

# End-to-End Analysis and Design of a Drone Flight Controller

Zhuoqun Cheng, Richard West, *Senior Member, IEEE*, and Craig Einstein

**Abstract**—Timing guarantees are crucial to embedded and cyber-physical applications that must bound the end-to-end delay between sensing, processing and actuation. For example, in a flight controller for a multirotor drone, the data from a gyro or inertial sensor must be gathered and processed to determine the attitude of the aircraft. Sensor data fusion is followed by control outputs that alter rotor speeds to adjust the drone’s flight. If the processing pipeline between sensor input and actuation is not bounded, the drone will lose control and possibly fail to maintain flight. This paper describes a composable pipe model for sensor data processing and actuation tasks. The pipe model is used to analyze two end-to-end semantics: 1) freshness and 2) reaction time. We provide a mathematical framework to derive feasible task periods and budgets that satisfy both schedulability and end-to-end timing requirements. We demonstrate the applicability of our design approach by using it to port the Cleanflight flight controller firmware to our in-house real-time operating system called Quest. Experiments show that Cleanflight ported to Quest is able to achieve end-to-end latencies within the predicted time bounds derived by analysis.

**Index Terms**—End-to-end timing analysis, flight controller, real-time systems.

## I. INTRODUCTION

COMMON to many embedded and cyber-physical applications is a communicating task *pipeline*, to acquire and process sensor data and produce outputs that control actuators. For example, a multirotor unmanned aerial vehicle (also known as UAV or drone) typically processes inertial sensors to estimate its current attitude,<sup>1</sup> and then uses a control function to adjust rotor speeds, which adapt the flight path.

The objective of this paper is to develop a system that supports real-time task pipelines with end-to-end timing guarantees. As a case study, we focus on the implementation of a real-time flight controller that is the basis for an autonomous

drone. Our first step is to refactor a popular flight control firmware, called Cleanflight [1] for use with our in-house real-time operating system (RTOS) [2]. Cleanflight [1] is used on racing drones that are operated by humans via radio-control. By porting Cleanflight to run as a real-time application on Quest, we have the opportunity to integrate it with additional functionality necessary for a fully autonomous flight management system. The aim is to replace radio-control with on-board tasks that perform configurable flight missions, such as aerial photography, package delivery, and search and rescue.

Using a multirotor drone flight controller as an example, the challenge is to determine the required task timing constraints necessary for end-to-end latency guarantees. Sensor data must be sampled and processed at a minimum rate to ensure the drone is operating according to the most recent estimates of its attitude, direction, altitude, and speed. Similarly, the rotor speeds must be updated within certain time bounds to be relevant to the current sensor readings. For this reason, we define two end-to-end timing constraints, in terms of sensor data *freshness* and actuator *reaction* time that constrain the problem of composing a feasible task pipeline.

In Quest, each pipelined task is a separate thread of control, mapped to a virtual CPU (VCPU). A VCPU is guaranteed  $C$  time units of service every period,  $T$ , when its corresponding thread is runnable. This is achieved by treating every VCPU as a bandwidth-preserving server [3]. Thus, for a set of tasks in a pipeline, each mapped to a separate VCPU, the problem is to determine a valid set of budgets and periods that ensure end-to-end freshness and reaction times, while satisfying the schedulability of all VCPUs.

The contributions of this paper are: 1) a composable pipe model that guarantees bounded end-to-end processing and communication delay amongst a set of independently activated tasks assigned to bandwidth-preserving servers; 2) a method to derive task periods and budgets from given end-to-end timing constraints in the application design stage; and 3) the reimplementation and evaluation of the Cleanflight flight controller on the Quest RTOS.

The following section provides background to the execution model used in this paper. Section III describes the end-to-end timing analysis of our proposed composable pipe model. Section IV shows how the end-to-end time is leveraged in the application design stage, while Section V details the reimplementation and evaluation of Cleanflight on the Aero board. Related work is discussed in Section VI, followed by the conclusions and future work in Section VII.

Manuscript received April 3, 2018; revised June 8, 2018; accepted July 2, 2018. Date of publication July 18, 2018; date of current version October 18, 2018. This work was supported in part by the Intel Corporation and in part by the National Science Foundation under Grant 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 NSF. This article was presented in the International Conference on Embedded Software 2018 and appears as part of the ESWEEK-TCAD special issue. (*Corresponding author: Richard West.*)

The authors are with the Computer Science Department, Boston University, Boston, MA 02215 USA (e-mail: czq@bu.edu; richwest@cs.bu.edu; einstein@bu.edu).

Color versions of one or more of the figures in this paper are available online at <http://ieeexplore.ieee.org>.

Digital Object Identifier 10.1109/TCAD.2018.2857399

<sup>1</sup>The attitude is the orientation of the drone relative to a reference frame, such as earth.

## II. EXECUTION MODEL

This paper is motivated by our objective to implement an autonomous flight management system for multirotor drones. An autonomous drone is one that is able to reason about and adapt to changes in its surroundings, while accomplishing mission objectives without remote assistance from a human being. As part of this effort, we have undertaken a port of the Cleanflight firmware to the Quest RTOS. The core software components of Cleanflight consist of sensor and actuator drivers, a PID controller, the Mahony attitude and heading reference (AHRS) algorithm, various communication stacks, and a logging system.

### A. Task Model

We model the flight controller program as a set of real-time periodic tasks  $\{\tau^1, \tau^2, \dots, \tau^n\}$ , whose initial release times are arbitrary. Each task  $\tau^j$  is characterized by its worst-case execution time  $e^j$ , period  $T^j$ , and a deadline that is equal to period. Additionally, each task communicates with zero or more successors and predecessors, to exchange and process data.

$e^j$  and  $T^j$  are determined during the design stage and are fixed at runtime.  $e^j$  is usually profiled offline under the worst-case execution condition. Deciding the value of  $T^j$  is a challenging process, which is the major topic of this paper. It mainly depends on the end-to-end latency constraints and the schedulability test. All the periodic tasks are implemented using user-level threads.

Apart from user-level threads, there are kernel threads dedicated to I/O interrupts, which originate primarily from the SPI and I2C bus in Cleanflight. Quest executes interrupts in a deferrable thread context, having a corresponding time budget. This way, the handling of an interrupt does not steal CPU cycles from a currently running, potentially time-critical task.

### B. Scheduling Model

Threads in Quest are scheduled by a two-level scheduling hierarchy, with threads mapped to VCPUs that are mapped to physical CPUs. Each VCPU is specified a processor capacity reserve [4] consisting of a budget capacity,  $C$ , and period,  $T$ . The value of  $C$  and  $T$  are determined by the  $e$  and  $T$  of the task mapped to the VCPU. A VCPU is required to receive at least  $C$  units of execution time every  $T$  time units when it is runnable, as long as a schedulability test [5] is passed when creating new VCPUs. This way, Quest's scheduling subsystem guarantees temporal isolation between threads in the runtime environment.

Conventional periodic tasks are assigned to main VCPUs, which are implemented as Sporadic servers [3] and scheduled using rate-monotonic scheduling (RMS) [6]. The VCPU with the smallest period has the highest priority. Instead of using the Sporadic server model for both main tasks and bottom half threads, special I/O VCPUs are created for threaded interrupt handlers. Each I/O VCPU operates as a Priority Inheritance Bandwidth preserving Server (PIBS) [7]. A PIBS uses a single replenishment to avoid fragmentation of replenishment list budgets caused by short-lived interrupt service routines

(ISRs). By using PIBS for interrupt threads, the scheduling overheads from context switching and timer reprogramming are reduced [8].

### C. Communication Model

Control flow within the flight controller is influenced by the path of data, which originates from sensory inputs and ends with actuation. Inputs include inertial sensors, optional cameras, and GPS devices, while actuators include motors that affect rotor speeds and the attitude of the drone. Data flow involves a pipeline of communicating tasks, leading to a communication model characterized by: 1) the interarrival times of tasks in the pipeline; 2) intertask buffering; and 3) the tasks' access pattern to communication buffers.

1) *Periodic Versus Aperiodic Tasks*: Aperiodic tasks have irregular interarrival times, influenced by the arrival of data. Periodic tasks have fixed interarrival times and operate on whatever data is available at the time of their execution. A periodic task implements asynchronous communication by not blocking to await the arrival of new data.

2) *Register-Based Versus FIFO-Based Communication*: A first-input, first-output (FIFO)-based shared buffer is used in scenarios where *data history* is an important factor. However, in a flight controller, *data freshness* outweighs the preservation of the full history of all sampled data. For example, the motor commands should always be computed from the latest sensor data and any stale data should be discarded. Moreover, FIFO-based communication results in loosely synchronous communication: the producer is suspended when the FIFO buffer is full and the consumer is suspended when the buffer is empty. Register-based communication achieves fully asynchronous communication between two communicating parties using Simpson's four-slot algorithm [9].

3) *Implicit Versus Explicit Communication*: Explicit communication allows access to shared data at any time point during a task's execution. This might lead to data inconsistency in the presence of task preemption. A task that reads the same shared data at the beginning and the end of its execution might see two different values, if it is preempted between the two reads by another task that changes the value of the shared data. Conversely, the implicit communication model [10] essentially follows a read-before-execute paradigm to avoid data inconsistency. It mandates a task to make a local copy of the shared data at the beginning of its execution and to work on that copy throughout its execution. Simpson's four-slot algorithm [9] ensures the *read* and *write* stages avoid data corruption without blocking.

This paper assumes a periodic task model, as this simplifies timing analysis. Applications, such as Cleanflight implement periodic tasks to sample data and perform control operations. Quest also adopts register-based, implicit communication for data freshness and consistency.

## III. END-TO-END TIMING ANALYSIS

In this section, we first distinguish two different timing semantics for end-to-end communication, which will be used as the basis for separate timing analyses. Second, we develop

TABLE I  
EXAMPLE PERIODS

	Gyro Period	Motor Period
Case 1	10 ms	1 ms
Case 2	1 ms	10 ms

a composable pipe model for communication, which is derived from separate latencies that influence end-to-end delay. Lastly, we use the pipe model to derive the worst-case end-to-end communication time under various situations.

### A. Semantics of End-to-End Time

To understand the meaning of end-to-end time, consider the following two constraints for a flight controller.

*Constraint 1:* A change in motor speed must be within 2 ms of the gyro sensor reading that caused the change.

*Constraint 2:* An update to a gyro sensor value must be within 2 ms of the corresponding update in motor speed.

The values before and after a *change* differ, whereas they may stay the same before and after an *update*. These semantics lead to two different constraints. To appreciate the difference, imagine the two cases in Table I. In case 1, the task that reads the gyro runs every 10 ms and the one that controls the motors runs every 1 ms. Case 1 is guaranteed to meet constraint 1 because the motor task runs more than once within 2 ms, no matter whether the gyro reading changes. However, it fails constraint 2 as the gyro task is not likely to run even once in an interval of 2 ms. Conversely, case 2 is guaranteed to meet constraint 2 but fails constraint 1 frequently.

This example demonstrates the difference between the two semantics of end-to-end time, defined as follows.

- 1) *Reaction time* is the time it takes for a sample of input data to flow through the system, and is affected by the period of each consumer in a pipeline. A reaction timing constraint bounds the time interval between a sensor input and the *first corresponding* actuator output.
- 2) *Freshness time* is the interval within which a sampled data input has influence on the system, and is affected by the period of each producer in a pipeline. A freshness timing constraint bounds the interval between a sensor input and the *last corresponding* actuator output.

Constraint 1, above, is a constraint on reaction time, while constraint 2 is on freshness time. We perform analysis of the two semantics of time in Sections III-E2 and III-E3, respectively.

### B. Latency Contributors

The end-to-end communication delay is influenced by several factors, which we will identify as part of our analysis. To begin, we first consider the end-to-end communication pipeline illustrated as a task chain in Fig. 1. Task  $\tau_1$  reads input data  $D_{in}$  over channel  $Ch_{in}$ , processes it and produces data  $D_1$ . Task  $\tau_2$  reads  $D_1$  and produces  $D_2$ , and  $\tau_3$  eventually writes output  $D_{out}$  to channel  $Ch_{out}$  after reading and processing  $D_2$ .

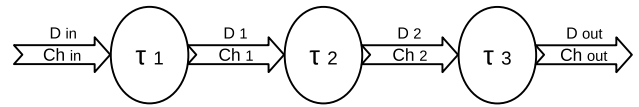


Fig. 1. Task chain.

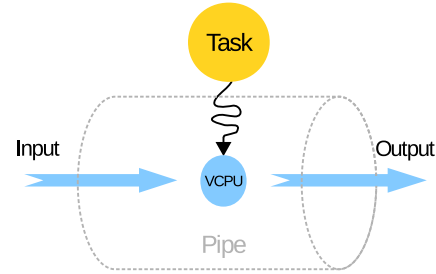


Fig. 2. Illustration of a pipe.

Each task handles data in three stages, i.e., read, process, and write. The end-to-end time should sum the latency of each stage in the task chain. Due to the asynchrony of communication, however, we also need to consider one less obvious latency, which is the waiting time it takes for an intermediate output to be read in as input, by the succeeding task in the chain. In summary, the latency contributors are classified as follows.

- 1) *Processing latency* represents the time it takes for a task to translate a raw input to a processed output. The actual processing latency depends not only on the absolute processing time of a task without interruption, but also on the service constraints (i.e., CPU budget and period of the VCPU) associated with the task.
- 2) *Communication latency* represents the time to transfer data over a channel. The transfer data size, channel bandwidth, propagation delay, and communication protocol overheads all contribute to the overall latency. Since our communication model is asynchronous and register-based as described in Section II, queuing latency is not a concern of this paper.
- 3) *Scheduling latency* represents the time between the arrival of data on a channel from a sending task and when the receiving task begins reading that data. The scheduling latency depends on the order of execution of tasks in the system, and therefore has significant influence on the end-to-end communication delay.

### C. Composable Pipe Model

In Section III-B, we identified the factors that influence end-to-end communication delay. Among them, the absolute processing time and the transfer data size are determined by the nature of the task in question. To capture the rest of the timing characteristics, we develop a composable pipe model, leveraging the scheduling approach described in Section II-B. A task and pipe have a one-to-one relationship, as illustrated in Fig. 2.

A pipe has one pipe terminal (PT) and two *pipe ends*, with one end for input and the other for output. A PT is represented by a VCPU, guaranteeing at least  $C$  units of execution time every  $T$  time units when runnable. PTs are associated with

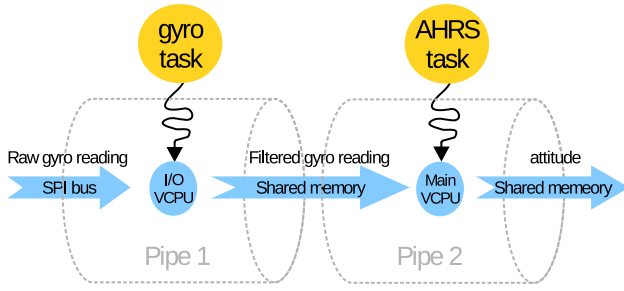


Fig. 3. Illustration of two communicating pipes.

conventional tasks bound to main VCPUs and kernel control paths (including interrupt handlers and device drivers) bound to I/O VCPUs, as described in Section II-B.

A pipe end is an interface to a communication channel, which is either an I/O bus or shared memory. The pipe end's propagation delay is assumed negligible, while the transmission delay is modeled by the bandwidth parameter,  $W$ , of the communication channel.  $\delta$  is used to denote the software overheads of a communication protocol. Though we are aware that  $\delta$  depends on the data transfer size, the time difference is negligible, compared to the time of actual data transfer and processing. Therefore, for the sake of simplicity,  $\delta$  is a constant in our model.

Note that in our definition of a single pipe there is only one terminal, not two at either end of the pipe. This differs from the idea of a POSIX pipe, which comprises a task at both sending and receiving ends. In our case, a pipe is represented by the single terminal that takes input and produces output.

An example of two communicating pipes is shown in Fig. 3. This is representative of a communication path between a gyro task and attitude calculation in Cleanflight on the Aero board. The gyro task is mapped to Pipe 1, whose input end is over the SPI bus connected to the gyro sensor and output end is over a region of memory shared with Pipe 2. Pipe 1's terminal is an I/O VCPU because the gyro task is responsible for handling I/O interrupts generated from the SPI bus. On the contrary, the terminal of Pipe 2 is a Main VCPU as the AHRS task is CPU-intensive. The gyro task reads raw gyro readings from Pipe 1's input end, processes them, and writes filtered gyro readings to Pipe 1's output end. Similarly, the AHRS task reads the filtered gyro readings from Pipe 2's input end and produces attitude data for its output end over shared memory.

1) *Notation:* The timing characteristics of a pipe are denoted by the 3-tuple,  $\pi = ((W_i, \delta_i), (C, T), (W_o, \delta_o))$ , where:

- $(W_i, \delta_i)$  and  $(W_o, \delta_o)$  denote the bandwidth and software overheads of the input and output ends, respectively;
- $(C, T)$  denotes the budget and period of the PT.

A task  $\tau$  is also denoted as a 3-tuple,  $\tau = (d_i, p, d_o)$ , where:

- $d_i$  denotes the size of the raw data that is read in by  $\tau$  in order to perform its job, and  $d_o$  denotes the size of the processed data that is produced by  $\tau$ ;
- $p$  denotes the uninterrupted processing time it takes for  $\tau$  to turn the raw data into the processed data.

In addition,  $\tau \mapsto \pi$  denotes the mapping between task  $\tau$  and pipe  $\pi$ . A task  $\tau = (d_i, p, d_o)$  is said to be mapped to a pipe  $\pi = ((W_i, \delta_i), (C, T), (W_o, \delta_o))$  when:

- 1) data of size  $d_i$  is read from the input end with parameters  $(W_i, \delta_i)$ , and data of size  $d_o$  is written to the output end with parameters  $(W_o, \delta_o)$ ;
- 2) the PT with parameters  $(C, T)$  is used for scheduling and accounting of the read and write operations, as well as the processing that takes time  $p$ .

For the composition of a chain of pipes, the operator  $|$  connects a pipe's output end to its succeeding pipe's input end. For example, Fig. 3 is represented as  $\tau_{\text{gyro}} \mapsto \pi_1 | \tau_{\text{AHRS}} \mapsto \pi_2$ . The scheduling latency between two pipes is denoted by  $S_{\tau \mapsto \pi | \tau' \mapsto \pi'}$ . Lastly, given a task set  $T = \{\tau_1, \tau_2, \dots, \tau_n\}$  identity mapped to a pipe set  $\Pi = \{\pi_1, \pi_2, \dots, \pi_n\}$ , where pipes are connected to each other in ascending order of subscript,  $E_{\tau_1 \mapsto \pi_1 | \tau_2 \mapsto \pi_2 | \dots | \tau_n \mapsto \pi_n}$  denotes the end-to-end reaction time of the pipe chain, and  $F_{\tau_1 \mapsto \pi_1 | \tau_2 \mapsto \pi_2 | \dots | \tau_n \mapsto \pi_n}$  denotes the end-to-end freshness time.

#### D. Reachability

Before mathematically analyzing end-to-end time, we introduce the concept of *reachability*, inspired by the *data-path reachability conditions* proposed by Feiertag *et al.* [11]. The need to consider reachability is due to a subtle difference between our register-based asynchronous communication model and the traditional FIFO-based synchronous communication. In the latter, data is guaranteed to be transferred without loss or repetition. This way, end-to-end time is derived from the time interval between the arrival of a data input and the departure of its *corresponding* data output. Unfortunately, this might result in an infinitely large end-to-end time in the case of register-based asynchronous communication, where not every input leads to an output. Instead, unprocessed input data might be discarded (overwritten) when newer input data is available, as explained in Section II-C.

An infinitely large end-to-end time, while mathematically correct, lacks practical use. Therefore, the following timing analysis ignores all input data that fails to "reach" the exit of the pipe chain it enters. Instead, only those data inputs that result in data outputs from the pipe chain are considered. We define this latter class of inputs as being *reachable*.

#### E. Timing Analysis

As alluded to above, the execution of a task is divided into three stages, involving: 1) reading; 2) processing; and 3) writing data. To simplify the timing analysis, we assume that tasks are able to finish the read and write stages within one period of the PT, to which the task is mapped. This is not unrealistic for applications, such as a flight controller, because: 1) data to be transferred is usually small and 2) all three stages are typically able to finish within one period. However, to maintain generality, we do not impose any restriction on the length of the processing stage.

1) *Worst-Case End-to-End Time of Single Pipe:* First, we consider the case, where there is a single pipe. Two key observations for this case are: 1) the absence of scheduling latency

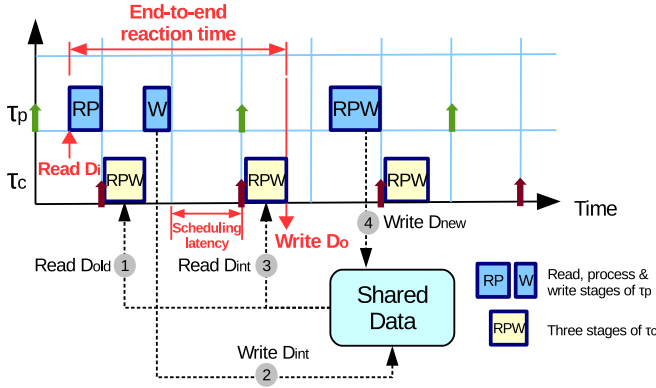


Fig. 4. End-to-end reaction time in case 1.

due to the lack of a succeeding pipe and 2) the equivalence of the two end-to-end time semantics (reaction and freshness time) due to the lack of a preceding pipe. We therefore use  $L_{\tau \rightarrow \pi}$  to unify the notation of  $E_{\tau \rightarrow \pi}$  and  $F_{\tau \rightarrow \pi}$ .

Given task  $\tau = (d_i, p, d_o)$  mapped to pipe  $\pi = ((W_i, \delta_i), (C, T), (W_o, \delta_o))$ , the worst-case end-to-end time is essentially the execution time of the three stages of  $\tau$  on  $\pi$ . Due to the timing property of  $\pi$ 's PT,  $\tau$  is guaranteed  $C$  units of execution time within any window of  $T$  time units. Hence, the worst-case latency  $L_{\tau \rightarrow \pi}^{wc}$  is bounded by the following:

$$L_{\tau \rightarrow \pi}^{wc} = \left\lfloor \frac{\Delta_{in} + p + \Delta_{out}}{C} \right\rfloor \cdot T + (\Delta_{in} + p + \Delta_{out}) \bmod C \quad (1)$$

where  $\Delta_{in} = (d_i/W_i) + \delta_i$  and  $\Delta_{out} = (d_o/W_o) + \delta_o$ .

2) *Worst-Case End-to-End Reaction Time of Pipe Chain:* In this section, we extend the timing analysis of a single pipe to a pipe chain. For the sake of simplicity, we start with a chain of length two. We show in Section III-F that the mathematical framework is applicable to arbitrarily long pipe chains. To distinguish the tasks mapped to the two pipes, we name the preceding task producer and the succeeding consumer. The producer is denoted by  $\tau_p = (d_i^p, p^p, d)$  and its pipe is denoted by  $\pi_p = ((W_i^p, \delta_i^p), (C^p, T^p), (W_o^p, \delta_o^p))$ . Similarly, the consumer task and pipe are denoted by  $\tau_c = (d, p^c, d_o^c)$  and  $\pi_c = ((W_i^c, \delta_i^c), (C^c, T^c), (W_o^c, \delta_o^c))$ . Following the definition of the end-to-end reaction time,  $E_{\tau_p \rightarrow \pi_p | \tau_c \rightarrow \pi_c}$ , in Section III-A, we investigate the time interval between a specific instance of input data, denoted by  $D_i$ , being read by  $\tau_p$ , and its first corresponding output, denoted by  $D_o$ , being written by  $\tau_c$ .

It is of vital importance to recognize that end-to-end time of a pipe chain is not simply the sum of the end-to-end time of each single pipe in the chain. We also need to account for the scheduling latency resulting from each appended pipe. As described in Section III-B, the scheduling latency depends on the order of execution of tasks. We, therefore, perform the timing analysis under two complementary cases: case 1— $\tau_c$  has shorter period and thus higher priority than  $\tau_p$  and case 2— $\tau_p$  has shorter period and thus higher priority than  $\tau_c$ , according to rate-monotonic ordering.

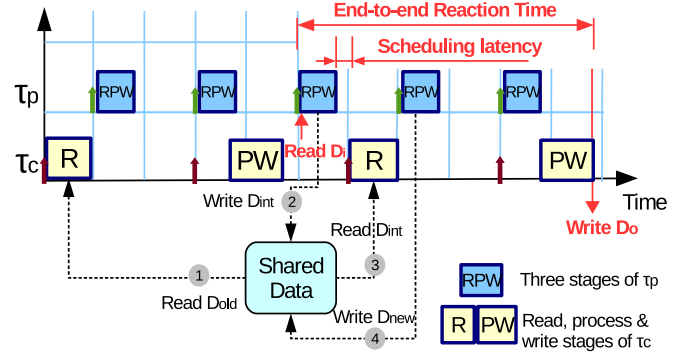


Fig. 5. End-to-end reaction time in case 2.

#### a) Calculating the end-to-end reaction time:

*Case 1:* the key to making use of  $L_{\tau_p \rightarrow \pi_p}^{wc}$  and  $L_{\tau_c \rightarrow \pi_c}^{wc}$  in the timing analysis of  $E_{\tau_p \rightarrow \pi_p | \tau_c \rightarrow \pi_c}^{wc}$ , is to find the worst-case scheduling latency,  $S_{\tau_p \rightarrow \pi_p | \tau_c \rightarrow \pi_c}^{wc}$ . As illustrated in Fig. 4, the worst-case scheduling latency occurs when  $\tau_c$  preempts  $\tau_p$  (step ①) immediately before  $\tau_p$  produces the intermediate output  $D_{int}$  corresponding to  $D_i$ . After preemption,  $\tau_c$  uses up  $\pi_c$ 's budget and gives the CPU back to  $\tau_p$ . Upon being resumed,  $\tau_p$  immediately produces  $D_{int}$  (step ②). For  $\tau_c$  to become runnable again to read  $D_{int}$  in step ③, it has to wait for its budget replenishment. The waiting time is exactly the worst-case scheduling latency

$$S_{\tau_p \rightarrow \pi_p | \tau_c \rightarrow \pi_c}^{wc} = T^c - C^c - \left( \frac{d}{W_o^p} + \delta_o^p \right). \quad (2)$$

After replenishment,  $\tau_c$  reads in  $D_{int}$ , processes it and eventually writes out  $D_o$ . As  $E_{\tau_p \rightarrow \pi_p | \tau_c \rightarrow \pi_c}$  is defined to be the time interval between the arrival of  $D_i$  and the departure of  $D_o$ , the worst case of  $E_{\tau_p \rightarrow \pi_p | \tau_c \rightarrow \pi_c}$  is as follows:

$$\begin{aligned} E_{\tau_p \rightarrow \pi_p | \tau_c \rightarrow \pi_c}^{wc} &= L_{\tau_p \rightarrow \pi_p}^{wc} + S_{\tau_p \rightarrow \pi_p | \tau_c \rightarrow \pi_c}^{wc} + L_{\tau_c \rightarrow \pi_c}^{wc} \\ &= L_{\tau_p \rightarrow \pi_p}^{wc} + L_{\tau_c \rightarrow \pi_c}^{wc} + T^c - C^c - \left( \frac{d}{W_o^p} + \delta_o^p \right). \end{aligned} \quad (3)$$

Note that if  $\tau_c$  runs out of budget before writing  $D_o$ ,  $\tau_p$  may overwrite  $D_{int}$  in the pipe with new data (step ④). However, the *implicit communication* property guarantees that  $\tau_c$  only works on its local copy of the shared data, which is  $D_{int}$  until  $\tau_c$  initiates another read.

*Case 2:* The situation is more complicated when  $\tau_p$  has higher priority than  $\tau_c$ . The worst-case scenario in case 1 does not hold in case 2 primarily because  $\tau_p$  might overwrite  $D_{int}$  before  $\tau_c$  has a budget replenishment. This is impossible in case 1 because  $\tau_p$  has a larger period than  $\tau_c$ , which is guaranteed to have its budget replenished before  $\tau_p$  is able to initiate another write. In other words, in Fig. 4, step ③ is guaranteed to happen before step ④.

The data-overwrite problem in case 2 is the reason for introducing reachability in Section III-D. To find the worst-case end-to-end reaction time in this case, we have to find the scenario that not only leads to the worst-case scheduling latency, but also originates from a reachable input. Fig. 5 illustrates a scenario that meets these requirements. In



the figure,  $\tau_p$  preempts  $\tau_c$  immediately after  $\tau_c$  finishes reading  $\tau_p$ 's intermediate output (step ③),  $D_{\text{int}}$ , corresponding to  $D_i$ . It follows that the longest possible waiting time, between  $D_{\text{int}}$  becoming available (step ②) and  $\tau_c$  reading the data (step ③), is the period of  $\tau_p$  minus both its budget and the execution time of the read stage of  $\tau_c$ . This waiting time is exactly the worst-case scheduling latency

$$S_{\tau_p \rightarrow \pi_p | \tau_c \rightarrow \pi_c}^{\text{wc}} = T^p - C^p - \left( \frac{d}{W_i^c} + \delta_i^c \right). \quad (4)$$

Between reading  $D_{\text{int}}$  and writing  $D_o$ ,  $\tau_c$  might experience more than one preemption from  $\tau_p$ , which repeatedly overwrites the shared data. This will not, however, affect  $\tau_c$ 's processing on  $D_{\text{int}}$  either spatially or temporally, thanks to the VCPU model and the implicit communication semantic. Similar to case 1, the worst-case end-to-end reaction time is again the sum of (1) of each pipe and (4)

$$\begin{aligned} E_{\tau_p \rightarrow \pi_p | \tau_c \rightarrow \pi_c}^{\text{wc}} &= L_{\tau_p \rightarrow \pi_p}^{\text{wc}} + S_{\tau_p \rightarrow \pi_p | \tau_c \rightarrow \pi_c}^{\text{wc}} + L_{\tau_c \rightarrow \pi_c}^{\text{wc}} \\ &= L_{\tau_p \rightarrow \pi_p}^{\text{wc}} + L_{\tau_c \rightarrow \pi_c}^{\text{wc}} + T^p - C^p - \left( \frac{d}{W_i^c} + \delta_i^c \right). \end{aligned} \quad (5)$$

Since the output end of  $\tau_p$  and the input end of  $\tau_c$  share the same communication channel, it is reasonable to assume that  $W_o^p = W_i^c$  and  $\delta_o^p = \delta_i^c$ . With that, we proceed to unify the worst-case end-to-end reaction time as follows:

$$E_{\tau_p \rightarrow \pi_p | \tau_c \rightarrow \pi_c}^{\text{wc}} = \begin{cases} T^c - C^c - \left( \frac{d}{W} + \delta \right) \\ \quad + L_{\tau_p \rightarrow \pi_p}^{\text{wc}} + L_{\tau_c \rightarrow \pi_c}^{\text{wc}}, & \text{if } T^c < T^p \\ T^p - C^p - \left( \frac{d}{W} + \delta \right) \\ \quad + L_{\tau_p \rightarrow \pi_p}^{\text{wc}} + L_{\tau_c \rightarrow \pi_c}^{\text{wc}}, & \text{otherwise.} \end{cases} \quad (6)$$

where  $W = W_o^p = W_i^c$  and  $\delta = \delta_o^p = \delta_i^c$ .

b) *Special Cases*: Real-time systems are often profiled offline to obtain worst-case execution times of their tasks. In our case, this would enable CPU resources for PTs to be provisioned so that each task completes one iteration of all three stages (read, process, write) in one budget allocation and, hence, period. This implies that  $\Delta_{\text{in}} + p + \Delta_{\text{out}} + \epsilon = C$  in (1), where  $\epsilon$  is an arbitrarily small positive number to account for surplus budget after completing all task stages. With that, it is possible to simplify the worst-case end-to-end reaction time derived in Section III-E2a. First, (1) is simplified as follows:

$$\begin{aligned} L_{\tau_c \rightarrow \pi_c}^{\text{wc}} &= \lfloor \frac{C - \epsilon}{C} \rfloor \cdot T + [(C - \epsilon) \bmod C] \\ &= 0 \cdot T + (C - \epsilon) \approx C. \end{aligned} \quad (7)$$

Using (7), (5) reduces to

$$\begin{aligned} E_{\tau_p \rightarrow \pi_p | \tau_c \rightarrow \pi_c}^{\text{wc}} &= T^p - C^p - \Delta_{\text{in}}^c + L_{\tau_p \rightarrow \pi_p}^{\text{wc}} + L_{\tau_c \rightarrow \pi_c}^{\text{wc}} \\ &= T^p - C^p - \Delta_{\text{in}}^c + C^p + C^c \\ &= T^p + C^c - \Delta_{\text{in}}^c. \end{aligned} \quad (8)$$

The same simplification applied to (3) of case 1 reduces (6) to

$$E_{\tau_p \rightarrow \pi_p | \tau_c \rightarrow \pi_c}^{\text{wc}} = \begin{cases} T^c + C^p - \Delta, & \text{if } T^c < T^p \\ T^p + C^c - \Delta, & \text{otherwise} \end{cases} \quad (9)$$

where  $\Delta = (d/W) + \delta$ .

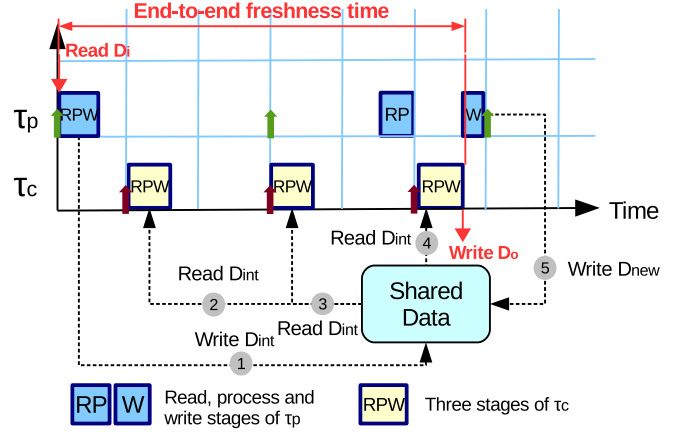


Fig. 6. End-to-end freshness time in case 1.

If we further assume that  $\pi_p$  and  $\pi_c$  communicate data of small size over shared memory, it is possible to discard communication overheads, such that  $\Delta = 0$ . With that, (9) simplifies to

$$E_{\tau_p \rightarrow \pi_p | \tau_c \rightarrow \pi_c}^{\text{wc}} = \begin{cases} T^c + C^p, & \text{if } T^c < T^p \\ T^p + C^c, & \text{otherwise.} \end{cases} \quad (10)$$

Finally, notice that by appending  $\tau_c \mapsto \pi_c$  to  $\tau_p \mapsto \pi_p$ , the worst-case end-to-end reaction time is increased by the following:

$$\begin{aligned} \uparrow E^{\text{wc}} &= E_{\tau_p \rightarrow \pi_p | \tau_c \rightarrow \pi_c}^{\text{wc}} - E_{\tau_p \rightarrow \pi_p}^{\text{wc}} \\ &= E_{\tau_p \rightarrow \pi_p | \tau_c \rightarrow \pi_c}^{\text{wc}} - C^p \\ &= \begin{cases} T^c, & \text{if } T^c < T^p \\ T^p - C^p + C^c, & \text{otherwise.} \end{cases} \end{aligned} \quad (11)$$

3) *Worst-Case End-to-End Freshness Time of Pipe Chain*: Techniques similar to those in Section III-E2 will be used to analyze end-to-end freshness time. To avoid repetition, we abbreviate the end-to-end freshness timing analysis by only focusing on the special cases described in Section III-E2b.

Recall that freshness time is defined to be the interval between the arrival of an input and the departure of its *last corresponding* output. Therefore, we investigate the interval between a specific instance of input data,  $D_i$ , being read by  $\tau_p$  and its *last corresponding* output,  $D_o$ , being written by  $\tau_c$ .

*Case 1*: As illustrated in Fig. 6,  $D_i$  is read by the first instance of  $\tau_p$  at time 0 and the intermediate output,  $D_{\text{int}}$ , is written to the shared data (step ①). After that,  $\tau_c$  produces three outputs corresponding to  $D_{\text{int}}$  (steps ②–④), or to  $D_i$  indirectly. The last output,  $D_o$ , is the one preceding  $\tau_p$ 's write of new data,  $D_{\text{new}}$  (step ⑤). Thus, the worst-case end-to-end freshness time,  $F_{\tau_p \rightarrow \pi_p | \tau_c \rightarrow \pi_c}^{\text{wc}}$ , occurs when: 1) the two consecutive writes (steps ① and ⑤) from  $\tau_p$  have the longest possible time interval between them and 2) the write of  $D_o$  happens as late as possible. The latest time to write  $D_o$  is immediately before the second write of  $\tau_p$ , which is preempted by higher priority  $\tau_c$ .

From Fig. 6 that the worst-case end-to-end freshness time is

$$F_{\tau_p \rightarrow \pi_p | \tau_c \rightarrow \pi_c}^{\text{wc}} = 2 \cdot T^p - \Delta_{\text{out}}^p. \quad (12)$$

Again, when communicating over shared memory, (12) can be further simplified to

$$F_{\tau_p \mapsto \pi_p | \tau_c \mapsto \pi_c}^{wc} = 2 \cdot T^p. \quad (13)$$

*Case 2:* When  $\tau_p$  has a smaller period than  $\tau_c$ , it is impossible for  $\tau_c$  to read the same intermediate output of  $\tau_p$  twice. In Fig. 6, step ⑤ is guaranteed to happen before ③. Thus, the worst-case freshness time (WCF) is essentially the worst-case reaction time (WCR), shown in (5).

In summary, the worst-case end-to-end freshness latency of two communicating pipes is represented in the following conditional equation:

$$F_{\tau_p \mapsto \pi_p | \tau_c \mapsto \pi_c} = \begin{cases} 2 \cdot T^p, & \text{if } T^c < T^p \\ T^p + C^c, & \text{otherwise.} \end{cases} \quad (14)$$

### F. Composability

The timing analysis for two pipes in Section III-E2 extends to pipe chains of arbitrary length. Every time an extra task  $\tau_{\text{new}}$  (mapped to  $\pi_{\text{new}}$ ) is appended to the tail end of a chain ( $\tau_{\text{tail}} \mapsto \pi_{\text{tail}}$ ), the worst-case end-to-end reaction time increases by the worst-case end-to-end time of the newly appended pipe, plus the scheduling latency between the new pipe and the tail pipe. The actual value of the increase, depending on the relative priority of the new pipe and the tail pipe, is shown in (11). Similarly, the added end-to-end freshness time can be derived from (14).

Composability is a crucial property of our pipe model, since it significantly eases the end-to-end time calculation for any given pipeline. This provides the basis for a design framework that derives task periods from given end-to-end timing constraints. This is detailed in the following section.

## IV. END-TO-END DESIGN

There are significant challenges to porting a flight control firmware, such as Cleanflight to run as a multithreaded application on an RTOS. One of the major issues is how to determine the period of each thread so that the application is able to meet its end-to-end timing constraints. A naive approach would be to start by choosing a tentative set of periods and use the timing analysis method in Section III to validate the timing correctness. Upon failure, the periods are heuristically adjusted and the validation step is repeated until end-to-end timing guarantees are met. This approach, however, is potentially time-consuming and labor-intensive when the number of tasks or constraints increase.

Inspired by Gerber *et al.* [12], we derive task periods from end-to-end timing constraints, by combining the timing analysis of the pipe model with linear optimization techniques. In this section, we generalize our method for use with a broader spectrum of cyber-physical control applications.

### A. Problem Definition

Consider a set of tasks  $\Gamma = \{\tau_1, \tau_2, \dots, \tau_n\}$  and a set of pipes  $\Pi = \{\pi_1, \pi_2, \dots, \pi_n\}$ , where  $\tau_j = (d_i^j, p^j, d_o^j)$  and  $\pi_j = ((W_i^j, \delta_i^j), (C^j, T^j), (W_o^j, \delta_o^j))$ . We additionally require the following information.

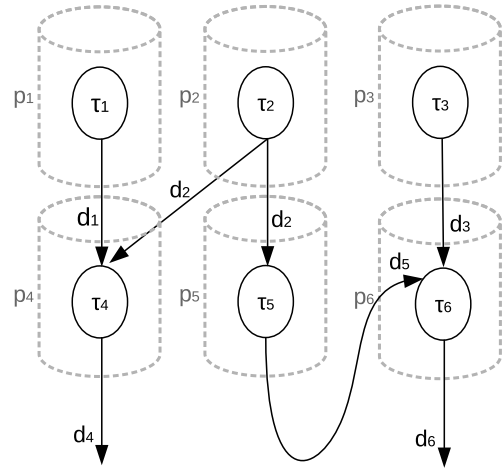


Fig. 7. Application task graph.

TABLE II  
APPLICATION TIMING CHARACTERISTICS

Reaction	Freshness
$E_{\tau_1 \mapsto \pi_1   \tau_4 \mapsto \pi_4} \leq 10,$ $E_{\tau_2 \mapsto \pi_2   \tau_4 \mapsto \pi_4} \leq 15,$ $E_{\tau_2 \mapsto \pi_2   \tau_5 \mapsto \pi_5   \tau_6 \mapsto \pi_6} \leq 25,$ $E_{\tau_3 \mapsto \pi_3   \tau_6 \mapsto \pi_6} \leq 15;$	$F_{\tau_1 \mapsto \pi_1   \tau_4 \mapsto \pi_4} \leq 20,$ $F_{\tau_2 \mapsto \pi_2   \tau_4 \mapsto \pi_4} \leq 30,$ $F_{\tau_2 \mapsto \pi_2   \tau_5 \mapsto \pi_5   \tau_6 \mapsto \pi_6} \leq 50,$ $F_{\tau_3 \mapsto \pi_3   \tau_6 \mapsto \pi_6} \leq 20;$
Schedulability	Execution Time
$\sum_{j=1}^6 \frac{C^j}{T^j} \leq 6(\sqrt[6]{2} - 1)$	$\forall j \in \{1, 2, \dots, 6\},$ $d_i^j = d_o^j = 3,$ $W_i^j = W_o^j = 20,$ $\delta_i^j = \delta_o^j = 0.1,$ $p^j = 0.5;$

- 1) The mapping between  $\Gamma$  and  $\Pi$ . For ease of notation, we assume tasks map to the pipe with the same subscript, hence  $\forall j \in \{1, 2, \dots, n\}, \tau_j \mapsto \pi_j$ .
- 2) The topology of  $\Pi$  (an example is shown in Fig. 7);
- 3)  $\forall j \in \{1, 2, \dots, n\}$ , the value of  $d_i^j$ ,  $p^j$ , and  $d_o^j$ .
- 4)  $\forall j \in \{1, 2, \dots, n\}$ , the value of  $W_i^j$ ,  $\delta_i^j$ ,  $W_o^j$ , and  $\delta_o^j$ .
- 5) The end-to-end timing constraints, namely the value of  $E_{\tau_i \mapsto \pi_i | \tau_j \mapsto \pi_j | \dots | \tau_k \mapsto \pi_k}$  and/or  $F_{\tau_p \mapsto \pi_p | \tau_q \mapsto \pi_q | \dots | \tau_r \mapsto \pi_r}$  where  $i, j, k, p, q, r \in \{1, 2, \dots, n\}$ .

The aim is to find a feasible set of  $\{(C^j, T^j)\}$  pairs for  $j \in \{1, 2, \dots, n\}$  that: 1) meets all the specified end-to-end timing constraints; 2) passes the task schedulability test; and 3) ideally but not necessarily minimizes CPU utilization. A task should not be run faster than necessary, to free resources for additional system objectives.

### B. Solving the Constraints

Our solution is carried out in a three-step process. To make it easier to understand, we use a concrete example with actual numbers to elaborate the process. Consider the pipe topology graph shown in Fig. 7, in which there are six tasks mapped to six pipes. Tasks 1–3 read inputs from sensors, Tasks 4 and 6 write their outputs to actuators, and Task 5 is an intermediary responsible for complicated processing, such as PID control or sensor data fusion. The timing characteristics of the tasks and pipes are shown in Table II. Note that the execution times

are assumed to be identical for all tasks. In practice this would not necessarily be the case but it does not affect the generality of the approach.

In step 1, we use the given  $d_i$ ,  $d_o$ ,  $p$ ,  $W_i$ ,  $\delta_i$ ,  $W_o$ , and  $\delta_o$  to compute the budget of each PT. The budget is set to a value that ensures the three stages (i.e., read, process, and write) finish in one period. To compute  $C^1$ , for example, we aggregate the times for  $\tau_1$  to read, process, and write data. Thus,  $C^1 = (d_i^1/W_i^1) + \delta_i^1 + p^1 + (d_o^1/W_o^1) + \delta_o^1$ . All budgets are computed in a similar way.

When the input to a PT comes from multiple sources the value  $d_i$  is aggregated from all input channels. For example,  $\tau_4$  receives a maximum of  $d_i^4 = d_1 + d_2$  amount of data every transfer from both  $\tau_1$  and  $\tau_2$ . Data from a PT is not necessarily duplicated for all PTs that are consumers. For example,  $\tau_2$  generates a maximum of  $d_o^2 = d_2$  data every transfer, by placing a single copy of the output in a shared memory region accessible to both  $\tau_4$  and  $\tau_5$ . If the communication channels did not involve shared memory, then data would be duplicated, so that  $d_o^2 = 2d_2$ .

In step 2, we derive a list of inequations involving period variables from the given end-to-end timing and scheduling constraints in Table II. For simplicity, the scheduling constraint is shown as a rate-monotonic utilization bound on the six pipe tasks. However, for sensor inputs and some actuator outputs, Quest would map those tasks to I/O VCPUs that have a different utilization bound, as described in our earlier work [7].

The derivation is based on (9) and (14), and the composability property of the pipe model. According to the conditional equations, however, every two pipes with undetermined priority can lead to two possible inequations. This exponentially increases the search space for feasible periods. In order to prune the search space, our strategy is to always start with the case, where  $T^p > T^c$ . This is based on the observation that tasks tend to over-sample inputs for the sake of better overall responsiveness. Thus, the reaction constraint  $E_{\tau_2 \rightarrow \tau_2 | \tau_5 \rightarrow \tau_5 | \tau_6 \rightarrow \tau_6} \leq 25$ , for example, is translated to inequation  $T^5 + C^2 - \Delta + T^6 \leq 25$ . This is derived by combining (9) and (11). It is then possible to translate all timing constraints to inequations with only periods as variables. In addition, periods are implicitly constrained by  $T^j > C^j, \forall j \in \{1, 2, \dots, n\}$ .

Given all the inequations, step 3 attempts to find the maximum value for each period so that the total CPU utilization is minimized. We are then left with a linear programming problem. Unfortunately, there is no polynomial time solution to the integer linear programming problem, as it is known to be NP-hard. Though solutions are possible under certain mathematical conditions [13], this is beyond the scope of this paper. Instead, in practice, the problem can be simplified because: 1) there are usually a small number of fan-in and fan-out pipe ends for each task, meaning that a period variable is usually involved in a small number of inequations and 2) a sensor task period is usually predetermined by a hardware sampling rate limit. For example, if we assume  $T^3$  is known to be 5, a feasible set of periods for the example in Table II is easily found:  $\{T^1 = 10, T^2 = 15, T^3 = 10, T^4 = 10, T^5 = 15, T^6 = 5\}$ . If we ignore the integer requirement, it is possible to find a

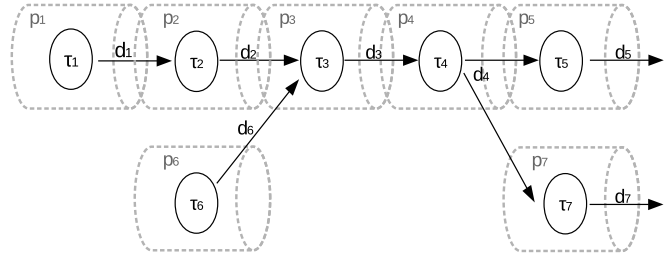


Fig. 8. Simulation pipe topology.

TABLE III  
SIMULATION SETTINGS

$\tau_1$	$\tau_2$	$\tau_3$	$\tau_4$	$\tau_5$	$\tau_6$	$\tau_7$
11.5	5.5	3.5	5.5	11.5	11.5	3.5
PT 1	PT 2	PT 3	PT 4	PT 5	PT 6	PT 7
(12,100)	(6,50)	(4,150)	(6,100)	(12,150)	(12,100)	(4,50)

feasible solution in polynomial time using rational numbers rounded to integers. Though rounding may lead to constraint violations, it is possible to increase the time resolution to ensure system overheads exceed those of rounding errors. If all else fails, one can resort to an exhaustive search of all possible constraint solutions.

Note that the above approach aims to find a feasible, rather than optimal, set of task periods. An optimal solution depends on which constraints (freshness, reaction time, or schedulability) are more important, which is out of the scope of this paper.

## V. EVALUATION

This section describes experiments on the Intel Aero board with an Atom x7-Z8750 1.6 GHz 4-core processor and 4GB RAM.

### A. Simulation Experiments

We developed simulations for both Linux and Quest, to predict the worst-case end-to-end time using the equations in Section III. The simulations consist of seven tasks, all of which search for prime numbers within a certain range and then communicate with one another to exchange their results. Each of the tasks is mapped to a separate pipe. The topology of the pipes is shown in Fig. 8. The communication channel is shared memory with caches disabled and the data size is set to 6.7 KB to achieve a non-negligible 1 millisecond communication overhead. Each task is assigned a different search range and the profiled execution time is shown in Table III in milliseconds. The budget of each pipe is set to be slightly larger than the execution time of its corresponding task, to compensate for system overheads. The settings of each PT are also shown in Table III, again in milliseconds. Apart from the seven main tasks, the system is loaded with low priority background tasks that consume all the remaining CPU resources.

We measure the end-to-end reaction time and freshness time separately, for pipeline  $p_1 \rightarrow p_2 \rightarrow p_3 \rightarrow p_4 \rightarrow p_5$  (denoted by P1), and  $p_6 \rightarrow p_3 \rightarrow p_4 \rightarrow p_7$  (denoted by P2), and compare them to corresponding theoretical bounds. Fig. 9 shows



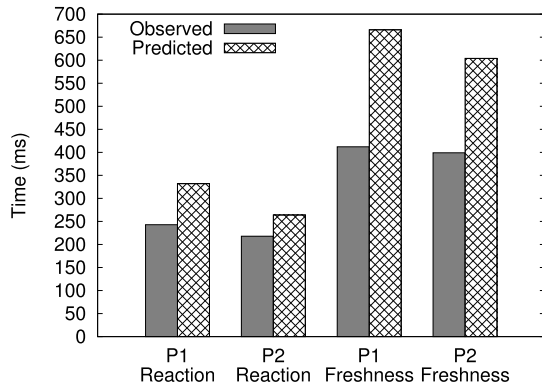


Fig. 9. Observed versus predicted freshness and reaction times.

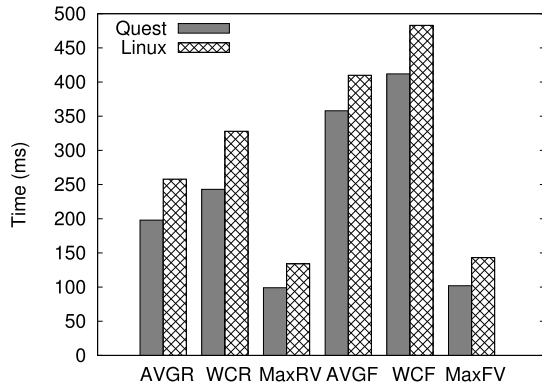


Fig. 10. Quest versus Linux end-to-end times.

the results after 100 000 outputs are produced by  $\tau_5$  and  $\tau_7$ , respectively, on Quest. As can be seen, the observed values are always within the prediction bounds.

The difference between observed and predicted freshness times is greater than between observed and predicted reaction times. This is because our strategy for deriving feasible task periods starts with producers having greater periods than consumers. As stated in Section III-A, the freshness time is affected by the period of each producer, so the prediction may not be as tight as for reaction time.

We also perform the same experiment on Yocto Linux shipped with the Aero board. The kernel is version 4.4.76 and patched with the PREEMPT\_RT patch. While running the simulation, the system also uncompresses Linux source code in the background. This places the same load on the system as the background tasks in Quest. Fig. 10 summarizes the average reaction time (AVGR), worst-case reaction time (WCR), maximum variance of reaction time (MaxRV), average freshness time (AVGF), worst-case freshness time (WCF) and maximum variance of freshness time (MaxFV) of pipeline P1 for Quest and Linux. Compared to Linux, there is less variance shown by the end-to-end times using Quest. Additionally, the freshness and reaction times are lower than with Linux.

### B. Cleanflight Experiment

Our next experiments apply the end-to-end design approach to determining the periods of each task in the reimplementation of Cleanflight. The flight controller is refactored

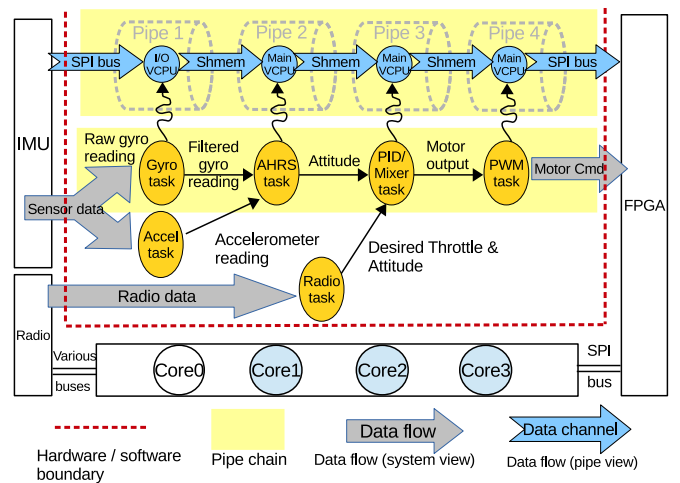


Fig. 11. Cleanflight data flow.

as a multithreaded application running on the Intel Aero board. The hardware and software architecture is shown in Fig. 11.

1) *Hardware*: We currently only use Core 0 to run Cleanflight on Quest. The remaining three cores are reserved for future development of a complete autonomous flight management system. Apart from the CPU, the Aero board also has a field-programmable gate array (FPGA) that emulates I/O interfaces including analog-to-digital conversion, UART serial, and pulse width modulation (PWM). Quest currently uses the I/O hub to send PWM signals to electronic speed controllers (ESCs) that alter motor and, hence, drone rotor speeds. We modified the FPGA logic to improve the timing resolution of PWM signals, as well as control their duty cycle and periods. We also wrote drivers to use an on-board Bosch BMI160 Inertial Measurement Unit (IMU). Both the I/O hub and IMU connect to the main processor via an SPI bus.

2) *Software*: To avoid writing further device drivers, we disabled nontime-critical auxiliary features of Cleanflight, such as telemetry, blackbox data logging, and UART-based flight control configuration. The essential components are shown as circular tasks in Fig. 11. The AHRS sensor fusion task takes the input readings of the accelerometer and gyroscope (in the BMI160 IMU) and calculates the current attitude of the drone. Then, the PID task compares the calculated and target attitudes, and feeds the difference to the PID control logic. In the original Cleanflight code, the target attitude is determined by radio-control signals from a human flying the drone. In an autonomous setting, the target attitude would be calculated according to on-board computations based on mission objectives and flight conditions. The output is nonetheless mixed with the desired throttle and read in by the PWM task, which translates it to motor commands. Motor commands are sent over the SPI bus and delivered as PWM signals to each ESC associated with a separate motor-rotor pair on the drone. We decouple all these tasks into separate threads. For safety reasons, the sensor and PWM tasks are given individual address spaces. For simplicity, we use a synthetic radio input value (20% throttle and 0 degree pitch, roll, and yaw angle) instead

TABLE IV  
TASK EXECUTION TIMES

	Gyro	AHRS	PID	PWM	Accl	Radio
Exec Times ( $\mu$ s)	174	10	2	970	167	12

TABLE V  
TASK PERIODS

Task	Gyro	AHRS	PID
Budget/Period ( $\mu$ s)	200/1000	100/5000	100/2000
Task	PWM	Accl	Radio
Budget/Period ( $\mu$ s)	1000/5000	200/1000	100/10000

of reading from the real radio driver. The tasks are profiled and execution times are shown in Table IV.

As can be seen in Fig. 11, there are three data paths, originating from the gyro, accelerometer, and radio receiver, respectively. Unfortunately, there is little information available on what end-to-end timing constraints should be imposed on each path to guarantee a working drone. Most timing parameters in the original Cleanflight are determined by trial and error. Instead of determining the optimum timing constraints, the focus of this paper is on guaranteeing given constraints. Therefore, we first port Cleanflight to Yocto Linux on the Aero board, as a reference implementation. The Linux version remains single-threaded and is used to estimate the desired end-to-end time. For example, for the gyro path, the WCR and freshness times are measured to be 9769 and 22972  $\mu$ s, respectively. We round them to 10 and 23 ms, and use them as end-to-end timing constraints for our flight controller implementation. Using the same approach, we determine end-to-end reaction and freshness times for the accelerometer path, which are set to 10 and 23 ms, respectively. Finally, for the radio path, we set the end-to-end reaction and freshness times to be 20 and 44 ms, respectively.

Using the execution times in Table IV and timing constraints above, we apply the end-to-end design approach to derive the periods. The results for each task are shown in Table V. Note that there are four extra tasks inherited from the original Cleanflight. They are responsible for checking system utilization, battery current and voltage, and emitting low battery alerts. However, they are not on the critical control path and we run them only when there is surplus CPU time. Thus for brevity, they are not shown in Fig. 11 and not involved in the period calculation.

3) *Evaluation*: To measure the actual end-to-end time, we focus on the longest pipe chain highlighted in Fig. 11. We instrument the Cleanflight code to append every gyro reading with an incrementing ID, and also record a timestamp before the gyro input is read. The timestamp is then stored in an array indexed by the ID. Every task is further instrumented to maintain the ID when translating input data to output. This way, the ID is preserved along the pipe chain, from the input gyro reading to the output motor command. After the PWM task sends out motor commands, it looks up the timestamp using its ID and compares it to the current time. By doing this, we are able to log both the reaction and freshness end-to-end time for every input gyro reading. We then compare the observed end-to-end time with the given timing constraints, as

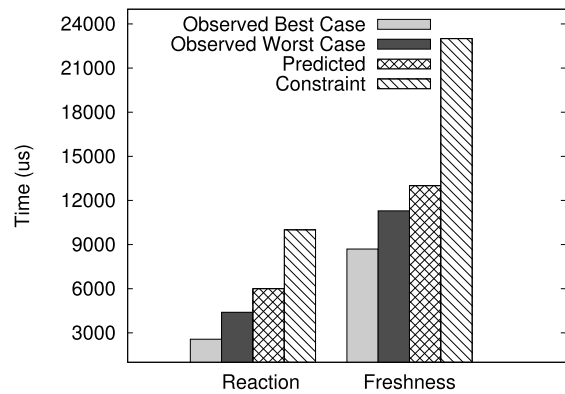


Fig. 12. Cleanflight times.

well as the predicted worst-case value. Results are shown in Fig. 12. As can be seen, the observed values are always within the predicted bounds, and always meet the timing constraints.

## VI. RELATED WORK

Feiertag *et al.* [11] distinguished four semantics of end-to-end time and provide a generic framework to determine all the valid data paths for each semantic. The authors do not perform timing analysis as no scheduling model is assumed. Hamann *et al.* [10] also discussed end-to-end reaction and age time. Their work focuses on integrating three different communication models, including the implicit communication model, into existing timing analysis tools, such as SymTA/S [14]. While our composable pipe model is also based on implicit communication, we perform timing analysis for a sequence of RMS-scheduled tasks running on bandwidth-preserving servers.

Others have proposed end-to-end timing analysis at the model level in the automotive domain [15], [16]. Our approach is applicable to any applications with data pipeline processing, freshness, and reaction constraints. A large portion of end-to-end reaction time analysis is based on the synchronous data-flow graph (SDFG) [17], where intertask communication is driven by the arrival of input data. Real-time scheduling techniques have been used to analyze end-to-end latencies in systems modeled by SDFGs [18], [19].

Gerber *et al.* [12] proposed a synthesis approach that determines tasks' periods, offsets, and deadlines from end-to-end timing constraints. Their work relies on task precedence constraints as there is no scheduling model used for the analysis. This paper uses the Quest scheduling model to perform end-to-end timing analysis. We then derive task periods and budgets to ensure specific reaction, freshness, and schedulability constraints.

In contrast to this paper, there are programming languages that allow the specification and verification of end-to-end timing properties. For example, Prelude [20] and Giotto [21] are languages designed to derive tasks' periods based on user-specified timing constraints. Lauer *et al.* [22] and Boniol *et al.* [23] used formal methods to verify end-to-end timing properties for avionic systems. Forget *et al.* [24]

defined a language to specify formally end-to-end constraints and propose a technique to verify those constraints.

## VII. CONCLUSION

This paper introduces a composable pipe model that is built on task, scheduling, and communication abstractions. The pipe model is used to derive a set of task periods and budgets that satisfy both schedulability and end-to-end *freshness* and *reaction* timing constraints. We analyze reaction and freshness time in the context of data flow through a task pipeline in a drone flight controller, which performs sensor data processing and motor actuation. Experiments show that Cleanflight ported to the Quest RTOS is able to achieve end-to-end latencies within the predicted time bounds derived by analysis. The scheduling framework in Quest is rich enough to provide bandwidth-guaranteed service to both tasks and ISRs via VCPUs. However, the analysis presented herein is applicable to any OS capable of providing CPU bandwidth guarantees to all task and I/O operations.

Future work will focus on the application of the pipe model to a fully autonomous flight management system involving additional tasks to those in Cleanflight. We intend to use our in-house partitioning hypervisor, called Quest-V [25] to simultaneously host Quest and Linux on the Aero board. Less time-critical tasks, such as telemetry, blackbox data logging, path planning, and camera-based object detection will be assigned to Linux, to leverage pre-existing device drivers and libraries. Time-critical flight control tasks will then be able to execute without interference in Quest. This avoids the need for a separate control board dedicated to time-critical tasks, which requires space on the drone, and increases weight and power consumption.

## REFERENCES

- [1] *Cleanflight*. [Online]. Available: <http://cleanflight.com/>
- [2] *Quest RTOS*. [Online]. Available: <http://questos.org/>
- [3] B. Sprunt, "Scheduling sporadic and aperiodic events in a hard real-time system," *Softw. Eng. Inst.*, Carnegie Mellon, Pittsburgh, PA, USA, Rep. CMU/SEI-89-TR-11, 1989.
- [4] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: Operating system support for multimedia applications," in *Proc. IEEE Int. Conf. Multimedia Comput. Syst.*, Boston, MA, USA, May 1994, pp. 90–99. [Online]. Available: <https://ieeexplore.ieee.org/document/292439/>
- [5] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," in *Proc. IEEE Real Time Syst. Symp. (RTSS)*, Santa Monica, CA, USA, 1989, pp. 166–171.
- [6] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.
- [7] M. Danish, Y. Li, and R. West, "Virtual-CPU scheduling in the Quest operating system," in *Proc. 17th Real Time Embedded Technol. Appl. Symp.*, Chicago, IL, USA, 2011, pp. 169–179.
- [8] E. Missimer, K. Missimer, and R. West, "Mixed-criticality scheduling with I/O," in *Proc. 28th Euromicro Conf. Real Time Syst. (ECRTS)*, Jul. 2016, pp. 120–130.
- [9] H. R. Simpson, "Four-slot fully asynchronous communication mechanism," *IEE Proc. E Comput. Digit. Techn.*, vol. 137, no. 1, pp. 17–30, Jan. 1990.
- [10] A. Hamann, D. Dasari, S. Kramer, M. Pressler, and F. Wurst, "Communication centric design in complex automotive embedded systems," in *Proc. 29th Euromicro Conf. Real Time Syst.*, Dubrovnik, Croatia, 2017, pp. 1–20.
- [11] N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson, "A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics," in *Proc. IEEE RTSS Workshop Compositional Theory Technol. Real Time Embedded Syst.*, Nov. 2008, pp. 17–20.
- [12] R. Gerber, S. Hong, and M. Saksena, "Guaranteeing real-time requirements with resource-based calibration of periodic processes," *IEEE Trans. Softw. Eng.*, vol. 21, no. 7, pp. 579–592, Jul. 1995.
- [13] S. Bradley, A. C. Hax, and T. L. Magnanti, *Applied Mathematical Programming*. Reading, MA, USA: Addison-Wesley, 1977.
- [14] R. Henia *et al.*, "System level performance analysis—The SymTA/S approach," in *Proc. IEEE Comput. Digit. Techn.*, 2005, pp. 29–74.
- [15] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "Synthesizing job-level dependencies for automotive multi-rate effect chains," in *Proc. 22nd IEEE Int. Conf. Embedded Real Time Comput. Syst. Appl.*, Aug. 2016, pp. 159–169.
- [16] S. Mubeen *et al.*, "End-to-end timing analysis of black-box models in legacy vehicular distributed embedded systems," in *Proc. 21st IEEE Int. Conf. Embedded Real Time Comput. Syst. Appl.*, Aug. 2015, pp. 149–158.
- [17] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987.
- [18] A. Singh, P. Ekberg, and S. Baruah, "Applying real-time scheduling theory to the synchronous data flow model of computation," in *Proc. 29th Euromicro Conf. Real Time Syst.*, 2017, pp. 1–22.
- [19] J. Khatib, A. Munier-Kordon, E. C. Kliikpo, and K. Trabelsi-Colibet, "Computing latency of a real-time system modeled by synchronous dataflow graph," in *Proc. 24th Int. Conf. Real Time Netw. Syst. (RTNS)*, 2016, pp. 87–96.
- [20] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens, "Multi-task implementation of multi-periodic synchronous programs," *Discr. Event Dyn. Syst.*, vol. 21, no. 3, pp. 307–338, Sep. 2011.
- [21] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," in *Embedded Software. EMSOFT 2001 (LNCS 2211)*. Berlin, Germany: Springer, 2001. [Online]. Available: [https://link.springer.com/chapter/10.1007/3-540-45449-7\\_12#citeas](https://link.springer.com/chapter/10.1007/3-540-45449-7_12#citeas)
- [22] M. Lauer, F. Boniol, C. Pagetti, and J. Ermont, "End-to-end latency and temporal consistency analysis in networked real-time systems," *Int. J. Crit. Comput. Based Syst.*, vol. 5, nos. 3–4, pp. 172–196, 2014.
- [23] F. Boniol, M. Lauer, C. Pagetti, and J. Ermont, "Freshness and reactivity analysis in globally asynchronous locally time-triggered systems," in *NASA Formal Methods*. Berlin, Germany: Springer, 2013.
- [24] J. Forget, F. Boniol, and C. Pagetti, "Verifying end-to-end real-time constraints on multi-periodic models," in *Proc. 22nd IEEE Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, Sep. 2017, pp. 1–8.
- [25] R. West, Y. Li, E. Missimer, and M. Danish, "A virtualized separation kernel for mixed-criticality systems," *ACM Trans. Comput. Syst.*, vol. 34, no. 3, p. 8, Jun. 2016.



**Zhuoqun Cheng** received the B.Eng. degree in computer engineering from Zhejiang University, Hangzhou, China, in 2012. He is currently pursuing the Ph.D. degree in computer science with Boston University, Boston, MA, USA, under the supervision of Prof. R. West.

His current research interests cover various aspects of system software, cyber-physical systems, RTOSs and hypervisors.



**Richard West** (SM'14) received the Master of Engineering (M.Eng.) degree (First Class Hons.) in microelectronics and software engineering from the University of Newcastle upon Tyne, Newcastle upon Tyne, U.K., in 1991 and the Master of Science (M.S.) and Ph.D. degrees in computer science from the Georgia Institute of Technology, Atlanta, GA, USA, in 1998 and 2000, respectively.

He is a Professor with the Computer Science Department, Boston University, Boston, MA, USA.

His research focuses on the development of real-time and embedded operating systems, addressing issues concerning safety, and predictability. He has studied real-time scheduling and resource management, cache-aware performance of multicore processors, and machine virtualization, amongst other topics. He is currently leading the development of the Quest RTOS for multicore processors. Its sister system, Quest-V is a secure and predictable separation kernel that forms a distributed system on a chip, providing efficient, predictable and safe execution of sandboxed guest systems including Linux.



**Craig Einstein** received the B.A. degree in geophysics and planetary sciences from Boston University, Boston, MA, USA, in 2016, where he is currently pursuing the Ph.D. degree in computer science, under the supervision of Prof. R. West.

He researches computer systems with an emphasis on real-time systems and autonomous control. Upon the completion of his doctorate, he would like to research in the space industry developing systems to make space travel more efficient and accessible.