department of Computer Science, Boston University, USA

Multicore PC-class embedded systems present an opportunity to consolidate separate microcontrollers as software-defined functions. For instance, an automotive system with more than 100 electronic control units (ECUs) could be replaced with one or, at most, several multicore PCs running software tasks for chassis, body, powertrain, infotainment, and advanced driver assistance system (ADAS) services. However, a key challenge is how to handle real-time device input and output (I/O) and host-level networking as part of sensor data processing and control. A traditional microcontroller would commonly feature one or more Controller Area Network (CAN) buses for real-time I/O. CAN buses are usually absent in PCs, which instead feature higher bandwidth Universal Serial Bus (USB) interfaces. This article shows how to achieve real-time device I/O and host-to-host communication over USB, using suitably written device drivers and a time-aware POSIX-like "tuned pipe" abstraction. This allows developers to establish task pipelines spanning one or more hosts, with end-to-end latency and throughput guarantees for sensor data processing, control, and actuation.

CCS Concepts: • **Software and its engineering → Real-time schedulability**; **Input/output**; • **Computer systems organization → Real-time operating systems**; **Embedded systems;**

Additional Key Words and Phrases: Universal Serial Bus (USB), extensible Host Controller Interface (xHCI), real-time input/output, real-time host-to-host communication

ACM Reference format:
Richard West, Ahmad Golchin, and Anton Njavro. 2023. Real-Time USB Networking and Device I/O. *ACM Trans. Embedd. Comput. Syst.* 22, 4, Article 67 (July 2023), 38 pages.
https://doi.org/10.1145/3604429

1 INTRODUCTION

Embedded and real-time systems interact with their environment using sensors and actuators. Sensor inputs are processed, leading to control decisions that produce output signals to the actuators. As the functional complexity of modern embedded systems has grown, the **input/output** (**I/O**) processing and control are distributed across a network of low-cost microcontrollers. For example, a modern automotive system features 10s to 100s of **electronic control units** (**ECUs**), each hosting a separate microcontroller, to support chassis, body, powertrain, infotainment, and **Advanced Driver Assistance System** (**ADAS**) services. Each such ECU is largely responsible for a single function, with multiple ECUs typically connected via a communication bus, such as **Controller Area Network** (**CAN**) [32] or FlexRay [22].

This work is funded in part by the National Science Foundation (NSF) Grant # 2007707.
Authors' address: R. West, A. Golchin, and A. Njavro, Department of Computer Science, Boston University, 111 Cummington Mall, Boston, Massachusetts 02215, USA; emails: {richwest, golchin, njavro}@bu.edu.

1539-9087/2023/07-ART67 $15.00
https://doi.org/10.1145/3604429

In an attempt to curtail the exponential growth in the number of ECUs or microcontrollers in embedded systems, there is now a focus on functional consolidation. Consolidation replaces a collection of microcontrollers with a smaller number of multicore computers (e.g., embedded Personal Computers, or PCs), which run multiple functions in software. However, the use of PC-class hardware in embedded systems poses several challenges. Firstly, PCs have limited I/O bus interfaces, mostly based on **Peripheral Component Interconnect Express** (**PCIe**) or **Universal Serial Bus** (**USB**), which differ from those in traditional embedded systems. Secondly, the **operating systems** (**OSs**) running on PCs to manage software functions provide poor support for real-time I/O. Thirdly, a PC-based OS must provide real-time scheduling of multiple software functions and I/O requests to meet end-to-end delay guarantees for sensor data processing and control.

To address the first challenge, there needs to be a way to connect I/O devices and controllers that traditionally use low-bandwidth interfaces such as CAN, I2C [62], SPI [53], and RS232 [80] to a PC. Fortunately, many embedded I/O interfaces are available as USB devices, and a USB host controller has the capacity to handle relative high-bandwidth sensors (e.g., cameras) and actuators that are now relevant to emerging applications, including autonomous vehicles. A USB host controller is equipped with a hardware bus scheduler that provides real-time guarantees to devices with periodic transfers. Unfortunately, many USB devices (e.g., USB-CAN interfaces) only support best-effort, asynchronous data exchange with the host. To combat this, bus scheduling within the host OS must provide real-time guarantees on asynchronous transfers.

To address the second challenge, an embedded OS must handle real-time I/O. This is a problem for systems such as Linux, which allow device interrupts to interfere with the execution of tasks. Either the interrupt handler is given precedence over the task it preempts, which might then miss a critical deadline, or the interrupt handler is deferred leading to delayed I/O transfers. Linux attempts to address this problem by splitting **interrupt service routines** (**ISRs**) into two parts: a *top* half that runs briefly when the interrupt occurs, and (2) a *bottom* half that performs the majority of the interrupt handling at a time that is potentially more convenient. As we have previously identified, the splitting of interrupt handlers into two parts does not guarantee that interrupts are handled at the correct priority [12, 88]. Moreover, it does not ensure temporal isolation between interrupt handling and task execution.

The third challenge concerns the coordination of host-level tasks and devices, or network-level resources, to ensure end-to-end throughput and delay guarantees on the transfer of data. USB has become an industry standard for its ability to support many different classes of devices and communication endpoints, with relatively simple hardware needed to connect to a host. The handling of interrupts, and **direct memory access** (**DMA**) control, for example, are addressed by the USB host controller. This differs from other bus technologies, such as PCIe, where the device or peripheral interface must include a controller that manages interrupts and DMA transfers. While this has influenced the popularity of USB, most OSs do not adequately implement a real-time host controller driver, to schedule access to the bus interface by different peripherals. Even worse, OSs lack the capability to coordinate host and device, or network-level, resources to meet end-to-end service requirements.

To address the challenges described above, the contributions of this article are: (1) a real-time USB 3.x scheduler that supports both periodic and asynchronous requests, for network and device transfers, (2) a system solution for integrated task and interrupt scheduling, and (3) the description of a "Tuned Pipes" OS-level abstraction for real-time sensor data processing and control [26]. Tuned Pipes provide a way to coordinate host and device or network resources according to end-to-end throughput and delay requirements.

This article builds on our earlier work on USB 2.0 scheduling, to circumvent the problems found in Linux systems [59]. We show how to implement a software-based USB 3 scheduler that works

with the host controller, to provide timing guarantees for asynchronous devices and host-to-host communication. This article shows how to combine interrupt and task scheduling, so that interrupts are handled at the priority of the task that leads to their occurrence. Tuned Pipes combines USB, task and interrupt scheduling to meet end-to-end service requirements.

Experimental results show that for a mixed-criticality system, our real-time bus scheduler guarantees **quality of service** (**QoS**) for high-criticality traffic, while inducing minimal loss on low-criticality data. Here, criticality is defined as the consequence of failure to the system. Likewise, experimental results show how USB 3.x is able to support predictable host-to-host communication and device I/O. We believe this is a step towards USB being a viable bus technology for control area networks, in systems requiring higher throughput and similar delays to those experienced by CAN buses.

In the following section, we briefly introduce the necessary background knowledge for this work with a focus on the structure of USB 3.x host controller hardware. This is followed by Section 3, which describes the scheduling problem of the USB 3.x bus, and presents our solution to address the problem in the context of a mixed-criticality I/O system. Section 4 focuses on designing different topologies of host computers using USB 3.x. Section 5 describes the design of our tuned pipes framework. An experimental evaluation of our USB system is shown in Section 6. Related work is discussed in Section 7, followed by conclusions in Section 8.

2 BACKGROUND

This section provides background relevant to coordinated task and interrupt scheduling, as well as fundamental concepts related to USB. The mathematical concepts underpinning this and subsequent sections are described by a series of equations. The key symbols used throughout the rest of the article are summarized in Table 1, with a reference to the equations where they appear. Values are shown in parentheses for symbols, where appropriate.

2.1 Task and Interrupt Scheduling

End-to-end guarantees on I/O processing require coordinated task and interrupt scheduling, which is a central feature of our Quest real-time OS [12]. As is the case with Linux, Quest splits interrupt handling into a top and bottom half. Top half processing is limited to acknowledging the interrupt, identifying the priority for a corresponding bottom half thread, and setting a wakeup event for when the bottom half is eligible to execute. While most of the interrupt processing is charged to a dedicated thread, the top half executes in the context of the preempted thread, and so its execution is kept minimal.

In Quest, bottom half interrupt handlers and tasks are bound to time-budgeted **virtual CPUs** (**VCPUs**). Each VCPUi is specified a processor capacity reserve [57], consisting of a budget capacity, C_i, and period, T_i, depending on the corresponding worst case execution time and period. VCPU i is eligible to receive up to C_i units of execution time every T_i time units when it is runnable, as long as a schedulability test is passed when creating new VCPUs. This way, Quest's scheduling subsystem guarantees temporal isolation between threads in the runtime environment.

Quest supports two types of VCPUs: Main VCPUs are associated with conventional tasks, while I/O VCPUs are associated with interrupt bottom halves. Each Main VCPU is implemented as a Sporadic Server [74] that is scheduled by default using **Rate-Monotonic Scheduling** (**RMS**) [51]. An optional configuration supports the earliest deadline first, but for this article, we assume RMS is in use. In contrast, each I/O VCPU operates as a bandwidth-preserving server having a dynamically-calculated budget and period based on the Main VCPU task it is handling I/O on behalf. Quest features several classes of I/O VCPUs for networking, disk, and USB devices, amongst others. By default, a single I/O VCPU is setup for a USB host controller driver to handle interrupt bottom halves.

Table 1. Main Symbol Definitions and Related Equations

Symbol	Equation	Description
C_i	(1)	Budget capacity for Virtual CPU (VCPU) i
T_i	(1), (8), (9), (10)	Period for VCPU i
$\frac{C_i}{T_i}$	(1)	CPU reservation (i.e., Utilization factor) for Main VCPU i
U_j	(1)	Utilization factor for I/O VCPU j
b_i	(2), (3), (4)	Maximum packet size in bytes for endpoint i
n_i	(2), (4), (5), (6), (7)	Maximum burst size (packets minus 1) for endpoint i
m_i	(2), (7)	Consecutive bursts (minus 1) for endpoint i
q_i	(2)	Quantum size bytes for endpoint i
O_{bits}	(3)	Bit stuffing overhead (3.167)
γ	(3)	Packet payload synchronization cost (1.167 [USB2.0], 1.0 [USB3.0])
w_i	(3), (5), (6), (7)	Packet transfer latency for endpoint i
h	(3)	Host controller delay (assumed 5ns, implementation-specific)
p	(3)	Protocol overhead delay (depends on endpoint type and bus speed)
α	(3)	Symbol (bit) transfer latency (0.2ns [USB 3@5Gbps], 2.083ns [USB 2@480Mbps])
$\mu frame$	(7)	Micro-frame time (125000ns)
k_j	(4), (5), (6)	Number of round-robin schedule passes for endpoint j
B_j	(4)	Transfer budget bytes for endpoint j
l_j	(5)	Time to transfer B_j bytes for endpoint j
c_j	(6), (8), (10)	Endpoint j transfer latency (in micro-frames) for B_j bytes
A	(5), (6)	Set of asynchronous endpoints
H	(7)	Set of high-criticality periodic endpoints
t_A	(6)	Time within micro-frame to complete asynchronous requests
t_{Amax}	(7)	Upper bound on t_A
E_{wc}	(9), (10)	Worst-case end-to-end latency

When a device interrupt occurs, the task associated with its occurrence is determined. For example, if some task τ initiated an I/O request that led to a device interrupt being generated, the I/O VCPU inherits the period (denoted by T_{main}) of the Main VCPU associated with τ. Since VCPUs are scheduled using RMS by default, the I/O VCPU also inherits the priority of τ's Main VCPU. This way, Quest is able to perform bottom half interrupt processing with the priority of the task that initiated the I/O requests. Consequently, interrupt handling does not defer the execution of higher priority tasks or suffer delays from lower priority tasks.

Instead of using Sporadic Servers for both Main and I/O VCPUs, each I/O VCPU j operates as a **Priority Inheritance Bandwidth-preserving Server** (**PIBS**) [12, 15, 44], where a certain utilization factor U_j is specified to limit its bandwidth. For example, the budget of I/O VCPU j will be limited to $T_{main} \cdot U_j$. With that, even if τ's Main VCPU has a large budget and issues a burst of I/O requests, the CPU cycles consumed by the handling of corresponding device interrupts are limited by the I/O VCPU's bandwidth. PIBS uses a single replenishment to avoid fragmentation of replenishment list budgets caused by short-lived interrupt bottom half service routines. By using PIBS for interrupt threads, the scheduling overheads from context switching and timer reprogramming are reduced [60]. In prior work [12], we showed that for n Main VCPUs and m I/O VCPUs running on a single physical CPU, temporal isolation is guaranteed amongst all VCPUs if Equation (1) is satisfied:

$$\sum_{i=0}^{n-1} \frac{C_i}{T_i} + \sum_{j=0}^{m-1} (2 - U_j) \cdot U_j \leq n \left(\sqrt[n]{2} - 1 \right) \tag{1}$$

Quest's support for PIBS-based I/O VCPUs contrasts with related work that attempts to maintain temporal isolation between tasks sharing resources. de Niz et al. [15] compare two schemes:

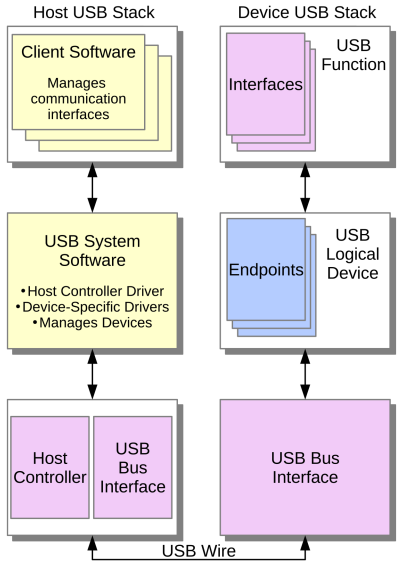

Fig. 1. USB host and device stacks.

(1) a priority inheritance approach that allows a task to use a non-depleted processor reserve for itself and any task that it blocks, and (2) a priority ceiling-based approach [70] that establishes a special processor reserve for use in critical sections. Similarly, Lamastra et al. [44] apply the idea of priority inheritance to tasks sharing resources using processor reserves managed by **Constant Bandwidth Servers** (**CBS**) [1].

Quest's I/O VCPUs have a separate reserve for the purposes of interrupt handling. Only the priority of the Main VCPU being served by the I/O VCPU is inherited, as the I/O VCPU utilization is established when the system is configured. This makes sense because interrupt handlers can be profiled, to establish how much CPU time they need to execute.

2.2 Universal Serial Bus

USB is a master-slave protocol that connects a host computer (the master) to one or more peripheral devices (the slaves). As of USB 3.0, a device operates at one of four possible communication rates: *low*, *full*, *high*, or *super* speed, with maximum throughputs of 1.5 Mbps, 12 Mbps, 480 Mbps, and 5 Gbps, respectively. Recent advances with USB 3.2 now increase bus bandwidth up to 20 Gbps. This maximum bandwidth is not necessarily shared by all the devices connected to a USB host controller. The effective bandwidth share depends on the speed of the USB device, and how the system software programs the host controller to schedule USB packets on the data transfer bus associated with the device in question. Figure 1 depicts the hardware-software structure of both a USB host and device, which communicate over a physical link. 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. A function supports alternative interfaces to enable or disable certain endpoints and/or change their data rate. Each endpoint specifies the type of data transfer, 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.

The USB protocol supports four types of data transfer between a host and a device: (1) *Control transfers* for device configuration, (2) *Bulk transfers* for reliable delivery of non-real-time data, (3) *Isochronous transfers* for real-time, loss-tolerant data, and (4) *Interrupt transfers* for real-time, loss-sensitive data. Different devices support different transfer types. For example, a USB camera is typically isochronous, a mass storage device usually supports bulk transfers, and a keyboard works with interrupt transfers. Each USB device exposes up to 32 *endpoints*, each of which is capable of transferring data in one direction only, according to one of the four transfer types mentioned above.

USB *transactions* are always initiated by the host computer, with peripheral devices only capable of responding to host requests. In USB 1.0/1.1, all transactions occur within a frame set at 1 millisecond. Any USB 1.0/1.1 transactions crossing a frame boundary are discarded. USB 2.0 and higher support micro-frames of 125 microseconds. Each frame contains eight micro-frames and USB 2.0+ transactions cannot cross a micro-frame boundary, or else they will be discarded. All transactions are split into two classes: (1) *periodic*, which is for isochronous and interrupt transfers, and (2) *asynchronous*, which is for bulk and control transfers.

The **Enhanced Host Controller Interface** (**EHCI**) [20] implementation for USB 2.0 organizes periodic transactions in a frame list. The host controller indexes the list using a *frame index register*, which is incremented every micro-frame.

Asynchronous transactions are ordered in a separate round-robin circular list. The USB Specification [82, 83] limits the time available to schedule data transfers from the periodic frame list before the circular asynchronous list is processed. For example, high-speed periodic transfers are limited to 80% of a micro-frame, leaving at least 20% to asynchronous transfers. USB 3 super-speed periodic traffic can occupy up to 90% of the micro-frame time, thus guaranteeing at least 10% for bulk and control traffic. Each time the asynchronous list is processed, it resumes with the next transfer after the one previously processed.

Periodic transactions are specified in terms of the number of bytes per packet, number of packets per transmission, and the transmission interval (in *frames* for low- and full-speed devices, or *micro-frames* otherwise). The scheduling problem is to ensure that transactions do not cross micro-frame boundaries for high or super-speed devices, and no more than eight micro-frames worth of transactions occupy each frame. The schedule must map transactions to frames and micro-frames so that intervals and transmission delays are guaranteed for periodic transmissions. Essentially, this is a bin-packing problem, with EHCI allowing software to construct the periodic and asynchronous schedules.

The **extensible Host Controller Interface** (**xHCI**) [85] is intended to be a replacement for EHCI, providing better power efficiency, performance, and new capabilities in USB 3.x. The host memory data structures differ significantly between EHCI and xHCI. The periodic frame list and circular asynchronous list in EHCI are no longer exposed to the system software. They are replaced with ring buffers in xHCI with data fields that influence how the frame list and the asynchronous list are formed by the host controller. An **extensible host controller** (**xHC**) manages three types of rings in host memory: (1) a single *command* ring, (2) an *event* ring, and (3) a separate *transfer* ring for each device endpoint. The command ring passes requests from the host system software to the controller. The event ring returns status information, and results of transfers to system software. Finally, each transfer ring contains a set of ***transfer descriptors*** (**TDs**) that consist of one or more ***transfer request blocks*** (**TRBs**). TRBs contain the data to be transferred in a given request. In the case of periodic endpoints, TRBs also contain a frame ID field. This allows periodic transfers to specify the starting frame ID, at the granularity of 1ms. For transactions requiring service

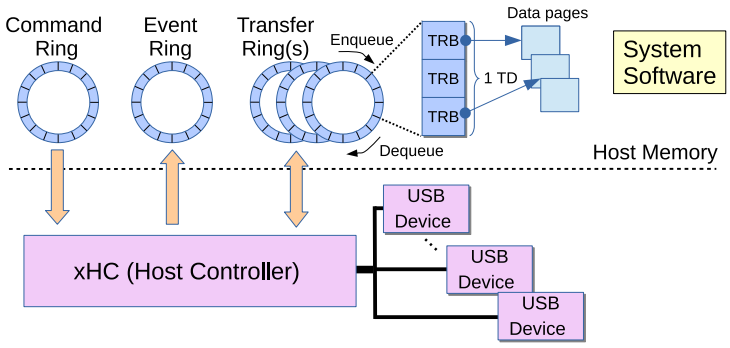

Fig. 2. The data organization of xHCI.

intervals of one or more micro-frames, it is possible to pack multiple isochronous or interrupt TDs in the same frame by specifying a common starting frame ID. Aside from this level of software control, all transfer rings are processed by the xHC. Figure 2 illustrates various transfer rings and buffers in an xHCI.

The xHCI Specification mandates multiple USB **bus instances** (**BIs**) be implemented by xHCs (USB 3.x host controllers) to provide support for the previous generations of the USB protocol. For example, a USB 3.0 host controller provides at least one USB 3.0 BI operating at 5.0 Gbps, a USB 2.0 BI at 480 Mbps, a USB 1.1 BI at 12 Mbps, and one USB 1.0 BI clocked at 1.5 Mbps. These BIs feature dedicated hardware logic for USB traffic scheduling, FIFO buffers, and transceivers. However, the host controller driver accesses the BIs via the same Transfer Ring interface without explicitly handling TRB to BI routing. Once the host controller driver enumerates a USB device on a port, the xHC will assign the device to the corresponding BI and handle the routing internally.

A noteworthy feature introduced by USB 3 is a **super-speed debugging capability** (**xDbC**) built in every standard USB 3 host controller. It provides a means to connect two systems via a cross-over cable[1] where one system is a Debug Host and the other a Debug Target (System Under Test). On the Debug Target's side, xDbC appears as a PCI device under xHC with two independent super-speed bulk transfer endpoints. On the Debug Host, xDbC appears as a USB 3.0 device with two bulk endpoints for communicating with the Debug Target. OSs such as Windows and Linux use xDbC as a serial debug interface, similar to a traditional UART. However, as we discuss in Section 4, it is possible to use xDbC as a high-speed host-to-host communication link, which can outperform conventional Gigabit Ethernet. Using our proposed USB 3.x real-time scheduler, xDbC and other USB host-to-host communication bridge devices are able to achieve predictable data transfer latency and throughput.

3 USB 3 BUS SCHEDULING

3.1 Hardware Bus Scheduler

xHCI [85] uses two schedulers per BI: a bandwidth-preserving periodic scheduler for isochronous and interrupt endpoints, and a round-robin scheduler for bulk and control endpoints. The system software places TDs on the ring buffers of endpoints, and notifies the host controller to assign each TD to one of the schedulers for the corresponding BI, based on the type of the endpoint. The host controller breaks down each TD into multiple USB packets to transfer. The exact size of each USB

[1]A USB cross-over cable features two Type-A USB connectors at each end. The VCC signal is disconnected as the cable is supposed to connect to host controllers that supply current. The receive pair of one connector is attached to the transmit pair of the other connector, and vice versa.

packet depends on the implementation of the downstream device, however, the maximum packet size is dictated by the USB standard [83]. USB 3.x control endpoints are limited to packets of up to 512 bytes (payload), while other types of endpoints have a maximum packet size of 1024 bytes.

The host controller schedules one quantum (burst of packets) for each endpoint during each micro-frame before switching to another endpoint. Equation (2) represents the quantum size of a super-speed endpoint, where q_i is the maximum number of bytes transferred for endpoint i before the xHC switches to another endpoint. In the same equation, b_i is the maximum packet size (in bytes) supported by the endpoint, n_i corresponds to the maximum burst size (packets minus one), and m_i is the number of consecutive bursts (minus one) transferred for endpoint i.

$$\begin{aligned} q_i &\leq (m_i + 1)(n_i + 1)b_i \\ 0 &\leq n_i \leq 15 \quad (Isochronous\ \&\ Bulk) \\ 0 &\leq m_i \leq 2 \quad (Interrupt\ \&\ Isochronous) \\ n_i, m_i &= 0 \quad (Others) \end{aligned} \tag{2}$$

Asynchronous endpoints are characterized by their quantum size, and the host controller treats them as best-effort traffic. However, periodic endpoints are scheduled in a time-triggered manner. This means that for each endpoint i, one TD of up to q_i bytes is schedulable every period, T_i. The period of an isochronous or interrupt endpoint is of the form $T_i = 2^{\delta_i - 1} \times \mu frame$ nanoseconds, where $1 \leq \delta_i \leq 16$.[2] The parameters b_i, n_i, m_i and δ_i are communicated to the system software when the USB device is enumerated by the host controller driver.

During each micro-frame, the host controller scheduler transfers one quantum of each periodic endpoint with available TDs, and then schedules asynchronous traffic once all periodic TDs allocated to the micro-frame are transferred, or the utilization of the micro-frame time by periodic TDs reaches 90%. The asynchronous TDs are transferred according to a round robin scheduler until the end of the current micro-frame.

The aforementioned parameters determine the amount of data that must be exchanged before the host controller services another endpoint. However, to understand the endpoint service times and switching points in a schedule, it is important to derive the transfer latency of USB packets and express q_i in time units (e.g., nanoseconds).

Equation (3) represents a generalized formulation for USB packet transfer latency, derived from various USB standards. In the equation, w_i denotes the worst-case transfer latency (in nanoseconds) of a packet of b_i bytes for endpoint i. h denotes the host controller's delay, and p represents the delay induced by the protocol overheads. As defined by the USB Specification, the value of h is the time required for the host to prepare for or recover from a transmission and is implementation-specific. We assume this value to be 5 nanoseconds, as is typically used by Linux kernels. This value is sufficient to account for variations in delays exhibited by different host controllers, and is independent of the host processor speed. The protocol overheads denoted by p include **synchronization** (**SYNC**) transfers, **per-packet identifiers** (**PIDs**), **End-of-Packet** (**EOP**) markers, **cyclic redundancy checks** (**CRCs**), as well as inter-packet gaps and bus turnaround times for multi-packet transactions. From the xHCI Specification [85], bulk and interrupt transactions have protocol overheads corresponding to the delay in transferring 48 bytes for **outgoing** (**OUT**) transfers, and an additional 36 bytes for simultaneous **incoming** (**IN**) transfers. Isochronous transactions have protocol overheads corresponding to the transfer delay of 32 bytes for OUT plus 16 bytes for IN transfers.

[2] δ_i is referred to by the term "bInterval" for the corresponding endpoint in the USB Specification. This one-byte value is restricted to $1 \leq bInterval \leq 16$ for periodic endpoints, with bInterval used as the exponent for a $2^{(bInterval-1)}$ number of micro-frames.

In the equation, α is the latency of transferring a symbol (i.e., one bit of data), which is derived from the inverse of the clock frequency of the BI's transceiver. For example, $\alpha = 0.2$ nanoseconds for a USB 3.0 BI at 5 Gbps. USB 2.0 and below use bit stuffing every 6 consecutive 1 symbols to ensure synchronized bit timing. The constant $O_{bits} = 19/6$ accounts for worst-case bit stuffing overheads of the required 8-bit packet ID, 7-bit address field, and 4-bit endpoint number. The USB 3.x Specification does not mention bit stuffing but we include it here for compatibility with other USB protocols, as we are interested in determining a worst-case time bound for packet transfers.

γ factors the cost of host-device bit-level SYNC on the packet payload. For USB 2.0 and below, $\gamma = 7/6$ to account for the worst-case added stuffing bit every 6 consecutive 1 symbols. For USB 3.x, we assume $\gamma = 1$, as it appears the protocol uses a one-to-one payload scrambling algorithm instead. From Equation (3) it is then possible to derive the transfer latency for a quantum q_i of packets, each with b_i bytes, for burst parameters n_i and m_i greater than 0.

$$w_i = h + p + (\alpha \times \lfloor O_{bits} + 8\gamma b_i \rfloor) \tag{3}$$

3.2 Scheduling Model and Timing Analysis

We define three sets of endpoints for the BI of interest: (1) the set of asynchronous endpoints, A, (2) high-criticality periodic endpoints, H, and (3) low-criticality periodic endpoints, L. Let the tuple $< B_i, T_i >$ be the budget (in bytes) and period (in micro-frames) of endpoint i specified by the USB device driver at the time of device initialization. The objective is to ensure all endpoints $i \in \{A \cup H\}$ are always capable of transferring their budget (B_i) in every period (T_i), as long as there is enough data in the host memory or the downstream device buffer to send or receive. However, endpoints in L only receive their budget in each period, provided there is enough remaining bandwidth after all high-criticality endpoints (in $\{A \cup H\}$) reserve their budgets.

Our approach to finding an assignment of USB transactions is to use a heuristic that parallels the **first-fit decreasing (FFD)** algorithm for bin-packing [24] and rate monotonic scheduling [71]. We build on our earlier work [59], which is shown to outperform Linux's first-fit algorithm for USB 2 traffic, to derive a solution for USB 3 scheduling.

Section 3.4 presents a slightly modified version of the FFD algorithm to provide bandwidth preservation for interrupt and isochronous endpoints using USB 3. However, the goal of this article is to ensure bandwidth preservation for asynchronous endpoints as well as highly critical periodic endpoints. One reason for this is that many devices that require service guarantees only support asynchronous endpoints, such as debug hosts, which are useful for host-to-host communication, or USB-CAN interfaces, which require control traffic to be handled by real-time host-level services.

Since the hardware scheduler transfers TDs from endpoints in H and L (i.e., the periodic endpoints) before scheduling endpoints in A, we build our mixed-criticality solution based on our earlier bin-packing algorithm as follows:

Every time a new USB device is enumerated and set to be scheduled on the BI:

(1) the minimum micro-frame time t_A required for all endpoints in A to meet their budget requirements is computed.
(2) An array of N micro-frame bins is allocated, with each bin's initial capacity set to $\mu frame - t_A$ nanoseconds. N is the least common multiple of periods of all endpoints in $\{H \cup L\}$. Each bin keeps track of the available micro-frame time for periodic traffic.
(3) The micro-frames are then filled with TDs from the endpoints in H. The schedule fails if not all TDs in H fit while respecting their budget and period requirements. Upon a successful return from the bin-packing algorithm, the capacities of each bin are reduced as a result of preserving bandwidth for the highly critical endpoints.

(4) Finally, the bin-packing algorithm is called once more to try to schedule TDs from endpoints in L in the remaining time within each bin. This ensures that low-criticality traffic will be scheduled by the xHC only when all high-criticality traffic is guaranteed its bandwidth. Given that t_A is reserved for every micro-frame, a feasible schedule ensures asynchronous traffic is guaranteed predictable service.

ALGORITHM 1: Quest USB Bus Scheduling

```
A ← (input) set of asynchronous endpoints
H ← (input) set of high-criticality periodic endpoints
L ← (input) set of low-criticality periodic endpoints
// (1) Calculate the reservation required for the asynchronous traffic (Algorithm 2)
t_A ← usb_sched_async(A, H)
if t_A < 0 then
    return FALSE
end if
// (2) Allocate and initialize the bins for the periodic traffic scheduling
N ← LCM({T_i ∪ T_j : i ∈ H, j ∈ L})
M ← alloc(N integers)
for f = 0 to N − 1 do
    M[f] ← μframe − t_A
end for
// (3) Allocate bins for the highly critical periodic traffic (Algorithm 3)
(ret, M) ← usb_sched_periodic(H, M, N, TRUE)
if ret = TRUE then
    // (4) Allocate bins for the low-criticality periodic traffic (Algorithm 3)
    M ← usb_sched_periodic(L, M, N, FALSE)
    return TRUE
end if
return FALSE
```

In Step (1), t_A is derived by calculating the maximum number of micro-frames an asynchronous endpoint $j \in A$ requires to successfully transfer its B_j bytes of data in a time less than or equal to T_j. During each pass of the round-robin scheduler for asynchronous endpoint j, one burst of packets having q_j bytes is transferred. Each burst consists of $(n_j + 1)$ packets, and each packet takes w_j nanoseconds to transfer. Therefore, endpoint j needs to be visited by the xHC round-robin scheduler k_j times to transfer B_j bytes, as shown in Equation (4).

$$k_j = \left\lceil \frac{B_j}{(n_j + 1)b_j} \right\rceil \tag{4}$$

For simplicity, consider a scenario in which there are no periodic endpoints associated with a BI, and thus all the bandwidth goes to the asynchronous endpoints in A. In the worst case, all asynchronous transfer requests arrive at the same time, and the endpoint of interest, j, always receives its quantum after every other endpoint. Therefore, the worst-case latency of endpoint j is simply the sum of transfer times of all asynchronous endpoints for the duration of k_j passes through the round-robin schedule. Equation (5) shows l_j as the time it takes for the host controller to transfer one full budget, B_j, of payload for endpoint j in the worst-case scenario under this assumption.

$$l_j \leq k_j \sum_{i=1}^{|A|} (n_i + 1)w_i \tag{5}$$

Now consider the addition of periodic endpoints associated with a BI. We must first address the initial time within each micro-frame where only periodic TDs are scheduled, in addition to the worst-case transfer time of an asynchronous endpoint as shown in Equation (5). To address this scenario, we assume that at least t_A nanoseconds surplus time in every micro-frame after all periodic TDs are scheduled is reserved for asynchronous transfers. Therefore, the worst-case time it now takes to transfer B_j bytes for asynchronous endpoint j is c_j micro-frames, as shown in Equation (6).

$$c_j = \left\lceil \frac{1}{t_A} k_j \sum_{i=1}^{|A|} (n_i + 1) w_i \right\rceil \tag{6}$$

It follows that the worst-case time to transfer B_j bytes for endpoint j is $\mu frame \times c_j$ nanoseconds.

3.3 Asynchronous Reservation

Given a group of asynchronous endpoints A and periodic endpoints H, we can find a range of possible values for t_A. We first derive t_{Amax} to reject sets of asynchronous endpoints that are not schedulable, when $\exists i \in A | c_i > T_i$. According to the USB 3.x Specification, periodic transfers take precedence for up to 90% of each micro-frame. Assuming there are $|H|$ periodic endpoints present in every micro-frame, which each transfer $(m_i + 1)(n_i + 1)b_i$ bytes, then the upper bound for t_A is

$$t_{Amax} = max \left\{ \mu frame - \sum_{i=1}^{|H|} (m_i + 1)(n_i + 1) w_i, 0.1\ \mu frame \right\} \tag{7}$$

t_{Amax} is at least $0.1\ \mu frame$ because of the amount of time within each micro-frame reserved for periodic transfers. If the micro-frame time consumed by high-criticality periodic requests in H is less than 90%, then t_{Amax} may be higher than $0.1\ \mu frame$. Once t_{Amax} is established, then the minimum feasible t_A ensures that every endpoint $i \in \{A \cup H\}$ transfers at least B_i bytes every T_i micro-frames. The minimum feasible t_A yields the smallest reservation necessary for asynchronous traffic, while maximizing the endpoints in L that are schedulable. Algorithm 2 determines whether a feasible t_A exists that guarantees service to endpoints in A and H while minimizing rejections of endpoints in L.

- Lines 4 to 6 compute k_j as defined in Equation (4) and are used in Equation (6) for each endpoint $j \in A$. The k_j values only change with a change in the asynchronous pool, therefore, it can be calculated once at the beginning of the algorithm. Any change in the asynchronous pool should invoke Algorithm 2 again.
- Line 10 computes the amount of time, t_p, the xHC spends on the periodic schedule at the beginning of each micro-frame, given only that highly critical endpoints are present.
- Lines 11 to 18 of the Algorithm 2 incorporate t_P into Equation (7) to calculate t_{Amax}, update the transfer latency c_j of each endpoint $\in A$, and verify the asynchronous admission criteria below:

$$\forall i \in A : c_i \leq T_i \tag{8}$$

- Then, if the resulting t_{Amax} is enough for all endpoints in A to transfer their B_i bytes every T_i, the algorithm proceeds with finding the minimum possible value for t_A by increasing the micro-frame time from the minimum asynchronous reservation in `step` increments (e.g., $\mu frame \times 10\%$ nanoseconds), as shown in lines 20 to 32.

The `step` value is a multiple of the smallest quantum size (see Equation (2)) among the endpoints associated with the BI for which we are establishing a schedule, as bus transfers are considered non-preemptive.

ALGORITHM 2: Asynchronous Reservation (t_A) Computation—**usb_sched_async()**

```
1: A ← set of Asynchronous endpoints
2: H ← set of high-criticality Periodic endpoints
3: step ← incremental step (in ns)
4: for j = 0 to |A| − 1 do
5:     A[j].k ← ceil(A[j].B / ((A[j].n+1)A[j].b)) // Equation (4)
6: end for
7: // Find latency of one pass over the RR schedule
8: t_r ← Σ_{i=0}^{|A|−1} (A[i].n + 1)A[i].w
9: // Find the maximum feasible t_A
10: t_P ← Σ_{i=0}^{|H|−1} (H[i].m + 1)(H[i].n + 1)H[i].w
11: t_Amax ← max{0.1 μframe, μframe − t_P} // Equation (7)
12: // Test the feasibility of t_Amax
13: for j = 0 to |A| − 1 do
14:     A[j].c = ⌈(A[j].k / t_Amax) t_r⌉ // Equation (6)
15:     if A[j].c > A[j].T then
16:         return −1 // not feasible
17:     end if
18: end for
19: // Find the minimum feasible t_A
20: for t_Amin = 0.1 μframe to t_Amax with step do
21:     feasible ←true
22:     for j = 0 to |A| − 1 do
23:         A[j].c = ⌈(A[j].k / t_Amin) t_r⌉ // Equation (6)
24:         if A[j].c > A[j].T then
25:             feasible ←false
26:             break
27:         end if
28:     end for
29:     if feasible then
30:         return t_A ← t_Amin
31:     end if
32: end for
33: return −1 // not feasible
```

3.4 Periodic Reservation

Algorithm 3 is provided with an array of periodic endpoints assigned to set P. It is first called with $P = H$ and then with $P = L$. Each endpoint, P_i, is specified with a packet transmission quantum of q_i bytes, and period T_i in micro-frames. A successful assignment of micro-frames to the endpoint i will start in micro-frame $f = j$ and proceed to be serviced in micro-frame $f = (j + a{\cdot}T_i)\%N$, where $a = 1, 2, \ldots$.

As shown in our earlier USB 2.0 scheduling work [59], sorting endpoints by increasing interval, with ties being broken by servicing the largest transmission delay first, tends to increase the likelihood of finding successful schedules. The alternative would be to exhaustively consider all possible permutations of endpoints, which is impractical for an online scheduler.

4 HOST-TO-HOST COMMUNICATION OVER USB 3

Real-time distributed and embedded systems often require data to be processed and distributed across more than one host. For example, an autonomous **vehicle management system** (**VMS**) might feature an array of cameras, LIDARs, and ultrasonic sensors that are connected to more than one host, which share the data processing load. Pipelines of tasks spanning different machines perform sensing, processing, control, and actuation.

ALGORITHM 3: Periodic Reservation Algorithm—**usb_sched_periodic()**

```
1: P ← (input) array of periodic endpoints fed to this algorithm
2: M ← (input/output) micro-frame bins
3: N ← (input) number of the bins
4: h ← (input) a boolean value indicating that P = H
5: // M[f]: The available transfer time in the micro-frame f
6: // P[i].s: The starting micro-frame allocated for the periodic endpoint i
7: // Sort endpoints by smallest period (T) first, breaking ties by largest transmission delay first
8: P ← SORT(P, T)
9: for i = 0 to |P| − 1 do
10:    P[i].s ← −1
11:    q ← (P[i].m + 1)(P[i].n + 1)P[i].w
12:    j ← 0
13:    while P[i].s = −1 ∧ j < P[i].T do
14:       feasible ← TRUE
15:       f ← j
16:       while f < N do
17:          if M[f] < q then
18:             feasible ← FALSE
19:          end if
20:          f ← f + P[i].T
21:       end while
22:       if feasible then
23:          P[i].s ← j // TDs of endpoint i start from micro-frame j
24:          f ← j
25:          while f < N do
26:             M[f] ← M[f] − q // Update bin's remaining capacity
27:             f ← f + P[i].T
28:          end while
29:       else
30:          j ← j + 1
31:       end if
32:    end while
33:    if P[i].s = −1 ∧ h then
34:       // Could not fit the highly critical endpoint i
35:       return FALSE
36:    end if
37: end for
38: return (TRUE, M)
```

CAN communication is an example fieldbus technology that is typically used in automotive systems but lacks the bandwidth capabilities of USB. USB 3.x supports host-to-host communication using the xHCI built-in debug capability known as xDbC. A Cross-over cable enables bulk transfers between a Debug Host and a corresponding Debug Target machine that appears as a device. Alternatively, a pair of hosts are able to exchange data via USB using an active bridge such as PL27A1 [31], or by connecting to a machine with a built-in **On-The-Go** (**OTG**) device controller.

In this article, we focus on the use of xDbC and active bridges for USB networking. OTG controllers are less common on embedded PCs, instead being more prevalent on tablets and smartphones. Single-board computers such as the UP Squared work with the Quest RTOS and include OTG device controllers. However, our experiences with the UP Squared suggest there are hardware limitations that preclude the use of super-speed (gigabit per second) transfer rates. Instead, OTG-based host-to-host transfers fall back to high-speed (several hundred megabit per second) rates. Nonetheless, if super-speed OTG is made available on a larger set of devices it has potential

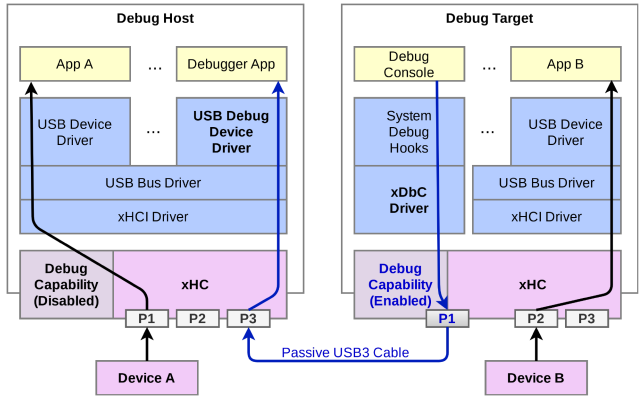

Fig. 3. xDbC/xHCI stacks in the debug host and debug target.

benefits over xDbC. Whereas xDbC provides only asynchronous endpoints, OTG supports periodic transfers. This potentially simplifies the scheduling of transactions as there would be no need to support bulk transfer guarantees for host-to-host communication as described in Section 3.

While some embedded systems are equipped with one or more Gigabit Ethernet interfaces, utilizing USB 3.x as the host-to-host communication medium has the following advantages:

— Direct connectivity between hosts is possible without the need for separately powered switches and hubs, as used by networks such as Ethernet. This saves space, cost, and energy by avoiding extra hardware and wiring.
— USB 3.0 xDbC and bridge cables offer transfer rates above 1 Gbps, exceeding the typical rate limit of most embedded Ethernet interfaces.
— Host-level USB 3.x scheduling is able to ensure predictable throughput guarantees.
— USB networking and device I/O are unified, meaning the same software stack within the OS is used for scheduling I/O and communication requests. There is no need to implement host-level protocol converters or bridges that translate USB packets into other formats (e.g., Ethernet frames) suitable for network communication. Pipelines of tasks spanning multiple hosts are then able to be scheduled in coordination with network transfers and device I/O.

4.1 xHCI Debug Capability for Networking

Figure 3 shows the software structure and physical connectivity between a Debug Host (master) and Debug Target (slave). The slave enables the Debug Capability of its USB 3.x host controller via an xDbC driver. Once the master senses connection to another USB 3.x host controller, it assigns control to a Debug Device Driver. This is shown in the figure with port P3 in the master connected to P1 in the slave via a passive USB cross-over cable. The Target then appears as having a super-speed USB 3.x link to the Debug Host.

The xHCI hardware allows a single machine to act as both a Debug Host and Target for two different remote machines. This allows the re-purposing of xDbC as a fully-duplex host-to-host communication link, with network topologies involving more than two host computers such as trees and stars.

USB communication using xDbC is readily available within xHC hardware. However, the following limitations must be considered:

(1) xDbC connections using passive cross-over cables that do not strengthen or repeat the signal are limited to 3–5 meters, based on the USB signaling standard.

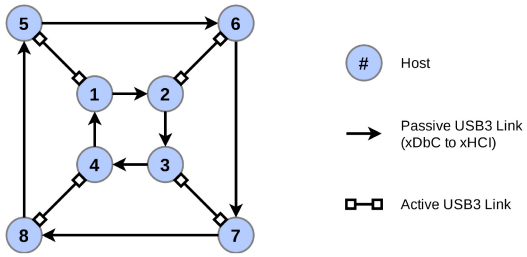

Fig. 4. A 3-D hypercube using xDbC and active USB bridge links.

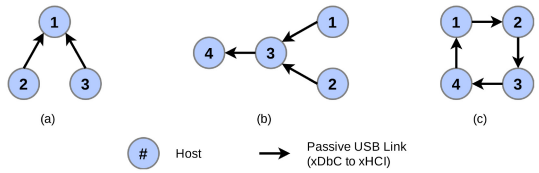

Fig. 5. Examples of USB network topologies using xDbC links.

(2) Each xHC only includes one xDbC controller, and therefore, topologies such as hypercubes that require more connections than hosts are not implementable by solely relying on xDbC links.
(3) USB is a master-slave protocol leading to asymmetric rather than peer connections between hosts. Establishing xDbC connections on a Debug Host must be ordered to achieve the desired topology.

4.1.1 Limitations 1 and 2. The first two issues are mitigated by the use of USB 3 bridge devices (active links). PL27A1 for example, is a USB-A to USB-A cable with a chipset in the middle. The PL27A1 chipset consists of two USB devices, each of which is enumerable by only one of the hosts connected by the bridge. The two USB devices share hardware buffers for data exchange between the host computers; the input buffer of one side is shared with the output buffer of the other side, and vice versa. The signal amplification performed by PL27A1 addresses Limitation 1. Moreover, one can use as many USB 3 active links as there are USB 3 ports in the system, and address Limitation 2 at the cost of about 190 mW per active link [31].

Figure 4 shows the implementation of a 3D hypercube using 8 xDbC and 4 PL27A1 links. The arrows represent xDbC links between two hosts. The tail of each arrow is connected to the machine that enables its xDbC controller and presents it as a USB device to the other machine attached to the head of the arrow. Using this representation, the second limitation translates to not allowing an out-degree of more than one.

It should be noted that a passive xDbC cable connection between a pair of hosts is less expensive than a PL27A1 or similar bridge cable, making it desirable to use as many xDbC connections as a given topology will allow. Additionally, xDbC connections support two endpoints for both inward (IN) and outgoing (OUT) communication, so separate cables for full-duplex communication are not needed. However, there are limitations to the complexities of network topologies handled by xDbC. If more complex network configurations are required, a combination of xDbC and active bridge connections makes sense, with the aim of using as few bridges as possible.

4.1.2 Limitation 3. Figure 5 shows several example networks that rely only on xDbC links, and which require ordered operations to form successful topologies. We assume that system designers

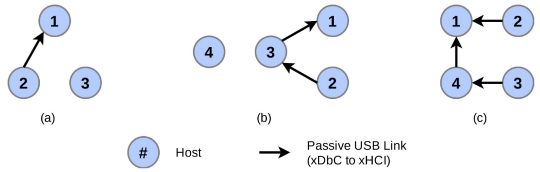

Fig. 6. Examples of misconfigured USB network topologies using xDbC links.

specify (e.g., at boot time) which hosts must enable their xDbC controllers. However, a Debug Target must enable its xDbC controller before the Debug Host is allowed to proceed with USB 3 device enumeration. The proper order of the enumeration (`xHC.enum`) and enable (`xDbC.enable`) events is shown below, for cases (a)–(c), with $event_i \rightarrow event_j$ indicating $event_i$ happens before $event_j$. Note that all hosts must enumerate their attached devices for I/O operations, but enabling xDbC is optional and depends on the desired network topology.

— (a) Hosts 2 and 3: `xHC.enum` → Hosts 2 and 3: `xDbC.enable` → Host 1: `xHC.enum`
— (b) Hosts 1 and 2: `xHC.enum` → Hosts 1 and 2: `xDbC.enable` → Host 3: `xHC.enum` → Host 3: `xDbC.enable` → Host 4: `xHC.enum`
— (c) Host 1: `xHC.enum` → Host 1: `xDbC.enable` → Host 2: `xHC.enum` → Host 2: `xDbC.enable` → Host 3: `xHC.enum` → Host 3: `xDbC.enable` → Host 4: `xHC.enum` → Host 4: `xDbC.enable` → Host 1: `xHC.enum`

Should hosts not follow the correct order of enumeration and xDbC enable events, the final connections may result in an unwanted network configuration. Figure 6 shows examples of how each of the three networks in Figure 5 may end in a partially disconnected state.

In Figure 6(a), Host 1 performs USB enumeration after Host 2 enables its xDbC controller, but before Host 3 enables its controller. Situations like this are easily resolved if the hosts repeat the enumeration of xDbC-connected ports for a set amount of time. In Figure 6(b), a connection is established between Hosts 1 and 3 rather than 3 and 4, if Host 1 enumerates the xDbC previously enabled by Host 3. In this case, Hosts 3 and 4 will not be linked unless Host 4 also enables its xDbC and it is enumerated by Host 3. This results in one more xDbC being enabled than necessary, which consumes extra power, assuming the xDbC of Host 1 is enabled and not used. Finally, Figure 6(c) shows a situation where the xDbCs of Hosts 2 and 3 are enabled and subsequently enumerated by Hosts 1 and 4, respectively, leaving a missing link between a pair of hosts wishing to form a circular network topology as shown in Figure 5(c).

The correct order of `xDbC.enable` and `xHC.enum` steps is ensured if each host is first provided with a list of xDbC dependencies. One possible approach is to assign these dependency lists to hosts at boot time.

Let P_i be a dependency list assigned to host h_i. Each host $h_j \in P_i \mid j \neq i$ enables its xDbC controller for connection to host h_i. Algorithm 4 shows how the xDbC dependencies are processed to ensure the correct order of enumeration for the desired network topology. After initializing xHCI (Line 5), xDbC is enabled locally if the current host is the designated leader and the flag to enable xDbC is true (Lines 6–8). In all other cases, the current host waits for device attachments from each host in its xDbC dependency list (Lines 9–13). The local host then performs xHCI enumeration (Line 14) and subsequently enables its xDbC, if required and it is not the leader (Lines 15–17). The use of a leader is to break cycles in network topologies.

In Figure 5(a), $P_1 = \{2, 3\}$ and $P_2 = P_3 = \{\}$. This prevents Host 1 from proceeding to enumeration until it has successfully sensed the xDbC controllers of Hosts 2 and 3. In Figure 5(b), $P_1 = P_2 = \{\}$, and, therefore, they enumerate and then enable their xDbC controllers first. Since $P_3 = \{1, 2\}$, Host

ALGORITHM 4: xDbC Network Initialization

```
1: i ← (input) the index of the current host
2: e ← (input) flag indicating whether or not xDbC must be enabled
3: P ← (input) dependency list for the current host
4: LEADER ← (input) the index of the host that enables xDbC first to avoid a deadlock
5: Initialize xHCI
6: if e = TRUE ∧ i = LEADER then
7:     xDbC.Enable // Enable xDbC for enumeration by other hosts
8: end if
9: for j = 0 to |P| − 1 do
10:    while xHCI.deviceAttached(P[j].host) = FALSE do
11:        Wait // Wait for attachment of xDbC from host h_j ∈ P_i
12:    end while
13: end for
14: xHCI.Enumerate
15: if e = TRUE ∧ i ≠ LEADER then
16:    xDbC.Enable // Enable xDbC if not a Leader
17: end if
18: return
```

3 only enables its xDbC once Hosts 1 and 2 are done with their enumeration of USB devices. As $P_4 = \{3\}$, Host 4 proceeds with enumeration once Host 3 has finished network initialization. In Figure 5(c), assigning $P_1 = \{4\}, P_2 = \{1\}, P_3 = \{2\}$, and $P_4 = \{3\}$ results in a deadlock if there is no leader. However, if one of the hosts is designated the leader it will enable its xDbC first, and the ring topology will be correctly formed.

5 TUNED PIPES PROGRAMMING INTERFACE

Sections 3 and 4 show how to schedule and manage USB 3.x endpoints associated with I/O devices and network connections. A bus scheduler is rendered ineffective if the host does not coordinate the execution of tasks and interrupt handlers associated with device I/O and network traffic. We address this problem using an abstraction known as a "tuned pipe".

A tuned pipe is a host-to-device communication channel that has throughput and delay bounds. Abstractly, it encompasses the control and data path necessary to execute the code that moves data between a user-space memory buffer and the device.

In the Quest RTOS, a tuned pipe is built on a device *endpoint* abstraction, which comprises an I/O VCPU and an optional Main VCPU, as well as kernel buffers used by the corresponding device driver. A tuned pipe extends an endpoint into a user-space thread, which runs an application-specific function to pre- or post-process data. Pre-processed data is sent through the tuned pipe to the endpoint for delivery to the device. Post-processed data is input from a device through its endpoint into a user-level address space.

Figure 7 shows the tuned pipe abstraction. Data flows between a user-space pipe buffer and an endpoint buffer. Data in the endpoint buffer may correspond to multiple pipes. Control flow for a tuned pipe encompasses the execution of a user-level thread associated with a Main VCPU and one or more threads in the device endpoint. A device endpoint has at least one thread to perform bottom half interrupt handling, and an optional thread to parse and process the data in the endpoint buffer.

5.1 Tuned Pipes API

Quest's tuned pipe API allows user-space programs to establish host-to-device communication channels with throughput and delay constraints. The pipe is guaranteed to be temporally isolated

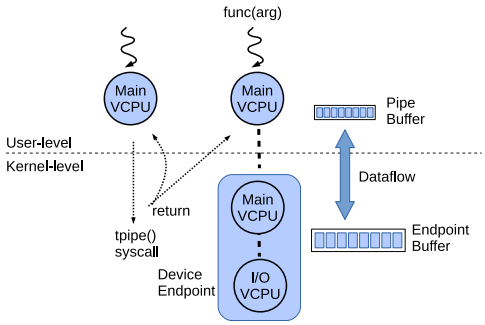

Fig. 7. Tuned pipe abstraction.

from the activities of other tasks and I/O pipes. The creation of a tuned pipe is established by a call to `tpipe`:

```
int tpipe(endpoint_t end, qos_t qspec,
        (void *)(*func)(void *), void *arg);
```

On success, `tpipe` returns an integer pipe identifier, otherwise, it returns −1. A newly-created tuned pipe spawns a user-level thread that is mapped to a Main VCPU and bound to a device endpoint. Figure 7 shows the binding of a Main VCPU to a device endpoint, which is associated with an endpoint type, as follows:

```
typedef struct {
  vcpu_id_t iovcpuid; // Bottom half VCPU ID
  vcpu_id_t mvcpuid;  // Main VCPU ID
  struct sched_param *iovcpu_params;
  struct sched_param *mvcpu_params;
  endpoint_attrs_t *eattrs; // Attributes
} endpoint_t;
```

An endpoint's I/O VCPU (identified by `iovcpuid`) provides budgeted CPU time for a bottom half device driver. The scheduling parameters of the I/O VCPU (`iovcpu_params`) are dependent on the device capabilities (e.g., how much data it is able to transfer in a given time interval), and the requirements of the tuned pipes associated with that endpoint. Depending on the device, multiple tuned pipes may be associated with a single endpoint. For example, a USB-CAN device might expose up to five separate channels and, hence, five separate pipes for its endpoint. This information is provided by a device driver information base when the driver is registered with the OS, and attributes in this information base are accessed through the `eattrs` member of the endpoint type, which is partially described as follows:

```
typedef struct {
  int max_channels;  //Max pipes for endpoint
  uint64_t max_tput; //Max bits per min_latency
  time_t min_latency;//Min delay [nanoseconds]
  int min_ebufsz; //Min endpoint buffer size
  int max_ebufsz; //Max endpoint buffer size
  int min_pktsz;  //Min transfer packet size
  int max_pktsz;  //Max transfer packet size
  ...
} endpoint_attrs_t;
```

In the above, `max_channels` is the maximum number of channels supported by the endpoint for the creation of unique pipes. The ratio $\frac{\texttt{max\_tput}}{\texttt{min\_latency}}$ is the highest sustainable throughput

achievable by the endpoint, and is limited by the device bandwidth. `min_latency` is the minimum delay between successive data items transferred between the host memory and the device. A device is not capable of reading or writing data faster than this time. `[min|max]_ebufsz` is the [minimum|maximum] endpoint buffer size, and `[min|max]_pktsz` is the [minimum|maximum] size of a packet transferred to or from the device.

The key attributes within `struct sched_param` are the corresponding VCPU's budget, C, and period, T. Depending on the device driver, an endpoint might use a Main VCPU (identified by `mvcpuid`) in addition to an I/O VCPU. We use such a configuration in the implementation of our USB-CAN driver, to parse incoming packet data in an endpoint buffer and associate it with separate pipe buffers.

A device driver developer establishes the default scheduling parameters for the endpoint I/O VCPU and, if it is used, the Main VCPU also. Depending on the tuned pipe requests from user-space, the endpoint's VCPU scheduling parameters, including both budget and period, might be adjusted from their defaults. They will nonetheless be constrained by the capabilities of the device. For example, suppose a USB-CAN device exposes five channels of up to 2.25 Mbps, such that four channels are limited to 500 Kbps and one is limited to 250 Kbps; for an endpoint with a 4 KB buffer, a Main VCPU thread must process the buffer every 14 ms to avoid overflow. If a device driver developer establishes the processing overhead is no more than 2 ms, then the endpoint Main VCPU budget and period are set to $C = 2$ ms and $T = 14$ ms, respectively.

The creation of a tuned pipe associates a thread function (`func`) and its argument (`arg`) with a new Main VCPU. This is similar to the semantics of thread creation APIs such as the POSIX `pthread_create` call. The difference is that the new thread in a tuned pipe is mapped to a time-constrained VCPU whose budget and period are automatically established to guarantee the QoS specified by the `tpipe` call. The QoS specification, `qspec`, is of the following type:

```
typedef struct {
  time_t latency;   // Nanoseconds
  uint64_t tput;    // Bits per given latency
  size_t IObufsz;   // Pipe buffer size in bytes
  time_t texec_time;// Thread exec time
} qos_t;
```

The `texec_time` is the thread (`func`) execution time to process `IObufsz` bytes of data in a given period of the user-level Main VCPU. This time is assumed to be determined by prior measurements of the data processing delay. As an example, suppose a user wishes to process a pipe buffer containing up to 128 bytes of packet data within 1 ms of its arrival on a 500 Kbps pipe associated with a device endpoint. `texec_time` is set to 1ms and assumed to be sufficient to accommodate the execution time of `func`. Along with `IObufsz`=128 bytes, `latency` is set to 1×10^9 nanoseconds, and `tput` is set to 512×10^3 bits/second. The Main VCPU associated with the thread `func` has an automatically-generated budget, $C = 1$ ms and period, $T = 2$ ms. The period is derived by the `tpipe` function using Little's Law, $L = \lambda W$, where L is the buffered data (here, 128 bytes), λ is the arrival rate (here, no more than 500 Kbps), and W is the time for the buffer to be filled before being reused. W effectively bounds the period of the Main VCPU. In this case, $W = T = 2$ ms, ensures the pipe buffer is processed before overflowing.

It should be observed that the constraints on a Main VCPU associated with a tuned pipe are limited by the capabilities of the device endpoint. Thus, it would be impossible to meet throughput and delay constraints outside the range of feasible values supported by the endpoint. However, the `tpipe` call may adjust the endpoint VCPU scheduling parameters to accommodate a pipe request, if the endpoint attributes (e.g., maximum endpoint throughput) are not violated. The successful

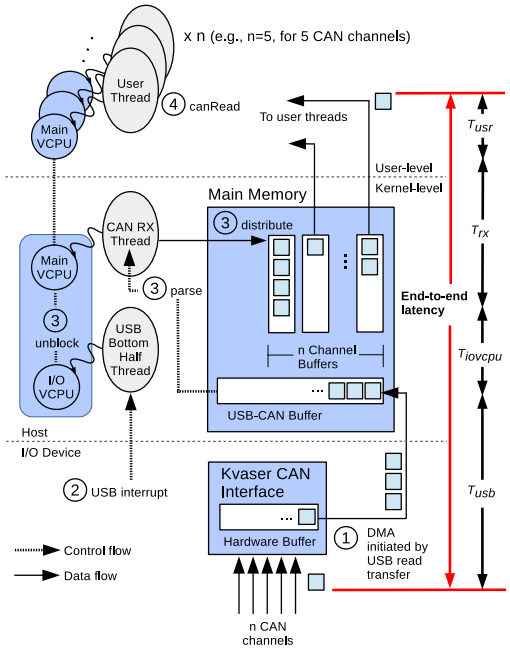

Fig. 8. Data and control flow for USB-CAN tuned pipes.

creation of a tuned pipe also requires the Main and I/O VCPUs to be feasibly scheduled according to the utilization bound test (shown in Equation (1)).

5.2 End-to-End Guarantee

The combination of Quest's VCPU and USB schedulers guarantees end-to-end latency requirements for device I/O and network transfers. In this section, we use a USB-CAN system as an example to elaborate on separate latencies that influence end-to-end time.

5.2.1 Latency Contributors. The end-to-end transfer delay is influenced by several factors, which we will identify as part of our analysis. To begin, we first consider the end-to-end data and control flow during a USB-CAN transfer, illustrated in Figure 8. The most complex path considers the input of data, as this has to be demultiplexed for different user-level threads. Consequently, the end-to-end time of a CAN message starts with its arrival at the interface to the CAN bus, and ends when the message is read by a user-space thread.

When a CAN message arrives, it is temporarily stored in a hardware buffer within the CAN controller before the host issues a USB read transfer request. A USB read transfer request will cause the data to be moved from the hardware buffer into a target host memory buffer[3] using DMA when the corresponding USB endpoint is scheduled by the host controller. Instead of triggering an interrupt upon the completion of each DMA transfer, the USB host controller can be configured to periodically trigger interrupts to poll the completion of multiple pending transfers at a time. The time that system software waits for the completion of the DMA contributes to the end-to-end latency, represented by (1) in Figure 8.

[3]In the case of USB-CAN, the target is a buffer allocated within the USB-CAN driver.

The time during which interrupts are disabled is minimized by delegating the polling operation to a threaded bottom-half routine, associated with a dedicated USB I/O VCPU. The delay between the completion of a DMA transfer and the invocation of the bottom-half thread also contributes to the end-to-end latency, represented by (2).

The bottom half updates the xHC host-resident data structures and invokes a specific callback function for each completed transfer. Common to subsystems that utilize a USB communication stack, our CAN driver registers a callback function that does nothing but wake up the thread that is waiting for the completion of the USB transfer in question. This is a dedicated CAN RX thread responsible for issuing all the USB read transfers. Upon waking up, it parses received CAN messages and distributes them to buffers assigned to each CAN channel. After that, the RX thread issues a new USB read transfer and yields the CPU. The delay between the invocation of the registered callback function and the completion of the RX thread execution is the third contributor to the end-to-end latency, represented by (3). As shown in Figure 8, the CAN RX thread is associated with a Main VCPU, whose budget is denoted by C_{rx} and period by T_{rx}. We select the initial budget C_{rx} through off-line profiling in order to provide the thread with enough time to issue one USB read request to the host controller, parse the received data, and distribute the data among separate CAN channel buffers for the user threads. The final value of C_{rx} and T_{rx} should be chosen based on the CAN bus load, and adjusted according to user requirements elaborated in Section 5.1.

A CAN message is queued in the channel buffer until a user-level thread, which has opened that channel, issues a system call to copy the message to user level. Each user-level application thread is assumed to be associated with a Main VCPU with C_{usr} budget and T_{usr} period. The waiting time a message spends in the channel buffer is the fourth, and last, contributor to the end-to-end latency, represented by (4). Section 5.1 also describes the way in which T_{usr} is determined.

5.2.2 End-to-End Timing Analysis. The scenario that leads to the worst case latency is the summation of largest delays incurred by steps (1)–(4). In order to derive the end-to-end transfer latency, we begin by describing the timing properties of all the steps involved in the analysis.

First, each USB transfer request causes data to be moved from the hardware buffer to the USB-CAN kernel buffer, which takes T_{usb} nanoseconds. We assume the USB device endpoint the driver reads from is accepted by our USB scheduler, and therefore, completes its transfer in T_{ep}[4] microframes. If the host controller is configured to trigger interrupts immediately, the worst-case delay of step (1) is $T_{usb} = T_{ep} \times \mu frame$ nanoseconds. However, in the case of xHCI triggering interrupts at every T_{xHCI} nanoseconds, $T_{usb} = T_{xHCI} \times \lceil \frac{T_{ep} \times \mu frame}{T_{xHCI}} \rceil$.

Second, the I/O VCPU associated with the bottom half of the xHCI driver is woken up every T_{iovcpu} nanoseconds, which contributes to the worst-case delay in step (2). The USB I/O VCPU is bounded by a predefined factor of U_{usb}% CPU utilization. Its budget is $C_{usb} = U_{usb} \cdot T_{rx}$ and its period is $T_{iovcpu} = T_{rx}$ because I/O VCPUs derive service constraints from the Main VCPU they serve when using PIBS.

Third, the worst-case overhead of the CAN RX thread to parse the USB-CAN buffer, distribute messages into separate channel buffers, and issue the next USB read request is T_{rx}. This is because the RX thread with a budget of C_{rx} may be preempted and, hence, will not necessarily complete execution before the end of its period. This contributes to the cost of step (3).

Finally, a user-level thread takes a maximum of T_{usr} time units to complete C_{usr} time units of execution when there is preemption, which contributes to the cost of step (4). C_{usr} is set to the worst-case value necessary to complete the copying of data into a user-level address space.

[4] T_{ep} is the period passed to the USB Scheduler by the USB-CAN driver for the input endpoint.

The four-step worst-case end-to-end latency is therefore:

$$\begin{aligned} E_{wc} &= T_{usb} + T_{iovcpu} + T_{rx} + T_{usr} \\ &= T_{usb} + 2 \cdot T_{rx} + T_{usr} \end{aligned} \tag{9}$$

Note that the worst-case delay is the sum of a sequence of periods associated with the four stages of data transfer. This is because Quest's scheduler guarantees that every task associated with a different VCPU will receive up to its reserved budget, C_i, every period T_i if Equation (1) is satisfied. The proof of this is omitted but is included in our work that extends tuned pipes across separate guest OSes supported by the Quest-V partitioning hypervisor [27].

6 EXPERIMENTAL EVALUATION

6.1 Real-Time USB 3 Scheduling

Our real-time USB 3.x scheduler is evaluated using the test setup shown in Figure 9. This represents a simplified version of a VMS under development for an industrial partner.

A Cincoze DX1100 embedded PC, featuring a 2.4 GHz 9th generation Intel Core-i7 hexacore processor consolidates multiple vehicle functions, including those for the **Instrument Cluster** (**IC**), **In-vehicle Infotainment** (**IVI**), and ADAS. The DX1100 runs our Quest RTOS, which implements the USB 3 scheduler described earlier. Two UP Squared single-board computers support additional vehicle control functions and backup services in case of hardware and software failures. In our production VMS, we run Quest alongside Linux to manage IC, IVI, and ADAS services. However, for these experiments, we simply run Quest on four cores, leaving the others which would ordinarily be available to Linux unused.

For this article, four USB cameras are used to represent front, rear, and side-facing visual sensors for obstacle detection and avoidance. Although the full VMS will feature more cameras and other sensors, such as LIDAR and RADAR, the basic setup used in this article is sufficient to show the benefits of our USB scheduling framework.

Four processes are forked in the Quest RTOS, one for each camera connected to the DX1100. These processes each run a single-threaded task bound to their own Main VCPU and separate physical core. All other tasks are responsible for processing and exchanging data with the Two UP Squared computers. The assignment of tasks to cores ensures they are guaranteed their corresponding CPU reserves. However, in this article, we make no additional attempts to avoid inter-core interference through implicitly shared resources such as caches and memory buses. Such interference is possible to mitigate using techniques such as cache partitioning [2, 9, 19, 33, 36, 50, 66, 67, 76, 78] and page coloring [10, 48, 49, 72, 79]. Our prior work on COLORIS shows an efficient method of dynamic page coloring [87]. This is, however, outside the scope of this article.

Each camera features a USB 3 IMX291 imaging chipset capable of capturing 50 HD frames per second (fps) at 1920 × 1080 pixel resolution. Each IMX291 has a hardware buffer of 4 MB, and supports an isochronous endpoint of 33 KB to transfer frames between the device and host memory. The endpoints have parameters $m = 2$, $n = 10$, and $b = 1024$ bytes, as described in Section 3.

Each UP Squared board runs an instance of Quest on a quad-core Pentium N4200 Intel processor operating at 1.1 GHz. These boards are used to represent the handling of timing critical tasks for chassis, body, and powertrain control, including torque vectoring and battery management. The DX1100 acts as the mission control center for the vehicle, gathering environmental information through one USB 3 isochoronous endpoint per camera, and communicating control signals to the UP Squared boards via two uni-directional USB 3 xDbC bulk endpoints per board. Depending on the operating mode of the vehicle, the DX1100 either processes driver-input commands or uses the signals generated by autonomous driving software.

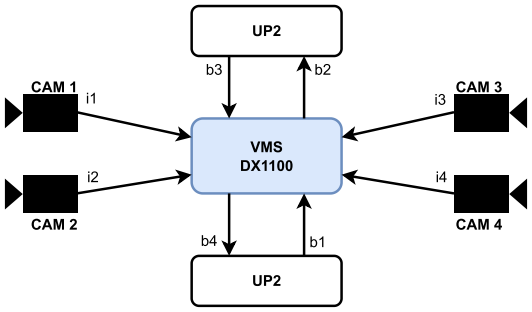

Fig. 9. USB 3 scheduling—experimental setup.

The service requirements of isochronous endpoints ($i_1, \ldots, i_4$) change at runtime. This is because the primary cameras of interest depend on whether the vehicle is moving forward, reversing, or changing lanes. However, bulk endpoints ($b_1, \ldots, b_4$) used for host-to-host communication and USB-CAN transfers are always critical for the safe operation of the vehicle. As stated in Section 3, the host controller hardware guarantees a minimum of 10% bandwidth for asynchronous transfers, but this may be insufficient without additional software management.

In this section, we present two sets of experiments using the setup described above to demonstrate how our bus scheduler guarantees throughput and delay requirements. We also discuss how the observed values follow our mathematical model in Section 3.

The maximum packet size of all endpoints in these experiments is 1024 bytes. Using Equation (3), we calculate the maximum packet transfer latency of each bulk and isochronous endpoint to be 1778.4 ns and 1720.8 ns, respectively.[5] The bus utilization values reported throughout this section are derived from the latency, and each endpoint's budget and period. However, the reserved throughput values exclude the various overheads of the USB bus, and they are simply calculated as the number of payload bits per microsecond. This matches what we measure in software on the DX1100 for each endpoint.

6.1.1 End-to-End Latency Guarantees. The objective of this experiment is to verify the worst-case latency analysis presented in Section 3. Five different scenarios are constructed, with the same total periodic and asynchronous bus utilizations but varying asynchronous transfer latencies. All scenarios pass the schedulability test, but exhibit different worst-case transfer latencies (c_j in Equation (6)) for bulk endpoints.

Camera one is assigned a budget of $33KB$ for every period of one micro-frame, while camera two is assigned the same budget for every other micro-frame. Cameras three and four are not used in this experiment. Therefore, the total bus utilization allocated to isochronous data transfers is 68.1%, which leaves 31.9% for the four bulk endpoints—i.e., $t_A = 39875$ ns in the worst case. The total bus utilization of asynchronous traffic is set to 5.7% to ensure all scenarios are schedulable.

In this set of experiments, each UP Squared generates a 60 minute series of timestamped 1 KB USB `PING` packets that are routed through the DX1100 to the opposite UP Squared board. The DX1100 constantly listens for new packets from the UP Squared boards, unpacks the headers, and forwards the packets to their destination based on a static routing table. Once an UP Squared receives a `PING` message, it generates a corresponding `PONG` message and transmits it to the original

[5]The latency parameters of a USB 3 host controller are as follows: $h = 5\ ns$, $\alpha = 0.2\ ns$, $\gamma = 1$, $p_{Bulk} = 134.4\ ns$, $p_{Isoc} = 76.8\ ns$.

Table 2. USB 3 Scheduling—Expected End-to-End Latencies for Bulk Endpoints

Scenario	Budget (B_j in bytes)	Period (T_j in micro-frames)	Endpoint Latency (c_j in micro-frames)	Worst-case End-to-End Latency (μs)
1	1,024	1	1	4,000
2	8,192	8	2	4,500
3	20,480	20	4	5,500
4	40,960	40	8	7,500
5	88,064	86	16	11,500

Table 3. USB 3 Scheduling—Delay Contributors

Entity	Period (μs)	Description
UP Squared		
T_{tx}/T_{rx}	250	xDbC sender/receiver MainVCPU
T_{xDbC}	125	Worst case latency of xDbC packets over the USB 3 Bus
DX1100		
T'_{tx}/T'_{rx}	250	Host-based router sender/receiver MainVCPU
T_{usb}	$125 \times c_j$	Worst-case latency of Bulk USB packets
T_{iovcpu}	250	xHCI bottom-half Handler

sender via the DX1100. Upon reception of a PONG message, each UP Squared verifies the integrity of the message using a checksum, and records the timestamp once again.

We calculate the round-trip message delay by subtracting the timestamp of each PONG message from the timestamp of its corresponding PING message. There are two possible round-trip paths: (1) from endpoint b_1 to b_4, and (2) from endpoint b_3 to b_2, each passing through the DX1100 and the opposite UP Squared. This type of round-trip path is representative of a range of throughput and delay contributors, and avoids performing clock-SYNC for an accurate time measurement as every delay item is observed using the same (local) timestamp counter. Table 2 shows the configuration of each bulk endpoint in every scenario, as well as the expected worst-case end-to-end latency values, calculated from Equation (10).

Table 3 shows the parameterization of the contributors responsible for the end-to-end latency of the packets originating from one UP Squared to the other (via DX1100) and back to the first UP Squared.

The round-trip latency of bulk packets follows from the analysis provided for the CAN interface driver in Section 5.2, with the following differences: (1) Our xDbC and USB-Bridge drivers support zero-copy data-transfer from user-space and do not perform any scatter-gather operations between threads, yielding $T_{usr} = 0$; (2) the xDbC Driver does not implement a bottom-half handler and polls for the completion of any pending operation; (3) to simplify T_{usb}, we configure the host controller to trigger interrupts immediately as they arrive.

The worst-case latency of one UP Squared sending a message to the DX1100 is $T_{tx} + T_{xDbC} = 375$ micro-seconds. Similarly, the worst-case delay of an UP Squared receiving a message is $T_{rx} + T_{xDbC} = 375$ micro-seconds. We expect an additional bottom-half handling delay of 250 micro-seconds for the DX1100 to exchange one message in each direction. Since the worst-case latency (c_j in Equation (6)) for each of the bulk endpoints in this experiment depends on the scenario, we express T_{usb} as a factor of c_j values reported in Table 2.

Table 4. USB 3 Scheduling—Observed Bulk Endpoint Latency Statistics

Latency (μs)	Scenario 1	Scenario 2	Scenario 3	Scenario 4	Scenario 5
Maximum	4,037	4,359	5,256	7,553	11,560
Minimum	3,033	3,426	2,859	3,268	3,521
Average	3,549	3,875	3,969	5,186	7,447
Std. Dev.	167	129	382	658	1,238

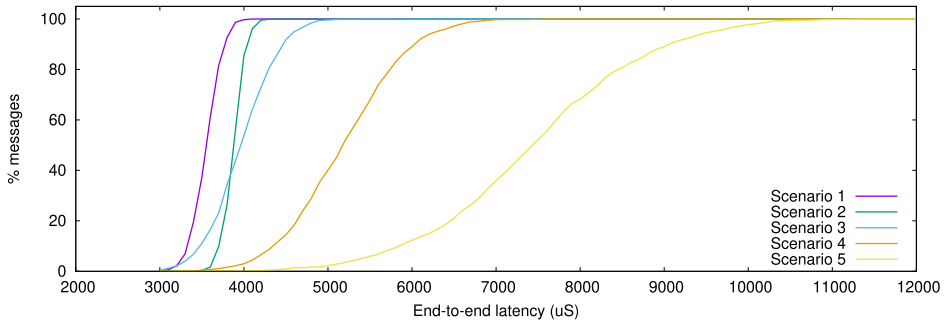

Fig. 10. USB 3 scheduling—CDF of end-to-end latencies of bulk endpoints.

Note that in schedulable configurations, the actual bulk endpoint latency for bus transactions ($125c_j$ micro-seconds for endpoint j) is always less than or equal to the period ($125T_j$ micro-seconds) of the endpoint depending on the amount of slack in t_A if the USB bus is under-utilized. For this reason, we calculate our expected worst-case latency values using c_j instead of T_j, to validate our latency analysis more precisely. The total end-to-end delay of a packet in this round trip is as follows:

$$\begin{aligned} E_{wc} &= 2(T_{tx} + T_{xDbC} + T_{usb} + T_{iovcpu} + T'_{rx} + T'_{tx} + T_{usb} + T_{iovcpu} + T_{xDbC} + T_{rx}) \\ &= 2(375 + 1000 + 2(125c_j) + 375) \\ &= 3500 + 500c_j \end{aligned} \tag{10}$$

Table 4 presents the message latencies for all five scenarios. The results of all four bulk endpoints are aggregated in each case since each such endpoint was configured with the same parameters and showed a similar latency. Figure 10 depicts a **cumulative distribution function** (**CDF**) of the results.

The maximum observed latencies shown in Table 4 in some cases exceeded the calculated worst-case latency values by at most 60 micro-seconds. The percentage of messages exceeding their expected worst-case latencies are 0.3%, 0.0%, 0.0%, 0.1%, and 0.1% for scenarios 1 to 5, respectively. We attribute these outliers to micro-architectural issues, such as cache misses and memory stalls, as well as measurement errors.

6.1.2 Throughput Guarantees. We again use the same setup as before to measure throughput guarantees. Three scenarios are considered, with various asynchronous and periodic traffic loads on the USB 3 bus. We show that bulk and high criticality isochronous endpoints receive their required throughputs when there is sufficient bus bandwidth for both classes of endpoints to be serviced. In contrast, low criticality isochronous endpoints are not guaranteed their service requests.

As stated earlier, the USB 3 hardware scheduler ensures service requirements of periodic endpoints are met by reserving bandwidth for $(m + 1)(n + 1)b$ bytes every period (in micro-frames), regardless of how much bandwidth is actually used by the USB device. In this set of experiments,

Table 5. USB 3 Scheduling—QoS Assignments and Throughput Expectations

Endpoint	Criticality	Budget (bytes)	Period (micro-frames)	% Bus Utilization	Throughput (Mbps)
Common to all scenarios					
i_1	High	33,792	1	45.4	2,162
i_2	Low	33,792	2	22.7	1,081
i_3	Low	33,792	4	11.4	541
i_4	Low	33,792	8	5.7	270
Scenario 1					
$b_1, \ldots, b_4$	High	4,096	8	0.71	33
			Bulk Endpoint Total	2.84	132
Scenario 2					
$b_1, \ldots, b_4$	High	6,144	2	4.27	197
			Bulk Endpoint Total	17.08	788
Scenario 3					
$b_1, \ldots, b_4$	High	8,192	1	11.38	520
			Bulk Endpoint Total	45.52	2,080

Table 6. USB 3 Scheduling—Observed Throughput Values of Individual Bulk Endpoints, $b_1, \ldots, b4$ (Mbps)

	Min	Max	Average	Std. Dev.
Senario 1	34.32	44.03	39.49	1.31
Senario 2	202.83	262.57	234.20	9.18
Senario 3	525.06	627.19	573.20	12.65

the cameras are configured with $m = 2$, $n = 10$, $b = 1024$ resulting in a budget of 33KB every period. The only exception to this is when an endpoint is rejected at the time of enumeration if the sum of periodic bandwidth reservations exceeds the 90% threshold. Without the addition of our software-based scheduler, the rejections happen in the order of USB device enumeration.[6] However, our scheduler will not submit low criticality periodic requests to the USB 3 hardware scheduler if they violate the requirements of bulk and high criticality isochoronous requests.

Table 5 shows the configured/expected QoS for each endpoint in the three scenarios. As mentioned earlier, the bus utilization values are derived from Equation (3) to account for the overheads induced by the USB protocol. For example, the bus utilization of the first isochronous endpoint is calculated as $\frac{(m_1+1)(n_1+1)w_1}{\mu frame} \times 100 = \frac{3\times11\times1720.8}{\mu frame} \times 100 = 45.4\%$.

Each scenario runs for 60 minutes and the throughput of each endpoint on the DX1100 is measured. The results of the four bulk endpoints are aggregated in each scenario, as they all have the same scheduling parameters. Table 6 summarizes the observed throughput values in Mbps.

Scenario 1 represents a relatively low-throughput asynchronous demand, where each bulk endpoint is guaranteed to transfer at least four 1024-byte packets every 8 micro-frames, requiring 0.71% of the USB 3 bus bandwidth. In this case, the total asynchronous bandwidth reservation is 2.84%, which is easily achievable as the host controller limits the periodic traffic to 90%. Moreover, the total bandwidth required by the four isochronous endpoints is 85.2%. Therefore, Scenario 1 must be schedulable even with a trivial periodic packet scheduler that starts every endpoint on the first micro-frame and assigns to each endpoint the subsequent micro-frames solely based on periodicity.

[6] At the time of device enumeration, the host controller reads all possible device configurations, and selects one with at least one interface. Periodic endpoints associated with that interface are rejected if they violate the bandwidth limit enforced by the USB Specification.

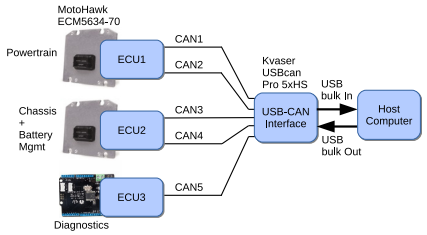

Fig. 11. Tuned pipes—experimental setup.

Scenario 2 presents a more challenging situation in which each of the four bulk endpoints transfer at least 6 packets every 2 micro-frames, resulting in a total asynchronous bandwidth reservation of 17.08%. This scenario is also schedulable by our bin-packing algorithm, which distributes the starting micro-frames of isochronous endpoints to leave at least 20.05% utilization left-over for asynchronous traffic.

In Scenario 3, the total bus utilization of asynchronous traffic equals 45.52%. Therefore, only the first and the fourth isochronous endpoints (i_1 and i_4) are schedulable. The low-criticality endpoints i_2 and i_3 are rejected by our software-based scheduler to ensure the highly-critical bulk endpoints are each guaranteed to transfer 8 KB every micro-frame. As a result, the actual periodic bandwidth reservation equals 51.1%. As seen, the minimum bandwidth for each bulk endpoint is 525.06 Mbps, which is above the reservation of 520 Mbps.

Running the above experiments using Yocto Linux 4.19 on the DX1100, we observed that only one camera is active at a time. This is due to the following differences between Quest and Linux: (1) the USB software stack in Linux only relies on the USB 3 hardware scheduler, and does not allow isochronous endpoint periods to be reconfigured to values other than what the devices report (here, 1 micro-frame), even if meaningful data is transferrable to the host; (2) the UVC driver in Linux always selects the 33 KB per micro-frame bandwidth reservation for the 50 fps FullHD streaming mode, as the next largest configuration presented by the cameras supports up to 37.5 frames per second. Due to reasons (1) and (2), each camera demands 45.4% of the overall bandwidth, which amounts to 90.8% for two cameras, which is above the 90% limit for periodic traffic. Therefore, only one of the cameras is accepted by the hardware bus scheduler.

Comparing Linux with Scenario 3 using the Quest software-based scheduler, bulk transfers are able to obtain $(100 - 45.4) = 54.6\%$ bandwidth, which is more than necessary to meet their requirements. However, unlike with Quest, none of the low criticality cameras are able to transfer data.

6.2 Tuned Pipes

The next set of experiments focuses on the performance of tuned pipes, using the testbed shown in Figure 11. A Kvaser USBcan Pro 5xHS five-channel CAN bus interface is connected via USB 3.0 to an UP Squared single-board computer. The UP Squared has 4 GB RAM and a dual-core Celeron N3350 processor operating at 1.1 Ghz. Tasks and interrupts are assigned to a shared single core, rather than having all I/O requests handled on a dedicated and potentially low-utilization core.

An automotive system is simulated by a sequence of CAN frames from several CAN devices acting as ECUs. ECU1 and ECU2 are represented by Woodward MotoHawk ECM5634-70 modules, commonly used for engine and powertrain control functions. An Arduino UNO with a SeeedStudio CAN-BUS Shield V1.2 is programmed to produce a repeatable sequence of CAN frames. This represents ECU3, which is connected to the fifth CAN channel (CAN5).

With the Kvaser USBcan Pro 5xHS interface, each CAN channel is associated with a separate bus having a maximum configurable bandwidth of 1 Mbps. It is worth noting that by empirical

Table 7. CAN Bus Traffic

Bus	CAN1	CAN2	CAN3	CAN4	CAN5
Bandwidth (bps)	500K	250K	500K	500K	500K
Throughput %	10	20	30	40	69
CAN frames	std	ext	std	ext	std

study, the USB-CAN interface appears to have a buffer size of 4 KB, which results in stale data being overwritten when it is full.

6.2.1 Endpoint Guarantees. In the first tuned pipes experiment, the ECUs generate CAN traffic at different rates to the host computer, to see if a USB-CAN endpoint is capable of receiving all frames without loss. Host tasks communicate with the Kvaser USB-CAN interface using the CANlib API [42]. The key functions of the CANlib API run on our Quest RTOS, to control bus timing properties and to read and write different CAN channels.

Quest's tuned pipes implementation is compared against Ubilinux, which includes the PREEMPT-RT patch. The bandwidth of each bus is limited to the maximum bit rates in **bits-per-second** (**bps**) shown in Table 7. This configuration has a maximum bandwidth aggregated across all channels of $4 \times 500 + 250 = 2250$ Kbps, resulting in a minimum of 14 ms to fill the 32 Kilobit hardware buffer of the USB-CAN interface. The actual steady-state throughputs, as a percentage of each channel's maximum configured bandwidth, are shown below the bandwidth values. The final row in the table shows the frame format on each channel, from standard (`std`) 11-bit, to extended (`ext`) 29-bit frame IDs. The data payload of each frame type is set to the maximum 8 bytes.

The Linux implementation of the Kvaser CAN driver uses a kernel (RX) thread to periodically issue USB read transfers. The RX thread is blocked until the USB xHCI bottom half processes the transfer completion event. The bottom half runs as a non-schedulable softirq. The SCHED_DEADLINE [34] policy is used with the RX thread, having a runtime budget of 2 ms and a period of 14 ms. A comparable Quest experiment uses an RX thread assigned to a Main VCPU with the same budget and period. However, Quest's xHCI bottom half interrupt handler is assigned to a *schedulable* I/O VCPU with a CPU utilization of 1%.

The budget is derived by measuring the time for the RX thread to parse and distribute 4 KB data into separate host memory buffers for different endpoints, as well as account for buffer overruns. The period and deadline of a thread are set to the same value, for all cases where SCHED_DEADLINE is used in Linux.

Table 8 shows the 6 scenarios used to compare Linux and Quest. These scenarios vary in experiment duration from 30 to 60 seconds, and whether there are I/O-, CPU-, or both I/O- and CPU-bound tasks. The aim is to show the extent to which Linux and Quest are able to achieve temporal isolation between tasks. For I/O, 5 separate CAN reader tasks are used, one per channel. I/O-bound tasks run at their default priorities in both Linux and Quest. For cases where CPU-bound tasks are involved, each task increments a counter every 10 μs and reports how far away the counter is from the expected value, for a given budget and period over the duration of the experiment. CPU-bound tasks run under SCHED_DEADLINE in Linux, each having a budget of 1 ms and a period of 7 ms. Quest uses the same budget and period values for corresponding Main VCPUs. These periods yield relatively higher priorities for the CPU-bound threads than the RX thread, potentially causing the greatest interference on I/O operations.

Table 9 shows that Quest does not experience any lost CAN packets, as there are no overruns of the USB-CAN 4 KB buffer for any of the experimental scenarios. Linux, however, experiences overruns in the two scenarios where there are higher priority CPU-bound tasks. These cause interference with the USB xHCI bottom half (softirq) in Linux. The softirq initially runs with the

Table 8. Experimental Scenarios

Scenario	Duration (s)	CPU-bound Tasks	I/O-bound Tasks
1	30	0	5
2	30	3	0
3	30	3	5
4	60	0	5
5	60	3	0
6	60	3	5

Table 9. USB-CAN Buffer Overruns

	Scenario	Buffer Overruns
Quest	All Scenarios	0
Linux	3	230
	6	405

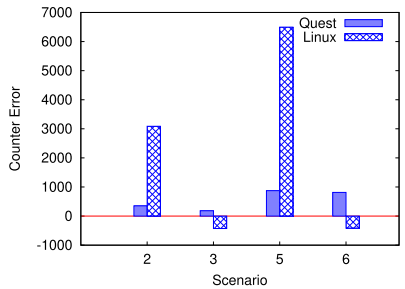

Fig. 12. Average counter error for CPU-bound tasks.

highest priority in the system, with interrupts enabled. However, if the softirq processing loops `MAX_SOFTIRQ_RESTART` times, as described in Section 2, and still finds more softirqs to process due to high rate of interrupts, it will wake up a *ksoftirqd* thread to handle the remaining work. Older versions of Linux set ksoftirqd to run at the lowest priority on the target CPU, but recent versions set the thread to normal user-level task priority. Notwithstanding, the consequences are that interrupt bottom halves in Linux are either handled at the highest priority, or a relatively low priority. SCHED_DEADLINE, therefore, has limited benefit for tasks with I/O requirements.

Figure 12 shows the average counter error for the three CPU-bound tasks running on Quest and Linux, in each of the corresponding scenarios. For Scenarios 3 and 6, the Linux tasks increment their counters beyond the expected value for their initial budgets. This appears to be an artifact of SCHED_DEADLINE, which redistributes unused budget of blocked deadline tasks amongst those that are runnable. The RX thread in Linux will block until the xHCI bottom half handles completion interrupts for USB transfers. As observed above, the bottom half is deferred when the frequency of interrupts surpasses a certain threshold, to allow interrupted tasks to proceed. For Scenarios 2 and 5, there are no I/O tasks active.

The CPU-bound tasks in Linux show significant errors between the actual and expected counter values. The positive values indicate that Linux tasks are *lagging* behind their expected progress. This is partly due to SCHED_DEADLINE tasks being unable to reclaim the unused budget of blocked tasks, and also the overhead of the task scheduler. In each case, Quest guarantees progress close to what is expected. Although not shown, Quest has the ability to allow tasks to use CPU

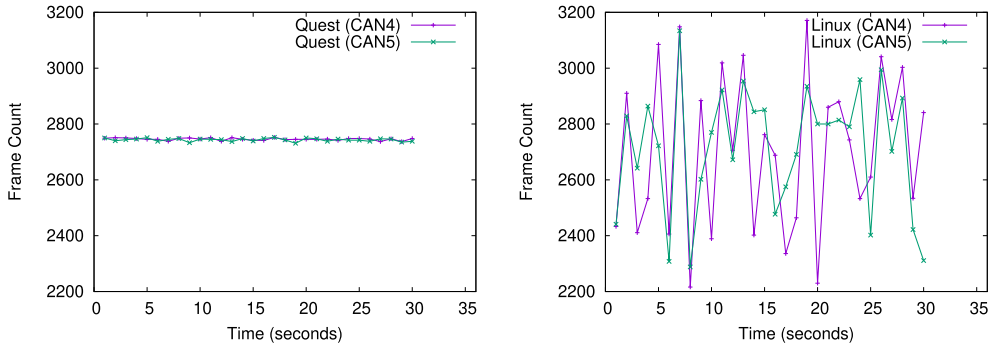

Fig. 13. Number of frames per second received by two user-level I/O tasks, with (a) Quest, and (b) Linux.

cycles above their budget limits when no other tasks have available budgets. This allows the CPU-bound tasks to increment their counters far in excess (more than 495,000) of their target values.

6.2.2 End-to-End Guarantees.

Input Requests. A further set of tuned pipes experiments concerns the delivery of data to user-space tasks according to throughput and delay constraints. The experimental setup is similar to that in Figure 11, except CAN4 is replaced with a second Arduino and CAN shield. Both CAN4 and CAN5 generate standard frames with a throughput of 69% of their 500 Kbps channel bandwidths. Five tasks on the UP Squared open pipes to the USB-CAN devices, and record the number of CAN frames received every second over a 30 second period. Standard CAN frames are 108 bits, while extended frames are 128 bits. However, the Kvaser USB-CAN interface requires each frame to be encapsulated in a 64-byte message. All subsequent throughput calculations are based on the Kvaser message size.

An oscilloscope measures the minimum and maximum transmission delay of a CAN frame from each Arduino, which is 363.4 μs and 366.2 μs, respectively. These values imply that the minimum and maximum throughput from the two Arduinos should be in the range [1/366.2 μs, 1/363.4 μs]=[2730 frames/s, 2752 frames/s].

For Quest, each I/O task creates a tuned pipe with the following QoS specification as defined in Section 5.1: `latency`=1×10^9 ns, `tput`=2752 CAN frames per second, `IObufsz`=128 CAN frames, and `texec_time`=2 ms. The throughput (`tput`) is calculated from the maximum effective transfer rate from a single Arduino when sending 108-bit (largest-size) standard CAN frames. The effective transfer rate accounts for additional bits on the CAN bus for bit stuffing and protocol overheads. Using Little's Law, the I/O task period is set to 46 ms, which is the largest interval before a user-space buffer of 128 frames could overflow. The budget for each I/O task is set to 2 ms, which is sufficient to process up to 128 buffered frames. Quest I/O tasks are assigned to Main VCPUs, while equivalent Linux tasks are assigned to the SCHED_DEADLINE class.

Figure 13 shows the number of frames per second received by the two I/O tasks associated with the Arduino devices, for both Quest and Linux. The first second of data is omitted to allow I/O buffers to populate. Similarly, Figure 14 shows the minimum, maximum and average throughput across both USB-CAN channels using Quest and Linux. The two horizontal lines show the target minimum and maximum values that should be maintained for successful throughput guarantees.

As can be seen, the Quest tasks receive data within the expected throughput bounds of 2,730 to 2,752 frames/s, whereas Linux tasks do not. The reason Linux tasks sometimes exceed their throughput bounds in 1s intervals is that they fail to receive sufficient data in the previous second, and subsequently copy additional frames from the endpoint buffer in the next second. Finally,

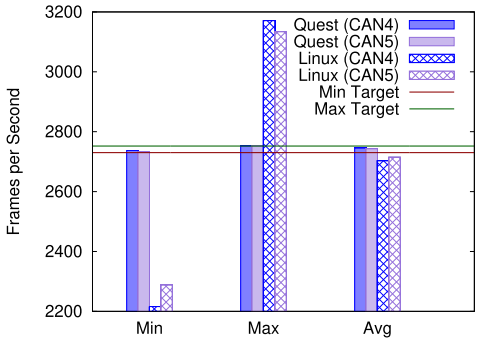

Fig. 14. Minimum, maximum, and average (input) throughput of the two CAN channels using Linux and Quest.

Linux fails to sustain an average throughput even above the minimum 2,730 frames/s generated by Arduino. This is because Linux loses some CAN frames as observed in the previous experiment (Table 9). With Linux, CAN4 and CAN5 average 2,703 and 2,714 frames per second, respectively. In Quest, CAN4 and CAN5 average 2,745 and 2,743 frames per second, respectively, without loss of packets.

Note that although the average throughput of Linux is close to the required target range, the fact that packets are lost could be critical to a real system. For example, a lost CAN frame that affects automotive braking or engine speed could lead to an accident, with potential loss of life.

Output Requests. For completeness, a series of experiments to show throughput and delay guarantees on data output to a USB-CAN device are performed. A setup similar to that in Table 7 is used, with three of the five channels (CAN1-CAN3) operating as inputs, as before. CAN4 and CAN5 are associated with two Arduino CAN devices, configured to receive data from the host. Given the limitations of the Arduino CAN shields, all data is sent from the host to these devices in standard frame format.

For Quest, a separate tuned pipe is established for CAN4 and CAN5, to output data to each Arduino device. As before, an oscilloscope is used to measure the latency of one iteration of an Arduino sketch, which toggles a GPIO pin upon reception of a CAN frame from the host.

The observed minimum and maximum latencies for one iteration of the Arduino sketch are 325.4 μs and 327.5 μs, respectively. These values are established while transferring data from the host as fast as possible to ensure the Kvaser USB-CAN interface buffer is always full. These measured latencies corresponded to a maximum and minimum throughput of 3,073 and 3,053 CAN frames per second, respectively.

The latency to send 128 64-byte messages from a user-level host buffer for each tuned pipe is measured to be no more than 2 milliseconds. Consequently, the QoS specification for each tuned pipe for CAN4 and CAN5 is set to: `latency`=$1{\times}10^9$ ns, `tput`=3073 CAN frames per second, `IObufsz`=128 CAN frames, and `texec_time`=2 ms. The main VCPU for each tuned pipe in Quest has a derived budget and period of $C = 2$ms and $T = 41$ms, respectively. For comparison with Linux, a separate SCHED_DEADLINE thread is established for each channel having equivalent constraints to those of Quest's Main VCPUs.

Figure 15(a) shows that Quest is able to transfer data from a host to each Arduino device via the USB-CAN interface according to the tuned pipe QoS specification. In comparison, Figure 15(b) shows that Linux sometimes experiences significant drops in throughput, below the minimum 3,053 CAN frames per second expected for each Arduino device. This is because of demotion in

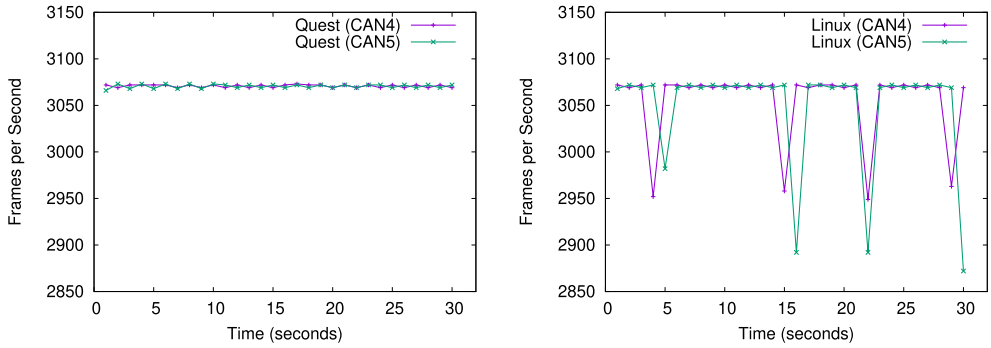

Fig. 15. Number of frames per second sent by two user-level I/O tasks, with (a) Quest, and (b) Linux.

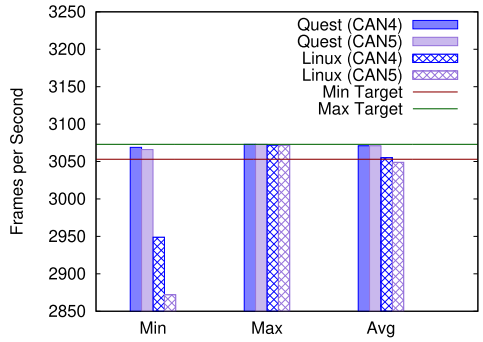

Fig. 16. Minimum, maximum, and average (output) throughput of the two CAN channels using Linux and Quest.

the priority of the xHCI bottom half used for USB transfers, as described earlier. As can be seen, bottom half priority inversion affects both input and output transfers.

Finally, Figure 16 shows that the average throughput for CAN4 and CAN5 is within the expected range with Quest. Although the average throughput with Linux is almost within the expected range, there is far greater variance in the minimum and maximum throughput than with Quest.

7 RELATED WORK

7.1 Resource Reservation

Central to the coordination of host processing and USB transfers is the combined scheduling of tasks and interrupt handlers. The USB host controller generates interrupts either on completion of a transfer request, or at a pre-determined rate. These events must be correctly scheduled with other tasks to achieve the end-to-end service guarantees provided by our tuned pipes framework. As noted in Section 2, the Quest RTOS provides the necessary abstractions for real-time end-to-end transfer and processing of USB data.

Unlike other real-time OSs [6, 23, 28, 43, 55, 56, 58, 69, 73, 84], Quest provides processor reservations to ensure task and I/O events are guaranteed timely execution. This is similar to resource reserves in Linux/RK [63], which are derived from processor capacity reserves in RT-Mach [57]. Other systems such as Redline [86] support the notion of budgets and replenishments, similar to how Quest works, using a task scheduling model similar to that used in **Deferrable Servers** [5, 77]. In contrast to both Redline and Linux/RK, Quest allows I/O events to be processed at priorities in-

herited from virtual servers responsible for executing tasks, for whom I/O event processing is being performed.

The HARTIK kernel [1] provides temporal isolation between hard and soft real-time tasks, using a **Constant Bandwidth Server (CBS)** approach. A CBS has a current budget, c_s and a bandwidth limited by the ratio Q_s/T_s, where Q_s is the maximum server budget available in the period T_s. When a server depletes all its budget it is recharged to its maximum value. A corresponding server deadline is updated by a function of T_s, depending on the state of the server when a new job arrives for service, or when the current budget expires.

CBS guarantees a total utilization factor no greater than Q_s/T_s, even in overloads, by specifying a maximum budget in a designated window of time. This contrasts with work on the **Constant Utilization Server** (**CUS**) [18] and **Total Bandwidth Server** (**TBS**) [75], which ensure bandwidth limits only when actual job execution times are no more than specified worst-case values. CBS has bandwidth preservation properties similar to that of the **Dynamic Sporadic Server** (**DSS**) [25] but with better responsiveness.

Linux now supports the SCHED_DEADLINE scheduling policy [46], based on the **Earliest Deadline First** (**EDF**) and CBS algorithms, with resource reservations. However, resource reservations are limited to tasks rather than I/O event handlers, unlike with Quest's virtual CPU scheduler.

7.2 Real-Time Interrupt Handling

Quest uses Main and I/O VCPUs to provide processor reservations for tasks and interrupt events, respectively. Several research works [16, 21, 37, 68] have investigated the idea of integrating interrupt handling with the scheduling and accountability of processes. Brandenburg et al. [8] highlighted the importance of accurate interrupt accounting for multiprocessor real-time systems. Lewandowski et al. [47] considered bandwidth constraints on device driver execution. Similarly, Manica et al. [54] presented a theoretical model for scheduling interrupt threads following the reservation-based approach. However, none of these works explore the dependency between interrupts and processes, and use that information to decide the priority of interrupt handlers (essentially bottom halves) in CPU scheduling. Motivated by Zhang and West [88], Quest combines the scheduling and accountability of interrupts associated with corresponding processes that issue service requests on I/O devices.

Previous work by Kuhns et al. [41] showcased the design and analysis for real-time I/O subsystems. The temporal isolation between interrupt handlers and tasks is used in Quest's tuned pipes for I/O transfers. Scout [61] exposes paths that are similar to pipes in our system, as a way to offer QoS guarantees to applications. However, paths in Scout are non-preemptive schedulable entities, ordered according to an EDF policy. Similarly, RAD-FLOWS by Pineiro et al. [65], provide a method to guarantee end-to-end inter-process communication, instead of throughput and delay-constrained I/O transfers.

7.3 Predictable Networking and I/O

Research shows how Linux's approach to USB 2.0 scheduling leads to unnecessary transfer request rejections, which are feasible according to available bus bandwidth [59]. The same work shows how a suitably written USB host controller driver is capable of providing throughput and delay guarantees on USB bulk data transfers that would otherwise be given best-effort service.

Other work provides a derivation of the worst-case transmission delay in a CAN bus for real-time communication, considering blocking delays and jitter [14, 81]. Similar work presents a scheduling analysis of CAN using FIFO rather than priority queues [13]. 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.

Real-time bus communication has been studied in both a theoretical framework [3, 35, 45] and in various physical implementations [13, 40, 52, 90]. Lehoczky et al. [35] 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.

Zuberi and Shin [90] address the utilization problem of CAN bus networks. They introduce the **mixed traffic scheduler** (**MTS**), which is a hierarchical scheduler using both EDF and **Deadline Monotonic Scheduling** (**DMS**). The authors state that pure EDF 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.

New alternative technologies are emerging in situations where CAN buses have insufficient bandwidth. **Time Sensitive Networking** (**TSN**) [11], **Audio Video Bridging** (**AVB**) [4], and Time Triggered Ethernet [38, 39] are comparable [89] technologies now found in the automotive and avionics domains. The scheduling properties of these network technologies have been modeled and analyzed by various research groups [7, 17, 64]. However, we believe that USB offers a noticeable advantage over these alternatives in that it offers a common bus solution for both networking and device I/O. Our future work will focus on a comparison between USB and the aforementioned technologies such as TSN and CAN for combined (host-to-host) network and device I/O transfers.

Huang et al. [29, 30] 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, endpoint service requirements are translated to have 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 an interval of 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 Specification as the endpoint interval rate is not respected. Similarly, polling at a higher rate is not guaranteed to work for all devices.

8 CONCLUSIONS

This article introduces a unified real-time USB software stack for networking and device I/O. Our USB 3.x scheduler is shown to support timing guarantees on both periodic and asynchronous requests. The software stack is built into the Quest RTOS, to leverage both Main and I/O VCPUs, which integrate interrupt and task scheduling. A *tuned pipes* abstraction that combines VCPU and USB scheduling shows how to provide end-to-end timing guarantees on pipelines between host tasks and devices. Experimental evaluations using a star topology of hosts and cameras corroborate our analysis of end-to-end service guarantees.

A USB-CAN implementation in Quest is compared to a standalone Linux system featuring real-time preemption and deadline-based scheduling support. I/O service constraints are shown to be guaranteed in all cases in Quest, but not Linux.

This work provides the impetus to consider USB as a viable high bandwidth control area network in place of traditional CAN buses. Before that is possible, further work is needed to investigate various networking topologies and device configurations for emerging industrial control applications, requiring the exchange of large volumes of data. Notwithstanding, in comparison to technologies such as TSN, USB has the ability to combine the scheduling of both device I/O and host-to-host communication data. In fact, USB 4.0 specifies support for host-to-host communication in combination with traditional device I/O. The work in this article should transfer to future USB standards where scheduling bulk transactions with service guarantees in the presence of periodic data is necessary.

ACKNOWLEDGMENTS

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 also to our colleagues at Drako Motors and Celenum, without whose support this work would not be possible.

REFERENCES

[1] Luca Abeni and Giorgio Buttazzo. 1998. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*. 4–13.

[2] David H. Albonesi. 1999. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO'99)*. 248–259.

[3] Caglan M. Aras, James F. Kurose, Douglas S. Reeves, and Henning Schulzrinne. 1994. Real-time communication in packet-switched networks. In *Proceedings of the IEEE*. 122–139.

[4] Lucia Lo Bello. 2014. Novel trends in automotive networks: A perspective on ethernet and the IEEE audio video bridging. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*. 1–8. DOI: https://doi.org/10.1109/ETFA.2014.7005251

[5] Guillem Bernat and Alan Burns. 1999. New results on fixed priority aperiodic servers. In *Proceedings of the IEEE Real-Time Systems Symposium*. 68–78. Retrieved from citeseer.ist.psu.edu/bernat99new.html.

[6] Gedare Bloom, Joel Sherrill, Tingting Hu, and Ivan Cibrario Bertolotti. 2021. *Real-Time Systems Development with RTEMS and Multicore Processors (Embedded Systems)*. CRC Press.

[7] Unmesh D. Bordoloi, Amir Aminifar, Petru Eles, and Zebo Peng. 2014. Schedulability analysis of ethernet AVB switches. In *Proceedings of the 2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*. 1–10. DOI: https://doi.org/10.1109/RTCSA.2014.6910530

[8] Björn B. Brandenburg, Hennadiy Leontyev, and James H. Anderson. 2009. Accounting for interrupts in multiprocessor real-time systems. In *Proceedings of the 2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. 273–283. DOI: https://doi.org/10.1109/RTCSA.2009.37

[9] Jichuan Chang and Gurindar S. Sohi. 2007. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the International Conference on Supercomputing (ICS'07)*. 242–252.

[10] Sangyeun Cho and Lei Jin. 2006. Managing distributed, shared L2 caches through OS-level page allocation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. 455–468.

[11] Cisco. 2017. Time-Sensitive Networking: A Technical Introduction. (2017). White paper, www.cisco.com.

[12] M. Danish, Y. Li, and R. West. 2011. Virtual-CPU scheduling in the Quest operating system. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. 169–179.

[13] Robert I. Davis. 2011. Controller area network (CAN) schedulability analysis with FIFO queues. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS)*. 45–56.

[14] Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. 2007. Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems* 35, 3 (2007), 239–272.

[15] D. de Niz, L. Abeni, S. Saewong, and R. Rajkumar. 2001. Resource sharing in reservation-based systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No.01PR1420)*. 171–180. DOI: https://doi.org/10.1109/REAL.2001.990608

[16] L. E. Leyva del Foyo, P. Mejia-Alvarez, and D. de Niz. 2006. Predictable interrupt management for real time kernels over conventional PC hardware. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*. 14–23.

[17] Libing Deng, Guoqi Xie, Hong Liu, Yunbo Han, Renfa Li, and Keqin Li. 2022. A survey of real-time ethernet modeling and design methodologies: From AVB to TSN. *ACM Computing Surveys* 55, 2, Article 31 (jan 2022), 36 pages. DOI: https://doi.org/10.1145/3487330

[18] Z. Deng, J. W. S. Liu, and J. Sun. 1997. A Scheme for Scheduling Hard Real-Time Applications in Open System Environment. (1997). https://ieeexplore.ieee.org/document/613785.
[19] Haakon Dybdahl, Per Stenström, and Lasse Natvig. 2006. A cache-partitioning aware replacement policy for chip multiprocessors. In *Proceedings of the High Performance Computing*, Vol. 4297/2006. 22–34.
[20] EHCI March 12, 2002. *Enhanced Host Controller Interface Specification for Universal Serial Bus* (1.0 ed.).
[21] T. Facchinetti, G. Buttazzo, M. Marinoni, and G. Guidi. 2005. Non-preemptive interrupt scheduling for safe reuse of legacy drivers in real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*. 98–105.
[22] FlexRay Consortium 2010. *FlexRay Communications System Protocol Specification* (3.0.1 ed.). FlexRay Consortium.
[23] FreeRTOS April 2023. Official website. Retrieved from https://www.freertos.org/.
[24] Michael R. Garey and David S. Johnson. 1981. Approximation algorithms for bin packing problems: A survey. In *Proceedings of the Analysis and Design of Algorithms in Combinatorial Optimization*. Springer, 147–172.
[25] T. M. Ghazalie and T. Baker. 1995. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems* 9 (1995), 31–67.
[26] Ahmad Golchin, Zhuoqun Cheng, and Richard West. 2018. Tuned pipes: End-to-end throughput and delay guarantees for USB devices. In *Proceedings of the 2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 196–207.
[27] Ahmad Golchin, Soham Sinha, and Richard West. 2020. Boomerang: Real-time I/O meets legacy systems. In *Proceedings of the 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 390–402.
[28] Dan Hildebrand. 1992. An architectural overview of QNX. In *Proceedings of the USENIX Workshop on Microkernels and Other Kernel Architectures*. 113–126.
[29] Chih Yuan Huang, Li Pin Chang, and Tei Wei Kuo. 2003. A cyclic-executive-based QoS guarantee over USB. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)*. IEEE Computer Society. Retrieved from http://dl.acm.org/citation.cfm?id=827266.828522.
[30] Chih Yuan Huang, Tei Wei Kuo, and Ai Chun Pang. 2004. QoS support for USB 2.0 periodic and sporadic device requests. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*. IEEE Computer Society, 395–404. DOI: https://doi.org/10.1109/REAL.2004.45
[31] Prolific Technology Inc. 2019. SuperSpeed USB 3.0 Host-to-Host Bridge Controller Datasheet. (2019). https://prolificusa.com/wp-content/uploads/2018/02/DS-27131201-PL27A1_V1.4.pdf.
[32] ISO 11898 2009. Road Vehicles—Controller Area Network (CAN). ISO 11898. https://www.iso.org/obp/ui/#iso:std:iso:11898:-1:ed-2:v1:en.
[33] Ravi Iyer. 2004. CQoS: A framework for enabling QoS in shared caches of CMP platforms. In *Proceedings of the18th Annual International Conference on Supercomputing*. 257–266.
[34] Kernel.org. 2014. Linux Deadline Scheduling Policy. (2014). Retrieved from https://www.kernel.org/doc/Documentation/scheduler/sched-deadline.txt.
[35] K. A. Kettler, J. P. Lehoczky, and J. K. Strosnider. 1995. Modeling bus scheduling policies for real-time systems. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS'95)*. IEEE Computer Society. Retrieved from http://dl.acm.org/citation.cfm?id=827267.828916.
[36] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. 2004. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the Parallel Architectures and Compilation Techniques (PACT'04)*.
[37] Steve Kleiman and Joe Eykholt. 1995. Interrupts as threads. *SIGOPS Oper. Syst. Rev.* 29, 2 (April 1995), 21–26.
[38] Hermann Kopetz. 2008. The rationale for time-triggered ethernet. In *Proceedings of the 2008 Real-Time Systems Symposium*. 3–11. DOI: https://doi.org/10.1109/RTSS.2008.33
[39] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer. 2005. The time-triggered ethernet (TTE) design. In *Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*. 22–33. DOI: https://doi.org/10.1109/ISORC.2005.56
[40] Hermann Kopetz and Günter Grünsteidl. 1994. TTP-a protocol for fault-tolerant real-time systems. *Computer* 27, 1 (Jan. 1994), 14–23. DOI: https://doi.org/10.1109/2.248873
[41] F. Kuhns, D. C. Schmidt, and D. L. Levine. 1999. The design and performance of a real-time I/O subsystem. In *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium*. 154–163. DOI: https://doi.org/10.1109/RTTAS.1999.777670
[42] Kvaser CANlib April 2023. CAN bus API. (April 2023). Retrieved from https://www.kvaser.co.
[43] J. J. Labrosse. 2002. *MicroC/OS-II: The Real Time Kernel*. Taylor & Francis.
[44] G. Lamastra, G. Lipari, and L. Abeni. 2001. A bandwidth inheritance algorithm for real-time task synchronization in open systems. In *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No.01PR1420)*. 151–160. DOI: https://doi.org/10.1109/REAL.2001.990606
[45] John P. Lehoczky and Lui Sha. 1986. 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)*. ACM, New York, NY, 44–53. DOI: https://doi.org/10.1145/317499.317538

[46] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli. 2016. Deadline scheduling in the Linux kernel. *Software: Practice and Experience* 46, 6 (June 2016), 821–839.
[47] M. Lewandowski, M. J. Stanovich, T. P. Baker, K. Gopalan, and A. I. A. Wang. 2007. Modeling device driver effects in real-time schedulability analysis: Study of a network driver. In *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*. 57–68.
[48] Jochen Liedtke, Hermann Härtig, and Michael Hohmuth. 1997. OS-controlled cache predictability for real-time systems. In *Proceedings of the 3rd IEEE Real-time Technology and Applications Symposium*.
[49] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. 2008. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proceedings of the 14th IEEE International Symposium on High Performance Computer Architecture*. 367–378.
[50] Chun Liu, Anand Sivasubramaniam, and Mahmut Kandemir. 2004. Organizing the last line of defense before hitting the memory wall for CMPs. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 176–185.
[51] C. L. Liu and James W. Layland. 1973. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM* 20, 1 (1973), 46–61.
[52] Jork Loeser and Hermann Härtig. 2004. Low-latency hard real-time communication over switched ethernet. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04)*. IEEE Computer Society, 13–22. DOI: https://doi.org/10.1109/ECRTS.2004.16
[53] M68HC11 2007. *M68HC11 Reference Manual* (6.1 ed.). https://www.nxp.com/docs/en/reference-manual/M68HC11RM.pdf.
[54] Nicola Manica, Luca Abeni, and Luigi Palopoli. 2010. Reservation-based interrupt scheduling. In *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. 46–55. DOI: https://doi.org/10.1109/RTAS.2010.25
[55] M. Masmano, Y. Valiente, P. Balbastre, I. Ripoll, and A. Crespo. 2010. LithOS: An ARINC-653 guest operating system for XtratuM. In *Proceedings of the 12th Real-Time Linux Workshop*.
[56] A. J. Massa. 2003. *Embedded Software Development with eCos*. Pearson Education, Inc., Upper Saddle River, NJ.
[57] Clifford Mercer, Stefan Savage, and Hideyuki Tokuda. 1993. Processor capacity reserves: An abstraction for managing processor usage. In *Proceedings of the 4th Workshop on Workstation Operating Systems*. 129–134.
[58] Miosix Accessed: April 2023. Official Website. (Accessed: April 2023). Retrieved from http://www.miosix.org.
[59] E. Missimer, Y. Li, and R. West. 2013. Real-time USB communication in the quest operating system. In *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 11–20.
[60] E. Missimer, K. Missimer, and R. West. 2016. Mixed-criticality scheduling with I/O. In *Proceedings of the 28th Euromicro Conference on Real-Time Systems (ECRTS)*. 120–130.
[61] David Mosberger and Larry L. Peterson. 1996. Making paths explicit in the Scout operating system. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*. ACM, New York, NY, 153–167.
[62] NXP Semiconductor October 1, 2021. *The I^2C Bus Specification and User Manual, Revision 7.0*.
[63] Shuichi Oikawa and Ragunathan Rajkumar. 1998. Linux/RK: A portable resource kernel in Linux. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*.
[64] Maryam Pahlevan, Nadra Tabassam, and Roman Obermaisser. 2019. Heuristic list scheduler for time triggered traffic in time sensitive networks. *SIGBED Review* 16, 1 (feb 2019), 15–20. DOI: https://doi.org/10.1145/3314206.3314208
[65] Roberto Pineiro, Kleoni Ioannidou, Scott A. Brandt, and Carlos Maltzahn. 2011. Rad-flows: Buffering for predictable communication. In *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 23–33.
[66] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. 2006. Architectural support for operating system-driven CMP cache management. In *Proceedings of the Parallel Architectures and Compilation Techniques (PACT'06)*. 2–12.
[67] Parthasarathy Ranganathan, Sarita V. Adve, and Norman P. Jouppi. 2000. Reconfigurable caches and their application to media processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. 214–224.
[68] John Regehr. 2001. HLS: A framework for composing soft real-time schedulers. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*. 3–14.
[69] RTLinux Accessed: April 2023. RT-Linux Community Website. (Accessed: April 2023). Retrieved from https://wiki.linuxfoundation.org/realtime/start.
[70] L. Sha, R. Rajkumar, and J. P. Lehoczky. 1990. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers* 39, 9 (September 1990), 1175–1185. DOI: https://doi.org/10.1109/12.57058
[71] Lui Sha, R. Rajkumar, and S. S. Sathaye. 1994. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. *Proceedings of the IEEE* 82, 1 (1994), 68–82. DOI: https://doi.org/10.1109/5.259427

[72] Timothy Sherwood, Brad Calder, and Joel Emer. 1999. Reducing cache misses using hardware and software page placement. In *Proceedings of the 13th International Conference on Supercomputing*. New York, NY, 155–164.
[73] Green Hills Software. 2015. INTEGRITY-178B RTOS. (2015). Retrieved from http://www.ghs.com/products/safety_critical/integrity-do-178b.html.
[74] B. Sprunt. 1989. *Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System*. Technical Report CMU/SEI-89-TR-011. Software Engineering Institute, Carnegie Mellon.
[75] M. Spuri and G. Buttazzo. 1994. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the IEEE Real-Time Systems Symposium*.
[76] Shekhar Srikantaiah, Mahmut Kandemir, and Mary Jane Irwin. 2008. Adaptive set pinning: Managing shared caches in CMPs. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*.
[77] Jay K. Strosnider, John P. Lehoczky, and Lui Sha. 1995. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environment. *IEEE Transactions on Computers* 44, 1 (January 1995), 73–91.
[78] G. E. Suh, L. Rudolph, and S. Devadas. 2004. Dynamic partitioning of shared cache memory. *Journal of Supercomputing* 28, 1 (April 2004), 7–26.
[79] George Taylor, Peter Davies, and Michael Farmwald. 1990. The TLB slice–a low-cost high-speed address translation mechanism. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*. New York, NY, 355–363.
[80] TIA 232 F 1997. TIA-232-F: Interface between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange. (1997).
[81] Ken Tindell, H. Hanssmon, and Andy J. Wellings. 1994. Analysing real-time communications: Controller area network (CAN). In *Proceedings of the Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, 259–263.
[82] USB 2.0 2000. *Universal Serial Bus Specification* (2.0 ed.). Universal Serial Bus Specification, Revision 2.0, April 27, 2000, Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips.
[83] USB 3.x 2017. *Universal Serial Bus Specification* (3.2 ed.). Universal Serial Bus 3.2 Specification, September 22, 2017, USB 3.0 Promoter Group (Apple Inc., Hewlett-Packard Inc., Intel Corporation, Microsoft Corporation, Renesas Corporation, STMicroelectronics, and Texas Instruments.
[84] VxWorks Accessed: April 2023. VxWorks Official Website. (Accessed: April 2023). Retrieved from http://www.windriver.com/products/vxworks.
[85] XHCI 2017. *eXtensible Host Controller Interface for Universal Serial Bus* (1.1 ed.). eXtensible Host Controller Interface for Universal Serial Bus (xHCI), May 2019, Revision 1.2, Intel Corporation.
[86] Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. 2008. Redline: First class support for interactivity in commodity operating systems. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*.
[87] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. 2014. COLORIS: A dynamic cache partitioning system using page coloring. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT'14)*. ACM, 381–392. DOI: https://doi.org/10.1145/2628071.2628104
[88] Yuting Zhang and Richard West. 2006. Process-aware interrupt scheduling and accounting. In *Proceedings of the 2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. IEEE, 191–201.
[89] Lin Zhao, Feng He, Ershuai Li, and Jun Lu. 2018. Comparison of time sensitive networking (TSN) and TTEthernet. In *Proceedings of the 2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*. 1–7. DOI: https://doi.org/10.1109/DASC.2018.8569454
[90] K. M. Zuberi and K. G. Shin. 1995. 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)*. IEEE Computer Society. Retrieved from http://dl.acm.org/citation.cfm?id=526671.828330.

Received 19 December 2022; revised 11 April 2023; accepted 28 May 2023