Tuned Pipes: End-to-end Throughput and Delay Guarantees for USB Devices

Ahmad Golchin, Zhuoqun Cheng and Richard West

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

***Abstract*—A fundamental problem in real-time computing is handling device input and output in a timely manner. For example, a control system might require input data from a sensor to be sampled and processed at a regular rate so that output signals to actuators occur within specific delay bounds. Input/output (I/O) devices connect to the host computer using different types of bus interfaces. One of the most popular interfaces in use today is the universal serial bus (USB). USB is now ubiquitous, in part due to its support for many classes of devices with simplified hardware needed to connect to the host. However, typical USB host controller drivers suffer from potential timing delays that affect the delivery of data between tasks and devices. Consequently, this paper introduces *tuned pipes*, a host controller driver and system framework that guarantees end-to-end latency and throughput requirements for I/O transfers. We expand on our earlier work involving USB 2.0 to support higher bandwidth USB 3.x communication. As a case study, we show how a USB-Controller Area Network (CAN) guarantees temporal isolation and end-to-end guarantees on communication between a set of peripheral devices and host tasks. A comparable USB-CAN bus setup using Linux is not able to achieve the same level of temporal guarantees, even when using SCHED_DEADLINE.**

I. Introduction

Embedded and real-time systems typically interact with their environment using sensors and actuators. For example, a control system for an autonomous vehicle might sample and process data from inertial sensors, cameras and LIDAR, to generate output signals for actuators such as motors, servos, switches and solenoids. Typically, these systems have delay bounds on the time between collecting sensor data and producing actuator outputs. This problem is compounded by the numerous bus interfaces used to connect sensors and actuators to the host computer responsible for data processing and control. Input/output (I/O) devices typically connect using PCIe, USB, CAN, I2C, or SPI buses. The host operating system is then responsible for the drivers necessary to talk to devices using these bus interfaces.

Unfortunately, most operating systems provide poor support for real-time I/O, with device drivers lacking temporal guarantees. For example, many systems such as Linux allow the processing of device interrupts to interfere with, and consume the timeslice of, the process context that was active at the time of the interrupt. Moreover, if the interrupt service routine (ISR) disables interrupts, a long-lived handler may cause subsequent interrupts to be lost. Similarly, an ISR that runs without disabling interrupts might be active at the time of another interrupt, leading to potential reentrancy problems.

Systems such as Linux handle long-lived ISRs by splitting them into two parts: (1) a *top* half that runs briefly when the interrupt occurs, and (2) a *bottom* half that performs the bulk 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 [1], [2]. Moreover, it does not ensure temporal isolation between interrupt handling and task execution.

Besides interrupts, the bus interface that connects I/O devices to the host computer must be correctly managed to ensure throughput and delay guarantees on the transfer of data. The universal serial bus (USB) has become an industry standard for its ability to support many different classes of devices with relatively simple hardware needed to connect to the host. The handling of interrupts, and direct memory access (DMA) control, for example, are addressed by the USB host controller rather than the device itself. This differs from other bus technologies, such as the peripheral component interconnect (PCI), where the device itself must include a PCI controller that manages interrupts and DMA transfers. While this has influenced the popularity of USB, most operating systems do not adequately implement a real-time host controller driver, to schedule access to the bus interface by different devices.

Our prior work [3] showed how Linux's approach to scheduling USB transactions led to rejections in real-time transfer requests, which would otherwise be possible given the available bus bandwidth. We also showed how a suitably written USB host controller driver is capable of providing throughput and delay guarantees to *bulk* devices that would otherwise be given best-effort service. However, our prior work was in the context of USB 2.0 and many modern computers are now equipped with USB 3.x host controllers.

Given the above, this paper introduces *tuned pipes*, a host controller driver and system framework that guarantees end-to-end latency and throughput requirements for I/O transfers between tasks and USB devices. We expand on our prior USB 2.0 work to briefly describe the similarities and differences we have experienced between USB 2.0 and USB 3.x.

Using a USB-Controller Area Network (CAN) as an example, we show how it is possible to provide temporal isolation and end-to-end guarantees on communication between a set of peripheral devices and host tasks. CAN controllers are now commonly used in many real-time domains, including automotive systems. Our example CAN controller uses a USB interface to connect to the host, thereby avoiding the need for a complex PCI adapter card that would be cumbersome in many space-constrained embedded applications. We show how the host operating system is able to consolidate CAN tasks, thereby performing end-to-end guarantees between the host and each device, in a way that is not possible on systems such as Linux.

In the following section, we briefly describe background to the I/O problem addressed in this paper. This is followed by Section III, which describes tuned pipes and their provision for end-to-end guarantees on data transfers between tasks and devices. An experimental evaluation of our approach is shown in Section IV. Related work is discussed in Section V, followed by conclusions and future work in Section VI.

II. Background

Providing real-time guarantees on I/O transfers between host memory and USB devices is problematic for several reasons. First, when the USB host controller completes a transfer, it may generate an interrupt on completion that needs to be handled in a timely manner. Second, a USB bus is shared amongst all devices attached to it, and transactions on that bus are scheduled according to the type of *endpoint* used by each device.

Regarding the first problem, it is unacceptable to simply allow the interrupt handler to execute in the kernel context of a preempted task, as in Linux. This causes the preempted task to be charged for CPU cycles that it does not use while the handler is executed. Here, the problem is that the interrupt handling is given precedence over the execution of a task, irrespective of the task priority. It is possible that repeated interrupts defer a high priority task by a sufficient amount to cause it to miss a deadline [1].

In systems such as Linux, the interrupt handling is split into a top and bottom half. The top half executes when the interrupt occurs, albeit with a small amount of dispatch latency. It acknowledges the interrupt before posting an event to perform potentially *deferrable* bottom half interrupt handling. The interrupt handling in a bottom half is not deferred unless multiple interrupts have occurred in a short time, thereby preventing the interrupted task from resuming progress. To address this problem, Linux defines a constant, `MAX_SOFTIRQ_RESTART` [1], which defaults to 10. This is the maximum number of times that bottom half processing is restarted in Linux before the cycle is broken and a preempted task is resumed. Once bottom half processing is deferred, it is handled in the context of a CPU-specific system task, *ksoftirqd*, when there are no other tasks to execute.

As we showed in prior work [1], this leads to a mismatch in the priority of interrupts and tasks. In many cases, interrupts occur as a result of I/O requests from tasks, and those interrupts should ideally be handled at the priority of the task that is blocked, or at least awaiting the completion of I/O on its behalf. We have addressed this problem in the design of the Quest real-time operating system [2], as we briefly explain in Section II-A.

[1]As named in the latest kernel version 4.18.x at the time of this paper.

Regarding the second problem, the USB host controller is responsible for ordering transactions, which request the transfer of data between a device and host memory. Systems such as Linux implement a *first-fit* approach to add transfer requests into the host controller schedule. We have shown in prior work [3] that a real-time USB host controller driver is able to accept transfer requests into its schedule that Linux rejects. A well-written USB host-controller driver should be able to correctly share bus bandwidth amongst devices that require throughput and delay guarantees, when it is possible to meet those requirements.

Part of the motivation for this paper is to provide throughput and delay guarantees on transfers between host tasks and USB-CAN devices. We are currently designing an automotive system that submits Controller Area Network (CAN bus) frames to a host where they are processed. A host equipped with a multicore processor and GPGPU has the capability to perform (semi-)autonomous vehicle control tasks, which would not be possible on the relatively low-powered electronic control units (ECUs) within a typical automobile of today. Our CAN bus interface connects to the host via USB to avoid the need for a bulky PCIe adapter card. However, the interface only supports bulk USB endpoints, which are traditionally only serviced in a best-effort manner *after* periodic transactions. We clarify the endpoint types supported by USB in Section II-B.

In what follows, we briefly summarize our prior work to address the two problems described above.

A. Task and Interrupt Scheduling

We developed a unified task and interrupt scheduling framework in our Quest real-time operating system. Top half interrupt processing is limited to acknowledging the interrupt, identifying the priority for the corresponding bottom half, and posting an event to wake up the bottom half task. We attempt to keep top half interrupt processing minimal, as it is system overhead that is charged to the running thread at the time of the interrupt.

In Quest, bottom half interrupt handlers and tasks execute as threads bound to time-budgeted virtual CPUs (VCPUs). Each VCPU is specified a processor capacity reserve [4], consisting of a budget capacity, C, and period, T, depending on the corresponding worst case execution time and period. A VCPU is eligible to receive up to C units of execution time every T 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.

There are two types of VCPUs in Quest. Conventional periodic threads are assigned to Main VCPUs, which are implemented as Sporadic Servers [5] and scheduled using Rate-Monotonic Scheduling (RMS) [6]. Interrupt bottom half threads are assigned to I/O VCPUs, which operate as bandwidth-preserving servers with a dynamically-calculated budget and period. 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.

When a device interrupt occurs, the thread associated with its occurrence is determined. For example, if some thread τ initiated an I/O request that led to the I/O 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 process I/O interrupts with the priority of the thread 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 operates as a Priority Inheritance Bandwidth-preserving Server (PIBS) [2], where a certain utilization factor U is specified to limit its bandwidth. For example, the budget of the I/O VCPU mentioned above will be limited to $T_{main} \cdot U$. 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 [7]. In prior work [2], 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:

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

B. Universal Serial Bus

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. These have maximum throughputs of 1.5, 12, 480Mbps and 5Gbps respectively. Recent advances to USB 3.x now increase bus bandwidth to 10 or even 20Gbps.

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 transfer mode, whether it is an input or output endpoint, the maximum packet size, how many packets it can receive during a single transaction,

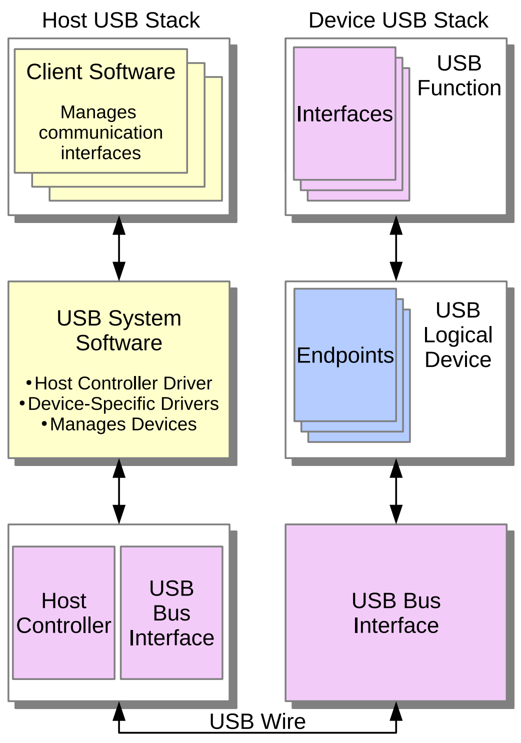


Fig. 1. USB Host and Device Stacks

and how often transactions should occur in the case of periodic endpoints.

The USB specification supports four basic *transfer types* for data exchange 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. All devices have at least one control endpoint.

USB *transactions* are always initiated by the bus master, 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. Transactions cannot cross a frame boundary. USB 2.0 and higher support micro-frames of 125 microseconds. Each frame contains eight micro-frames and transactions cannot cross a micro-frame boundary. The host controller will discard any transactions that span these micro-frame boundaries.

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.

C. Differences Between USB 2.0 and USB 3.x

The Enhanced Host Controller Interface (EHCI) [8] 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 [9], [10] limits the time available to schedule transactions from the periodic frame list before the circular asynchronous list is processed. For example, high-speed transfers are limited to 80% of a micro-frame, leaving at least 20% to asynchronous transfers. Each time the asynchronous list is processed, it resumes with the next transaction 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 full or super-speed devices, and that 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 tranmission 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) [11] 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, as we discovered when implementing support for a USB-CAN interface on a host computer with only the newer xHCI support. The periodic frame list and circular asynchronous list in EHCI are replaced with hardware-managed transfer rings in xHCI. To be precise, 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. These are shown in Figure 2.

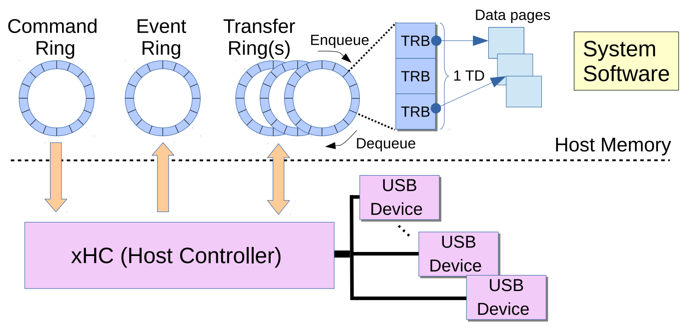


Fig. 2. Various Data Rings used by an eXtensible Host Controller (xHC)

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. The xHC allows isochronous transfers to specify the starting frame ID, at the granularity of 1ms. For transactions requiring service intervals of one or more micro-frames, it is possible to pack up to 8 isochronous 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.

In our experience with the Kvaser USBcan Pro-series of interfaces, we discovered they only support *bulk* endpoints. This seems counter-intuitive for a CAN bus that is intended for real-time systems. As a consequence, we focused on the development of tuned pipes for USB-CAN that tackled the problem of host device driver scheduling rather than USB bus scheduling. Notwithstanding, our xHCI host controller driver is able to implement a bin-packing scheduler to support multiple USB devices with bandwidth and delay constraints, similar to how we implemented our EHCI scheduler. In this way, we are able to reserve bus bandwidth for both asynchronous (e.g., bulk) and periodic endpoints by controlling the start frame ID of different TDs. The scheduling algorithm is shown below, with further details in our earlier work [3]. It has been slightly modified for xHCI, but operates similarly with EHCI.

Algorithm 1 Quest USB Scheduling Algorithm

```
$R \leftarrow$ array of $n$ USB requests (ie., TDs)
$A \leftarrow$ array of starting frames for the scheduled assignments
$T \leftarrow$ 1024 element array initially all zero
// $T[f]$ = time used in micro-frame $f$
$B \leftarrow 125000$

// Sort transactions by increasing interval,
// breaking ties by largest transmission delay first
$R \leftarrow$ SORT($R$)
for $i = 0$ to $n-1$ do
    $w_i \leftarrow$ TRANSMISSION_DELAY($R[i]$)
    $t_i \leftarrow$ INTERVAL($R[i]$)
    $A[i] \leftarrow -1$
    $j \leftarrow 0$
    while $A[i] = -1 \wedge j < t_i$ do
        $feasible \leftarrow$ TRUE
        $f \leftarrow j$
        while $f < 1024$ do
            if $T[f] + w_i > B$ then
                $feasible \leftarrow$ FALSE
            end if
            $f \leftarrow f + t_i$
        end while
        if $feasible$ then
            $A[i] \leftarrow \lfloor j/8 \rfloor$ // Request $i$ starts in frame $\lfloor j/8 \rfloor$
            $f \leftarrow j$
            while $f < 1024$ do
                $T[f] \leftarrow T[f] + w_i$
                $f \leftarrow f + t_i$
            end while
        end if
        $j \leftarrow j + 1$
    end while
    if $A[i] = -1$ then
        return FALSE
    end if
end for
return (TRUE, $A$)
```

The algorithm is provided with an array of n requests assigned to set R. Each request R_i is specified with a transmission delay w_i and period t_i in micro-frames. A successful request will start in micro-frame j and proceed to be serviced in micro-frame $(j + a{\cdot}t_i)\%1024$, where $a = 1, 2, \cdots$. The requirement is that the total transmission delay of one or more requests does not cross a micro-frame boundary, and that 8 micro-frames are available in a single frame. We have shown that by sorting requests by increasing interval, with ties being broken by servicing the largest transmission delay first, this tends to increase the likelihood of finding successful schedules. The alternative would be to exhaustively consider all possible permutations of requests, which is impractical for an online scheduler. Each time a request is feasible (i.e., can be scheduled), it is committed to an ordered array, A. A holds the starting frame ID for each scheduled request, because xHCI will not allow software to explicitly set the micro-frame for a TD.

III. Tuned Pipes

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. 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 3 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.

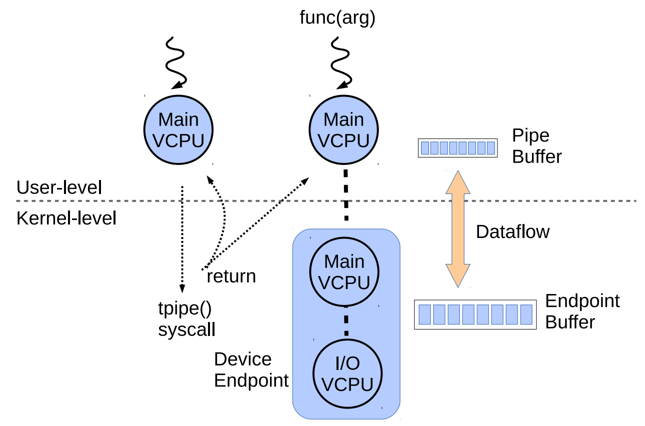


Fig. 3. Tuned Pipe Abstraction

A. Tuned Pipes API

The tuned pipe API allows user-space programs to establish a host-to-device communication channel with throughput and delay constraints. The pipe is guaranteed to be temporally isolated 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 3 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 is associated with an I/O VCPU (identified by `iovcpuid`) that 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 operating system, 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 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 quality of service (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 execution 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`. For `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 creation of a tuned pipe also requires the Main and I/O VCPUs to be feasibly scheduled according to the utilization bound test.

B. *End-to-end Guarantee*

Thanks to the temporal isolation provided by host scheduling, our system is able to guarantee end-to-end latency requirements for I/O transfers. In this section, we use a USB-CAN system as an example to elaborate on separate latencies that influence end-to-end time.

1) Latency Contributors: The end-to-end I/O 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 4. The most complex path considers the input of data, as this has to be demultiplexed for different user-level threads. We therefore omit further discussion of the output path, although it is largely the reverse control and data flow. 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.

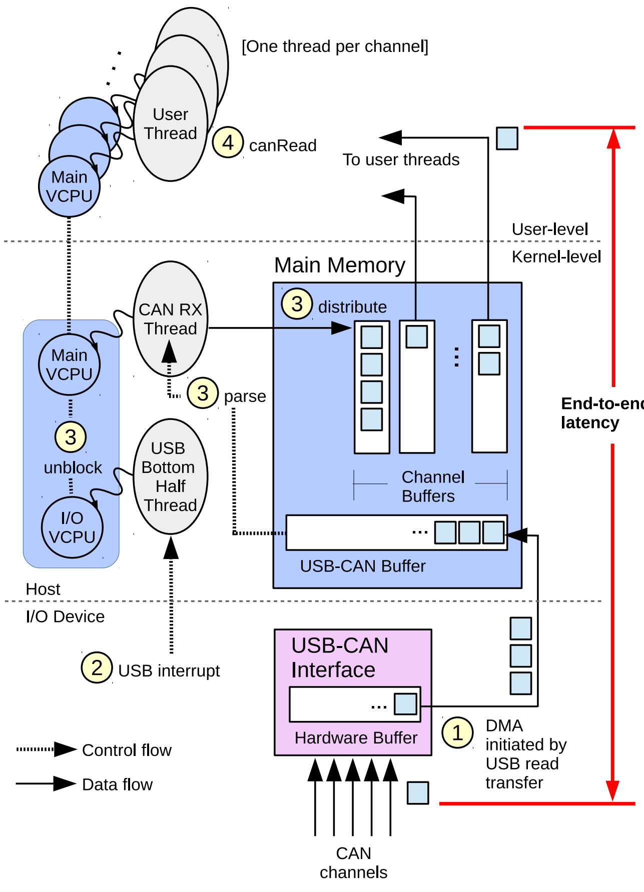


Fig. 4. Input Data and Control Flow for USB-CAN Tuned Pipes

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 using direct memory access (DMA). In the case of USB-CAN, the target is a buffer allocated within the CAN driver. The waiting time that data spends in the hardware buffer contributes to the end-to-end latency, represented by ① in Figure 4.

Instead of triggering an interrupt upon the completion of each DMA transfer, the USB controller, as required by the xHCI specification, periodically triggers interrupts to poll the completion of all the pending transfers. In order to minimize the time during which interrupts are disabled, we delegate the polling operation to a threaded bottom half, 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 wakes 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.

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. 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.

2) End-to-end Timing Analysis: In order to derive the end-to-end transfer latency, we begin by describing the timing properties of all the entities involved in the analysis. As shown in Figure 4, the CAN RX thread is associated with a Main VCPU, whose budget is denoted by C_{rx} and period by T_{rx}. The period of the USB interrupt is denoted by T_{usb}. The USB I/O VCPU is bounded by $U_{usb}\%$ CPU utilization. Its budget is $C_{usb} = U_{usb} \cdot T_{rx}$ and its period is $T_{usb} = T_{rx}$, because I/O VCPUs derive service constraints from the Main VCPU they serve when using PIBS. Each user-level application thread is assumed to be associated with a Main VCPU with C_{usr} budget and T_{usr} period.

The scenario that leads to the worst case latency is the summation of largest delays incurred by steps ①-④, as follows:

First, the CAN RX thread issues a DMA transfer request once every period T_{rx}, which contributes to the delay in step ①. Each transfer request causes data to be moved from the hardware buffer to the USB-CAN buffer. The assumption here is that the cost of step ① is bounded by T_{rx}, which is greater than the delay of a DMA transfer.

Second, the I/O VCPU associated with the bottom half of the xHCI driver is woken up every T_{usb} time units as a result of the periodic USB interrupts, which contributes to the worst-case delay in step ②. As multiple USB devices could be transferring data within the interval T_{usb}, we set C_{usb} to be sufficient to process all bus transactions in one period of the xHCI bottom half. As our host controller driver schedules bus transactions, it is possible to identify the number and type of transactions that occur in T_{usb}. In turn, it is then possible to derive the largest C_{usb} necessary to process those transactions.

Third, the worst-case overhead of the CAN RX thread, to parse the USB-CAN buffer and distribute messages into separate channel buffers 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.

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

$$\begin{aligned} E_{wc} &= T_{rx} + T_{usb} + T_{rx} + T_{usr} \\ &= 3 \cdot T_{rx} + T_{usr} \end{aligned} \tag{2}$$

Equation 2 is simplified as a result of T_{rx} being set equal to T_{usb}. In our system, T_{usb} is configured to $1\ ms$, which is the default value used by the USB host controller hardware. The 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 III-A. The same section also describes the way in which T_{usr} is determined.

IV. Experimental Evaluation

We conducted several sets of experiments on a testbed as shown in Figure 5. The testbed consisted of a Kvaser USBcan Pro 5xHS five channel CAN bus interface connected via USB 3.0 to an UP Squared single-board computer. The Up Squared had 4 GB RAM and a dual-core Celeron N3350 processor operating at 1.1 Ghz. For all experiments, we assigned all tasks and interrupts to a single core, rather than offloading I/O tasks to a dedicated and potentially low-utilization core.

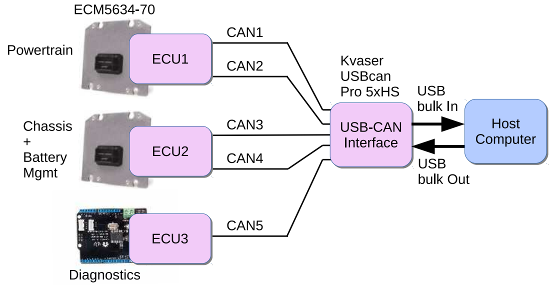


Fig. 5. Experimental Testbed

We mimicked the behavior of an automotive system by simulating CAN frames from several CAN devices acting as Electronic Control Units (ECUs). ECU1 and ECU2 were represented by Woodward MotoHawk ECM5634-70 modules, commonly used for engine and powertrain control functions in real automotive systems.

For the purposes of providing an easily reprogrammable source of CAN frames, we used an Arduino UNO with a SeeedStudio CAN-BUS Shield V1.2. This represented ECU3 and was connected to the fifth CAN channel (CAN5). Although a real automotive system might have numerous ECUs, our testbed was representative of different classes of vehicle control, such as for the powertrain, chassis, battery management, and diagnostics. Importantly, it demonstrated the use of tuned pipes to transfer data between CAN bus nodes and the host computer, which acted as a CAN concentrator. A CAN concentrator could theoretically process CAN frames for different subsystems without requiring the processing to be offloaded to numerous separate ECUs.

With the Kvaser USBcan Pro 5xHS interface, each CAN channel is associated with a separate bus having a maximum configurable bandwidth of 1 Mbps. We determined by empirical study that the USB-CAN interface had an internal buffer size of 4 KB. When the hardware buffer is full, the interface overwrites stale data with the latest incoming data.

A. Endpoint Guarantees

In the first experiment, we used the ECUs to generate CAN traffic at different rates to the host computer, to see if a USB-CAN endpoint was capable of receiving all frames without loss. Host tasks communicated with the Kvaser USB-CAN interface using the CANlib API [12]. We ported the core CANlib API calls for use with our Quest RTOS, to allow us to configure the timing properties of a bus and to read and write different channels.

We compared Quest with support for tuned pipes against Ubilinux, which included the PREEMPT-RT patch. The bandwidth of each bus was limited to the maximum bit rates shown in Table I. The actual steady-state throughputs, as a percentage of each channel's maximum configured bandwidth, are shown below the bandwidth values. This configuration has a maximum bandwidth aggregated across all channels of 288 KBps, resulting in a minimum of 14 ms to fill the 4 KB hardware buffer of the USB-CAN interface. 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 was set to the maximum 8 bytes.

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

TABLE I
CAN BUS TRAFFIC

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. We configured the RX thread to be scheduled under the SCHED_DEADLINE [13] policy with the runtime budget set to 2 ms and the period set to 14 ms. The runtime budget was derived by measuring the time for the RX thread to parse and distribute 4 KB data into separate host memory buffers for different endpoints. Included in the execution budget of the RX thread is the time to report the number of hardware buffer overruns. It should be noted that the period and deadline of a thread were set to the same value, for all cases where SCHED_DEADLINE was used. The period of 14 ms accounts for the shortest time to fill the 4 KB internal buffer in the USB-CAN interface.

For comparison with Quest, we setup an RX thread having the same parameters to run on a Main VCPU. The corresponding xHCI bottom half was assigned to a *schedulable* I/O VCPU with a utilization of 1%.

Table II 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. For I/O, we used 5 separate CAN reader tasks, one per channel. I/O-bound tasks ran at their default priorities in both Linux and Quest. For cases where CPU-bound tasks were involved, each task incremented a counter every $10\mu s$ and reported how far away the counter was from the expected value for a given budget and period over the duration of the experiment. CPU-bound tasks ran under SCHED_DEADLINE in Linux, each having a budget of 1 ms and period 7 ms. Quest used the same budget and period values for corresponding Main VCPUs. These periods yielded relatively higher priorities for the CPU-bound threads than the RX thread, potentially causing the greatest interference on I/O operations.

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 II
EXPERIMENTAL SCENARIOS

Table III shows that Quest did not experience any lost CAN packets, as there were no overruns of the USB-CAN 4 KB buffer for any of the experimental scenarios. Linux, however, experienced overruns in the two scenarios where there were higher priority CPU-bound tasks. These caused interference with the USB xHCI bottom half (softirq) in Linux. The softirq initially runs with the highest priority in the system, with interrupts enabled. However, if the softirq processing loops `MAX_SOFTIRQ_RESTART` times, as described in Section II, and still finds more softirqs to process due to high rate of interrupts, it will wake up a *ksoftirqd*. The ksoftirqd thread handles the remaining softirqs at a much lower priority. This used to be the lowest priority but in newer versions of Linux it is now set 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.

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

TABLE III
USB-CAN BUFFER OVERRUNS

Figure 6 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 incremented 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 we 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 error 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 unused budget of blocked tasks, and also the overhead of the task scheduler. In each case, Quest guarantees progress close to expected. Although not shown, Quest has the ability to allow tasks to use CPU cycles above their budget limits when no other tasks have available budgets. This actually allowed the CPU-bound tasks to increment their counters far in excess (more than 495000) of their target values.

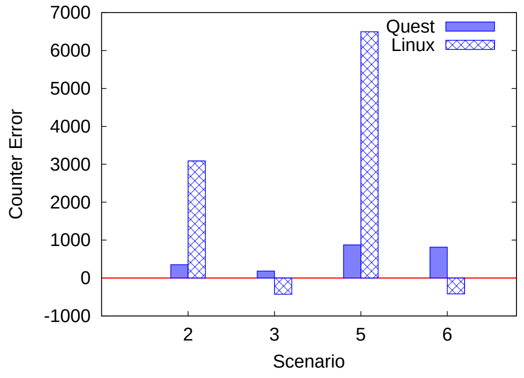


Fig. 6. Average Counter Error for CPU-bound Tasks

B. End-to-end Guarantees

Input Requests. To test the full benefits of tuned pipes, we conducted a second set of experiments that delivered data to user-space tasks according to throughput and delay constraints. The experimental setup was similar to that in Figure 5, except CAN4 was replaced with a second Arduino and CAN shield. Both CAN4 and CAN5 generated standard frames with a throughput of 69% of their 500 Kbps channel bandwidths. Five tasks on the Up Squared opened pipes to the USB-CAN devices, and recorded 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.

We used an oscilloscope to measure the minimum and maximum transmission delay of a CAN frame from each Arduino, which was observed to be 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 created a tuned pipe with the following QoS specification as defined in Section III-A: `latency`=1×10^9 ns, `tput`=2752 CAN frames per second, `IObufsz`=128 CAN frames, and `texec_time`=2 ms. The throughput (`tput`) was 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, we derived an I/O task period of 46 ms, which was the largest interval before a user-space buffer of 128 frames could overflow. We set the budget for each I/O task to be 2 ms, which was considered sufficient time to process up to 128 buffered frames. Having derived the budgets and periods, Quest I/O tasks were assigned to Main VCPUs, while equivalent Linux tasks were assigned to the SCHED_DEADLINE class.

Figure 7 shows the number of frames per second received by the two I/O tasks associated with the Arduino devices, for both Quest and Linux. We skipped the first second to allow I/O buffers to populate. Similarly, Figure 8 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 2730 to 2752 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, Linux fails to sustain an average throughput even above the minimum 2730 frames/s generated by Arduino. This is because Linux loses some CAN frames as observed in the previous experiment (Table III). With Linux, CAN4 and CAN5 averaged 2703 and 2714 frames per second, respectively. In Quest, CAN4 and CAN5 averaged 2745 and 2743 frames per second, respectively, without loss of packets.

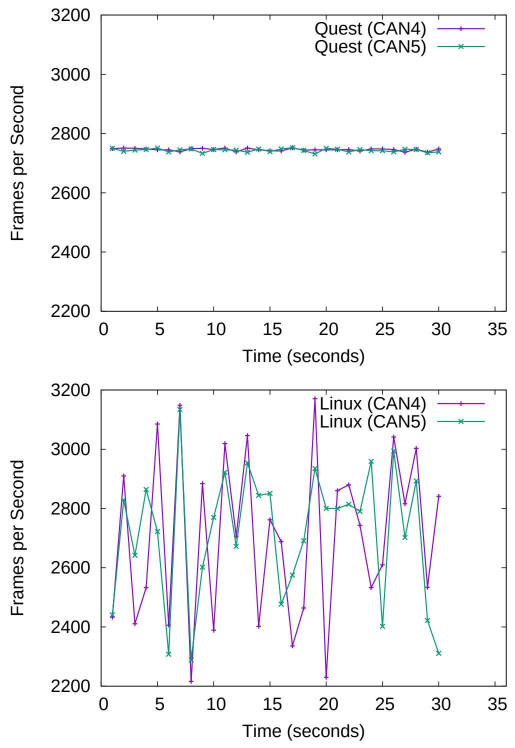


Fig. 7. Number of Frames per Second Received by Two User-Level I/O Tasks, with (a) Quest, and (b) Linux

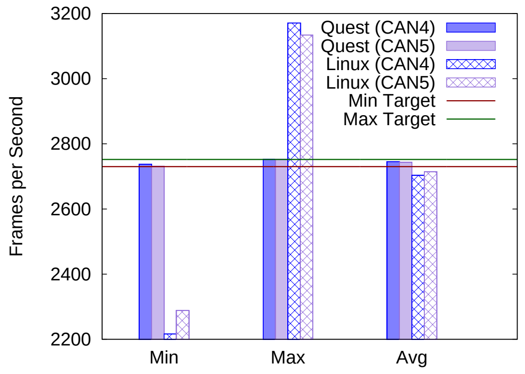


Fig. 8. Minimum, Maximum and Average (Input) Throughput of the Two CAN Channels using Linux and Quest

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

For Quest, we established a separate tuned pipe for CAN4 and CAN5 to output data to each Arduino device. As before, an oscilloscope was used to measure the latency of one iteration of an Arduino sketch, which toggled a GPIO pin upon reception of a CAN frame from the host. Details of the Arduino sketch are shown in Appendix A.

The observed minimum and maximum latencies for one iteration of the Arduino sketch were $325.4\mu s$ and $327.5\mu s$, respectively. These values were established while transferring data from the host as fast as possible to ensure the Kvaser USB-CAN interface buffer was always full. These measured latencies corresponded to a maximum and minimum throughput of 3073 and 3053 CAN frames per second, respectively.

We measured the latency to send data from a user-level host buffer for each tuned pipe. For a buffer size of 128 64-byte messages, the latency was no more than 2 milliseconds. Consequently, the QoS specification for each tuned pipe for CAN4 and CAN5 was set to: `latency`=1×10^9 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 had a derived budget and period of $C = 2$ms and $T = 41$ms, respectively. For comparison with Linux, we established a separate SCHED_DEADLINE thread for each channel having equivalent constraints to those of Quest's Main VCPUs.

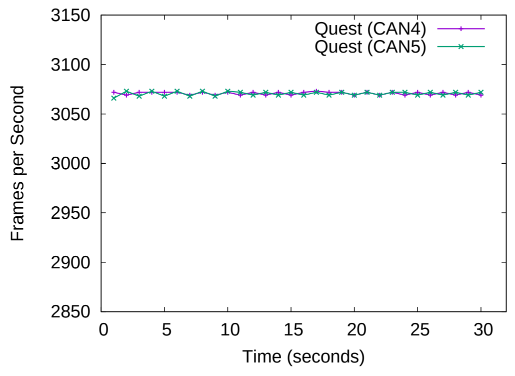
Figure 9(a) shows that Quest is able to transfer data from a host to each Arduino device via the USB-CAN interface according to the tuned pipe QoS specification. In comparison, Figure 9(b) shows that Linux sometimes experiences significant drops in throughput, below the minimum 3053 CAN frames per second expected for each Arduino device. This is because of demotion in 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 10 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.

V. Related Work

While many real-time operating systems exist today, it is less common to see systems with explicit support for temporal isolation using resource reservations or budgets. One early system that is built around the notion of resource reserves is Linux/RK [14]. Linux now supports the SCHED_DEADLINE scheduling policy, based on the Earliest Deadline First (EDF) and Constant Bandwidth Server (CBS) [15] algorithms, with resource reservations. Quest differs from the above systems, by focusing on the temporal isolation between tasks and interrupt handlers using a hierarchy of virtual servers, acting as either Main or I/O VCPUs.

The interrupt-handling mechanism in existing off-the-shelf operating systems is largely independent of process man-



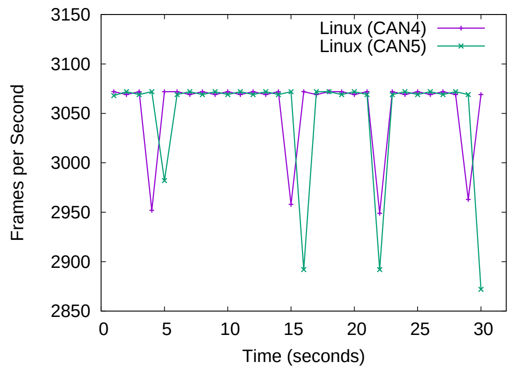


Fig. 9. Number of Frames per Second Sent by Two User-Level I/O Tasks, with (a) Quest, and (b) Linux

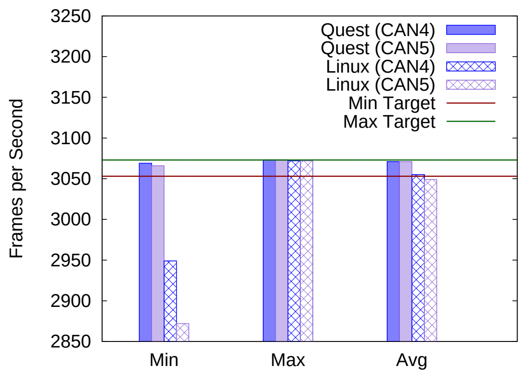


Fig. 10. Minimum, Maximum and Average (Output) Throughput of the Two CAN Channels using Linux and Quest

agement, which compromises the temporal predictability of the system. Several research works [16], [17], [18] have investigated the idea of integrating interrupt handling with the scheduling and accountability of processes. Leyva-del-Foyo *et al.* [19] proposed a unified mechanism for synchronization and scheduling of both interrupts and processes. Lewandowski *et al.* [20] considered bandwidth constraints on device driver execution. However, none of these works explore the dependency between interrupts and processes, and use that information to decide the priority of interrupts (essentially bottom halves) in CPU scheduling. Motivated by Zhang *et al.* [1], Quest combines the scheduling and accountability of interrupts associated with corresponding processes that issue service requests on I/O devices.

The temporal isolation between interrupt handlers and tasks is used in Quest's tuned pipes for I/O transfers. Scout [21] exposes paths that are similar to pipes in our system, as a way to offer quality of service 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.* [22], provide a method to guarantee end-to-end inter-process communication, instead of throughput and delay-constrained I/O transfers.

With the increasing popularity of open source hardware and maker communities, numerous libraries and APIs now exist to ease the interaction with I/O devices. Examples include the Arduino APIs [23] to manage a range of different devices, and the MRAA library [24] targeted at Intel and Raspberry Pi IoT devices. However, none of these libraries or APIs provide support for user-specified I/O timing properties. The tuned pipe API allows user-space programs to establish a host-to-device I/O pipe that has throughput and delay bounds.

VI. Conclusions and Future Work

This paper introduces *tuned pipes*, and abstraction for guaranteeing throughput and delay constraints on the transfer of information between USB devices and host tasks. We have implemented a full xHCI host controller driver in Quest that supports tuned pipes. We tested our system with a port of the *mhydra* USB-CAN driver and showed that I/O service constraints are possible with Quest but not with a comparable Linux system. This was even the case for a Linux system supporting appropriately tuned device driver tasks serviced using the SCHED_DEADLINE policy.

Future work includes the development of a tuned pipe abstraction in our sister Quest-V system [25]. USB 3.x provides support for host controller virtualization so it is possible to have Quest real-time system services sharing a single host controller and bus with legacy Linux services. The aim is to use Quest for tasks that require real-time I/O, while Linux provides legacy support for devices drivers, libraries and services that need not be real-time. We see this as being beneficial to cyber-physical applications, ranging from driverless cars to autonomous drones and beyond.

VII. Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1527050. 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 National Science Foundation.

Thanks to Drako Motors for providing the equipment for our testbed, and additional support for this research.

Supporting code for this paper will be made available for download from `www.questos.org`.

References

[1] Y. Zhang and R. West, "Process-Aware Interrupt Scheduling and Accounting," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, ser. RTSS '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 191–201.

[2] M. Danish, Y. Li, and R. West, “Virtual-CPU Scheduling in the Quest Operating System,” in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2011, pp. 169–179.

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

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

[5] B. Sprunt, “Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System,” Software Engineering Institute, Carnegie Mellon, Tech. Rep. CMU/SEI-89-TR-011, 1989.

[6] C. L. Liu and J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment,” *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[7] E. Missimer, K. Missimer, and R. West, “Mixed-Criticality Scheduling with I/O,” in *28th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2016, pp. 120–130.

[8] *Enhanced Host Controller Interface Specification for Universal Serial Bus*, 1st ed., March 2002.

[9] *Universal Serial Bus Specification*, 2nd ed., April 27 2000.

[10] *Universal Serial Bus Specification*, 3rd ed., September 22 2017.

[11] *eXtensible Host Controller Interface for Universal Serial Bus*, 1st ed., November 2017.

[12] CANlib: CAN bus API, https://www.kvaser.com.

[13] Linux SCHED_DEADLINE Policy: https://www.kernel.org/doc/Documentation/scheduler/sched-deadline.txt.

[14] S. Oikawa and R. Rajkumar, “Linux/RK: A Portable Resource Kernel in Linux,” in *Proc. 19th IEEE Real-Time Systems Symposium*, 1998.

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

[16] S. Kleiman and J. Eykholt, “Interrupts As Threads,” *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 2, pp. 21–26, Apr. 1995.

[17] J. Regehr, “HLS: A Framework for Composing Soft Real-time Schedulers,” in *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, 2001, pp. 3–14.

[18] T. Facchinetti, G. Buttazzo, M. Marinoni, and G. Guidi, “Non-preemptive Interrupt Scheduling for Safe Reuse of Legacy Drivers in Real-time Systems,” in *17th Euromicro Conference on Real-Time Systems (ECRTS’05)*, July 2005, pp. 98–105.

[19] L. E. L. del Foyo, P. Mejia-Alvarez, and D. de Niz, “Predictable Interrupt Management for Real Time Kernels over conventional PC Hardware,” in *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS’06)*, April 2006, pp. 14–23.

[20] M. Lewandowski, M. J. Stanovich, T. P. Baker, K. Gopalan, and A. I. A. Wang, “Modeling Device Driver Effects in Real-Time Schedulability Analysis: Study of a Network Driver,” in *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS’07)*, April 2007, pp. 57–68.

[21] D. Mosberger and L. L. Peterson, “Making Paths Explicit in the Scout Operating System,” in *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI ’96. New York, NY, USA: ACM, 1996, pp. 153–167.

[22] R. Pineiro, K. Ioannidou, S. A. Brandt, and C. Maltzahn, “RAD-FLOWS: Buffering for Predictable Communication,” in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2011, pp. 23–33.

[23] Arduino Language Reference: http://arduino.cc/en/Reference/ HomePage.

[24] MRAA Library: http://mraa.io.

[25] R. West, Y. Li, E. Missimer, and M. Danish, “A Virtualized Separation Kernel for Mixed Criticality Systems,” *ACM Transactions on Computer Systems*, vol. 34, 2016.

Appendix A
Arduino Receiver Sketch

```
#include <SPI.h>
#include <mcp_can.h>

const int SPI_CS_PIN = 10;
MCP_CAN CAN(SPI_CS_PIN); // Set CS pin


#define CAN_SCOPE_PIN       3
#define CAN_REC_DELAY       4

int lastState = LOW;
int cnt = 0;
int msg_per_sec[31];
int cur_sec_indx = 0;
unsigned long last_ms;
bool start = false;

void setup(){
    Serial.begin(9600);
    // init can bus baudrate = 500k
    while (CAN_OK != CAN.begin(CAN_500KBPS)){
      Serial.println("CAN BUS Shield init fail");
      Serial.println("Init CAN BUS Shield again");
      delay(100);
    }
    Serial.println("CAN BUS Shield init ok!");

    pinMode(CAN_SCOPE_PIN, OUTPUT);
    digitalWrite(CAN_SCOPE_PIN, LOW);

    pinMode(CAN_REC_DELAY, OUTPUT);
    digitalWrite(CAN_REC_DELAY, LOW);

    int i;
    for (i = 0 ; i < 31 ; i++)
      msg_per_sec[i] = 0;

    Serial.println("Waiting for the 1st msg to start...");
    while (CAN_MSGAVAIL != CAN.checkReceive()){}
    last_ms = millis();
}

void loop(){
  unsigned char len = 0;
  unsigned char buf[8];
  char strOut[16];
  digitalWrite(CAN_REC_DELAY, HIGH);
  unsigned long ms = millis();

  if (ms - last_ms > 1000){
    cur_sec_indx++;
    last_ms = ms;
  }
  if (cur_sec_indx >= 31){
    Serial.println("# of messages per second in first 31s:");
    int i;
    for (i = 0 ; i < 31 ; i++){
      sprintf(strOut, "%d ,", msg_per_sec[i]);
      Serial.print(strOut);
    }
    delay(10000); //delay for 10 seconds
  }else{
  if(CAN_MSGAVAIL == CAN.checkReceive()){
    CAN.readMsgBuf(&len, buf);
    int canId = CAN.getCanId();
    digitalWrite(CAN_SCOPE_PIN, lastState);
    lastState = !lastState;
    msg_per_sec[cur_sec_indx]++;
    delayMicroseconds(218); /*data proc delay*/
    }
  }
  digitalWrite(CAN_REC_DELAY, LOW);
}
```