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

Ahmad Golchin, Zhuoqun Cheng and Richard West
Boston University
Motivations

- Cyber-physical systems
- Ubiquity of USB
- Sensor-actuator loops
- Need for predictable I/O communication
  - Between device & application tasks
- Avoid manually fine-tuning system parameters for control & data flow
Contributions

- Tuned Pipes system framework
  - Guarantees end-to-end latency and throughput requirements between USB devices and host tasks
- A host controller driver with early demultiplexing
  - Allows USB bottom-half handler to run with the right priority and in a timely manner as opposed to Linux
- Extended our previous USB bus scheduling algorithm to comply with xHCI
Quest RTOS

- Real-time OS supporting multicore x86 platforms
  - Intel’s Aero, UP, UP2, Skull Canyon, Edison, Minnowmax,...
- Dual-mode kernel
- Unified task and I/O (bottom-half) scheduling through time-budgeted virtual CPUs (VCPUs)
  - Tasks scheduling: Main VCPUs
  - Interrupt bottom-half scheduling: I/O VCPUs
- More info: www.questos.org
VCPU Scheduling in Quest RTOS

- Main VCPUs
  - Sporadic Server + RMS
  - Guarantees budget $C$ every period $T$ for tasks
- I/O VCPUs
  - PIBS
  - BW limited by utilization factor $U_j$
  - Inherits $T$ from the task
- Temporal isolation condition:
  \[
  \sum_{i=0}^{n-1} \frac{C_i}{T_i} + \sum_{j=0}^{m-1} (2 - U_j) \cdot U_j \leq n(\sqrt{2} - 1)
  \]
Tuned Pipes

- Host-to-device communication channel
- Throughput and delay bounds (QoS)
- Temporal isolation
- Endpoint-pipe: 1:N registered by drivers
Tuned Pipes - User-level API
Tuned Pipes - User-level API

tpipe()
Tuned Pipes - User-level API

```
tpipe()
```

Callback

```
func(arg)
```

User-level
Kernel-level

Dataflow

```
tpipe()
syscall
```

Device Endpoint

Pipe Buffer

Endpoint Buffer
Tuned Pipes - User-level API

tpipe()

Main VCPU

Callback

Pipe Buffer

Dataflow

QoS

Endpoint Buffer

User-level
Kernel-level

tpipe() syscall

Device Endpoint

return

func(arg)
Tuned Pipes - User-level API

QoS Specification:

- Execution Time (C)
- Throughput (\(\lambda\))
- IO Buffer Size (B)
Tuned Pipes - User-level API

QoS Specification:

- Execution Time (C)
- Throughput (λ)
- IO Buffer Size (B)

Example:

tput = 500Kbps
IObufsize = 128 bytes
texec_time = 1 ms
Tuned Pipes - User-level API

QoS Specification:

- Execution Time (C)
- Throughput ($\lambda$)
- IO Buffer Size (B)

Example:

* tput = 500Kbps
* IObufsize = 128 bytes
* texec_time = 1 ms

Little’s law: $B = \lambda T$
Tuned Pipes - User-level API

QoS Specification:

- Execution Time (C)
- Throughput (λ)
- IO Buffer Size (B)

Example:

\[ \text{tput} = 500\text{Kbps} \]
\[ \text{IObufsize} = 128 \text{ bytes} \]
\[ \text{texec\_time} = 1 \text{ ms} \]

Little’s law: \( B = \lambda T \)

Main VCPU Parameters

\[ C = 1\text{ms} \]
\[ T = \frac{128\times8}{512000} = 2\text{ms} \]
Tuned Pipes - Kernel API

Endpoint:

- Endpoint attributes
- IOVCPU & sched param
- MainVCPU & sched param

Endpoint Attributes:

- Max # of Channels
- Max Throughput
- Min Latency
- Min/Max EP Buffer Size
- Min/Max Packet Size
Tuned Pipes - Kernel API

Example

● 4 channels at 500Kbps
● 1 channel at 250Kbps
● max_tput = 2.25Mbps
● ebuf_sz = 4KB
● Driver applies Little’s law to set proper budget and period for it’s I/O thread:
● E.g.: C = 2ms, T= 14ms
End-to-end Rx Data Path

4 Delay contributors
- User thread
- Driver thread
- DMA of data
- USB bottom-half

Question:
How to enforce QoS?
End-to-end Rx Data Path

4 Delay contributors
- User thread
- Driver thread
- DMA of data
- USB bottom-half

Question:
How to enforce QoS?
End-to-end Rx Data Path

4 Delay contributors
- User thread
- Driver thread
- DMA of data
- USB bottom-half

Question:
How to enforce QoS?
End-to-end Rx Data Path

4 Delay contributors
- User thread
- Driver thread
- DMA of data
- USB bottom-half

Question:
How to enforce QoS?
End-to-end Rx Data Path

4 Delay contributors
- User thread
- Driver thread
- DMA of data
- USB bottom-half

Question:
How to enforce QoS?
Challenges with Linux:

- USB BUS scheduling
- USB bottom-half handler priority mismatch!

What currently happens:

- Soft-IRQs
  Highest priority until MAX_SOFTIRQ_RESTART → Low priority

- Threaded-IRQs (e.g. PREEMPT_RT)
  Fixed SCHED_FIFO priority (Default: 50)
Experimental Environment

CAN Interface
- Kvaser USBcan Pro 5xHS
- 5 channels: up to 1Mbps w/ 4KB buffer
- 2 ECUs: each exposing 2 channels
- 1 Arduino UNO + CAN-BUS Shield
Experimental Environment

UPSquared SBC
- Dual-core Celeron N3350 @ 1.1 GHz
- xHCI 1.1 Interface

Quest RTOS
- VCPU Scheduling
- Ubilinux (PREEMPT_RT)
- SCHED_DEADLINE
Test 1 - Endpoint Guarantees

Objective: Receiving frames without:
● Loss of CAN packets
● Intervening with other tasks of higher priority

Generated data traffic:

<table>
<thead>
<tr>
<th>Bus</th>
<th>CAN1</th>
<th>CAN2</th>
<th>CAN3</th>
<th>CAN4</th>
<th>CAN5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandwidth (bps)</td>
<td>500K</td>
<td>250K</td>
<td>500K</td>
<td>500K</td>
<td>500K</td>
</tr>
<tr>
<td>Throughput %</td>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>69</td>
</tr>
</tbody>
</table>
Test 1 - Endpoint Guarantees

3 CPU-bound Tasks

Main VCPU

Main VCPU

Main VCPU

... 5 CAN-Read Tasks

User-level

Kernel-level

4KB Buffer

5 Channel USB-CAN Endpoint

Main VCPU

I/O VCPU
Test 1 - Endpoint Guarantees

C = 1ms  T = 7ms

C = 2ms  T = 14ms
Test 1 - Endpoint Guarantees

Observations:

● Quest:
  ○ No buffer overrun
  ○ Negligible interference

● Linux:
  ○ 230 overruns over 30 seconds
  ○ 405 overruns over 60 seconds
  ○ More interference
Test 2 - End-to-end Guarantees - Rx

Objective: Guaranteeing throughput using tuned pipes

- 5 Tuned pipes receiving data
- CAN 4 & 5 Throughput: 2730 to 2752 fps
- QoS: tput=2752, IObufsz=128, exec_time=2ms

<table>
<thead>
<tr>
<th>Bus</th>
<th>CAN1</th>
<th>CAN2</th>
<th>CAN3</th>
<th>CAN4</th>
<th>CAN5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandwidth (bps)</td>
<td>500K</td>
<td>250K</td>
<td>500K</td>
<td>500K</td>
<td>500K</td>
</tr>
<tr>
<td>Throughput %</td>
<td>10</td>
<td>20</td>
<td>30</td>
<td><strong>69</strong></td>
<td><strong>69</strong></td>
</tr>
</tbody>
</table>
Test 2 - End-to-end Guarantees - Rx
Test 2 - End-to-end Guarantees - Rx

![Graph showing frames per second for different conditions and targets.]
Conclusions

- Tuned pipes abstraction
- Auto-tuning of system parameters
- Guarantee of throughput and delay constraints
  - Not solved with SCHED_DEADLINE in Linux
- Early demultiplexing of entities waiting for INT
- Handling BH with the RIGHT priority (IOVCPU)
  - Not solved with PREEMPT_RT Linux patch
Thank you!

Comments or Questions?
Test 3 - End-to-end Guarantees - Tx

Objective: Guaranteeing throughput using tuned pipes

Similar to the previous test, except:
● CAN 4 & 5 Receiving data every 325.4 to 327.5 uS
● Arrival rate: 3053 to 3073
● QoS: tput=3073, IObufsz=128, exec_time=2ms
Test 3 - End-to-end Guarantees - Tx

![Graphs showing frames per second over time for Quest (CAN4) and Quest (CAN5) on the left, and Linux (CAN4) and Linux (CAN5) on the right. The graphs demonstrate consistent performance for Quest, whereas Linux shows fluctuations.]
Test 3 - End-to-end Guarantees - Tx

Frames per Second

- Quest (CAN4)
- Quest (CAN5)
- Linux (CAN4)
- Linux (CAN5)

Min Target
Max Target

Min | Max | Avg