Preemption-resistant control on a non-real-time operating system

Justin P. Pearson, David A. Copp, João P. Hespanha

***Abstract*— We propose an architecture for implementing discrete-time control algorithms on a non-real-time operating system so that sensing and actuation occur at precise times, even if the OS preempts the control task. The architecture has the controller perform its computation on one processor and its sensing/actuation on a separate microcontroller. The controller sends arrays of future time-stamped actuator commands to the microcontroller, thereby allowing actuation to continue if preemption occurs. We use this architecture to drive a DC motor with a Beaglebone Black single-board computer, with the controller running on Linux and the I/O performed on a subsidiary microcontroller within the Beaglebone. This scheme achieves a timing accuracy of 40 μs. We demonstrate that this configuration improves a PID controller's performance in the presence of OS preemption, even when the preemption persists across several sampling periods. In effect, this provides a mechanism to make a controller resistant to OS preemption.**

I. Introduction

A. Motivation

Modern computing systems offer many performance-enhancing features like multi-tasking operating systems, multiple layers of caching, and multiprocessor support. However, these features often result in nondeterminstic runtime behavior, making it a challenge to implement control systems on such platforms (see Figure 1). Consequently, controllers are often implemented on specialized platforms like "bare metal" microcontrollers, real-time OSs, or FPGAs, which offer finer control of execution and timing. It would be advantageous for a controller architecture to offer both the flexibility of a general-purpose computing platform and also the determinism of a specialized solution. This paper proposes a controller architecture that addresses this need.

B. Results

The specific contribution of this paper is a controller architecture that enables a controller to run on a non-real-time OS like Linux, yet maintain precise timing of the sensing and actuation despite OS preemption. This is achieved by performing the sensing and actuation on a dedicated "bare metal" microcontroller that in essence serves as a real-time I/O coprocessor; we refer to this as the "Real-Time Unit" (RTU). Since the OS may preempt the controller at any time, we cannot rely on it for precise sampling or actuation. On the other hand, the RTU does not run an OS, so it can sample and actuate at precise times without danger of OS preemption. The key idea is to have the RTU buffer time-stamped sensor measurements from the plant and apply buffered time-stamped actuation commands to the plant at precise times. Asynchronously, the controller requests an array of past measurements from the RTU, computes an array of future time-stamped actuation commands, and sends it to the RTU to be executed at the correct times. Consequently, the controller can be preempted but the RTU will continue to apply actuation on its behalf.

Center for Control, Dynamical Systems, and Computation, University of California, Santa Barbara, Santa Barbara, CA 93106, USA jppearson@ece.ucsb.edu, dacopp@engr.ucsb.edu, hespanha@ece.ucsb.edu

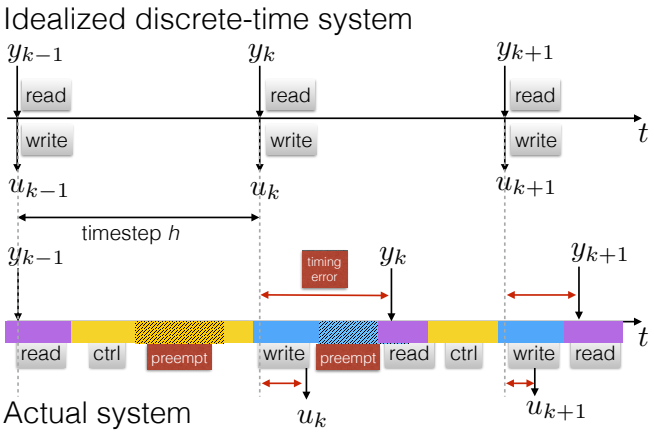

Fig. 1. Timing in an idealized discrete-time system (top) versus a physical control system running on a non-real-time operating system (bottom). Due to OS preemption and other sources of nondeterminacy, the sensor and actuator signals y_k and u_k neither occur at their intended sample times nor align with each other.

We implemented this controller architecture on a Beaglebone Black (BBB) to drive a DC motor. For demonstration purposes, a simple PID controller runs on Linux on the BBB's main 1-GHz CPU, and we use a subsidiary processor core on the BBB as the RTU. The RTU reads a rotary encoder and actuates the motor with PWM every 5 ms with 40 μs accuracy. The PID controller uses a simple method of predicting future measurements to compute an array of future PWM values, which it sends to the RTU. We compare this setup to a PID controller with identical gains that uses the BBB's standard file-based interface for I/O. We observe that although both setups perform well when the CPU is idle, when run alongside several other high-priority tasks the RTU-based setup far out-performs the standard I/O mechanism.

C. Backgroud and related work

A program may execute nondeterministically for several reasons. On a multithreaded processor, the task scheduler may interrupt a task to let another task use the CPU, or to service an interrupt [10]. The time required to fetch data from memory varies wildly depending on whether the data was

Preprint submitted to 56th IEEE Conference on Decision and Control.
Received March 20, 2017.

cached [13, 14]. In a Non-Uniform Memory Access (NUMA) multiprocessor architecture, CPUs are grouped into nodes, each with its own dedicated local memory. The speed of a memory reference therefore depends on whether the desired data resides in the executing processor's local memory or in another node's memory [6, 15]. System Management Interrupts commandeer the CPU and RAM to perform system maintenance, e.g., turning on the fan or verifying memory consistency [12].

Several platforms aim to provide determinism in program execution. FPGA designs are synthesized to obey strict user-supplied timing constraints and are therefore well-suited for control applications, see [9]. Matlab's "Simulink Coder" converts Simulink diagrams into code for embedded targets without OSs; [5] studies its use in rapid prototyping of real-time control algorithms. Real-time OSs like VxWorks, Integrity RTOS, μCOS, and FreeRTOS all provide OS-related functionality like multi-tasking, networking, file system support, and memory management while providing timing guarantees. The current RTOSs are surveyed in [2]. RTOS applications to control are explored in [1]. The authors of [3] analyze RTOS task scheduler algorithms. While powerful, these platforms lack the flexibility of a true general-purpose OS in terms of availability of libraries and device drivers.

Work has also been done to modify Linux itself to provide real-time performance guarantees. The OS provided by the "Real-Time Linux" project allows interrupt service routines (ISRs) to be run as regular tasks. Similarly, the Xenomai software augments Linux with a second kernel that runs above the main Linux kernel. The Xenomai kernel can disable the Linux scheduler in order to guarantee timely task execution. RT Linux and Xenomai have found applications in low-latency audio processing [7] and electrical substation automation [11]. The both allow the user to prioritize userspace tasks above interrupts. However, this practice can result in reduced performance if misconfigured, e.g., dropping network packets due to the network ISR being neglected in favor of the control task. These solutions do not overcome the fundamental problem that when several real-time tasks share the same CPU, preemption is inevitable and is either managed by the OS or by the programmer.

This paper implements the proposed control architecture on a Beaglebone single-board computer, running the controller on the primary CPU and utilizing a subsidiary processor core as a real-time I/O coprocessor. The practice of placing application-specific coprocessors beside a general-purpose procesor is a feature of the "ARM big.LITTLE" heterogeneous computing architecture and the family of processors in Texas Instruments' "Open Multimedia Applications Platform" initiative, discussed in [4].

This paper is laid out as follows. In Section II we describe a general architecture for running a control task on a non-real-time OS with a real-time I/O coprocessor. In Section III we apply this technique to a PID controller on a Beaglebone single-board computer and observe its resiliency against OS preemptions.

II. Real-time I/O coprocessor concept

In this section we describe the architecture of our control and sensing/actuation scheme. Figure 2 illustrates the basic idea: The controller runs on a non-real-time OS like Linux, whereas the sensing and actuation are performed by a dedicated "bare-metal" microcontroller called the Real-Time Unit (RTU). The RTU contains two circular buffers, one of size n_s for time-stamped sensor measurements, and one of size n_a for time-stamped actuator commands. At each timestep, the RTU reads, time-stamps, and saves a new sensor measurement in the sensor buffer, and then applies the appropriate actuator command from the actuator buffer. The controller and RTU may be co-located on the same circuit-board or even within a single system-on-a-chip.

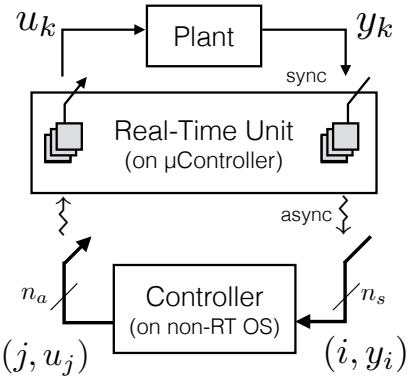

Fig. 2. Schematic of the control architecture. The real-time I/O coprocessor measures sensors y_k and applies actuator values u_k every T_s time units from its two buffers. Asynchronously, the controller retrieves the n_s most recent sensor values and transmits n_a time-stamped actuator values for the RTU to apply to the plant.

We now explain Algorithms 1 and 2 below, which summarize the code that runs on the controller and the RTU.

Algorithm 1: Controller

while true **do**
  Retrieve the RTU's sample buffer.
  Generate a list of n_a time-stamped actuator values.
  Send the list to the RTU.
  Wait for next timestep.
end while

Algorithm 2: RTU

while true **do**
  if controller requested data, **then**
    Send the sensor buffer to the controller.
  end if
  if controller sent a new actuation schedule, **then**
    Copy the schedule to the actuation buffer.
  end if
  Check the time.
  if sample time $T_s k$, $k \in \mathbb{N}$ just elapsed, **then**
    Sample sensors and store with time-stamp k.
    if (k, u_k) is in the actuation buffer, **then**
      Apply input u_k to the plant.
    else

Preprint submitted to 56th IEEE Conference on Decision and Control.
Received March 20, 2017.

20: Apply default input to the plant.
21: **end if**
22: **end if**
23: **end while**

(Algorithm 1: Controller.) At an arbitrary time t, the controller requests the RTU's measurement buffer and receives n_s time-stamped measurements $(k - n_s + 1, y_{k-n_s+1}), \ldots, (k, y_k)$, where $k \coloneqq \lfloor t/T_s \rfloor$ is the index of the last timestep before time t. The controller then computes a list of n_a time-stamped actuator values. The resulting actuation sequence could follow the retrieved sample sequence by starting at sample $k+1$, e.g., $(k+1, u_{k+1}), \ldots, (k+n_a, u_{k+n_a})$. However, if it is known that the controller will take at least C sample times to run, the controller may instead compute and send actuation signals to be applied at sample times $k+C+1, \ldots, k+C+n_a$. In either case, the controller then transmits the actuation sequence to the RTU's actuator buffer. Note that because the controller may be preempted, the controller's actions occur asynchronously with respect to the RTU's sampling and actuation times. Section II-A discusses the issues of generating future actuation sequences.

(Algorithm 2: RTU.) The RTU loop starts by checking whether the controller requested data or delivered a new actuation schedule. If so, the RTU transfers sensor data to the controller or copies new actuation commands into the RTU's private buffer. At sample time $t = T_s k$, $k = 1, 2, \ldots$, the RTU reads the sensor measurement y_k and stores the time-stamped measurement (k, y_k) in its circular buffer. It then searches its actuation buffer for an actuation command of the form (k, u_k) and applies u_k to the plant over the time interval $[T_s k, T_s(k+1))$. If the actuation buffer does not contain a command for timestep k, the RTU applies some default actuation, e.g., u_{k-1} or 0.

The RTU's ability to sample at precise times depends crucially on its ability to check the time rapidly. Consequently, it is important that the RTU be able to execute Algorithm 2 lines 8–13 quickly. Therefore the interconnection between the controller and the RTU needs to be fast, e.g., a shared memory. Similarly, the actual sampling and actuation also needs to happen quickly (lines 15–21).

Figure 3 illustrates this architecture with buffer sizes $n_s = 3$ and $n_a = 5$. At each sample time $t = T_s k$, $k \in \mathbb{N}$, the RTU reads and stores a sensor measurement. At some time between T_s and $2T_s$, the controller delivers an actuation schedule (k, u_k), $k = 2, \ldots, 6$, then gets preempted. Despite the controller being preempted, the RTU executes the actuation schedule. Some time between $3T_s$ and $4T_s$ the controller requests the sensor buffer, which contains (k, y_k), $k = 1, 2, 3$. The controller then begins computing the actuation sequence (k, u_k), $k = 4, \ldots, 8$, but gets preempted partway through. Later, between $5T_s$ and $6T_s$, the controller awakens and delivers the actuation sequence. Note that because of preemption, the new actuation schedule arrives too late to apply the new (underlined) actuator values intended for sample times $4T_s$ and $5T_s$; instead, the RTU applied actuator commands u_4 and u_5 from the previous actuation schedule. After time $7T_s$ the controller receives the sample buffer with measurements for $k = 5, 6, 7$. Note that sample y_4 was overwritten and so is not available to the controller.

The key idea in this architecture is that the closed-loop system can tolerate some amount of OS preemption because the RTU continues to gather measurements and apply actuation even while the controller is asleep. The aim is for the controller to provide a sufficient number of future actuator values so the RTU can continue to stabilize the plant if the controller gets preempted. Even though the future actuations are applied "open-loop", we shall see that they are better than holding the actuators constant until the controller awakens.

A. Building the actuation schedule

The architecture proposed here requires the controller to produce, at each sample time k, an actuation schedule with control values for the next n_a future sample times $k+1, k+2, \ldots, k+n_a$. Two options are available: a model-free approach that generates the future control signals without an explicit model for the process and a model-based approach that uses such a model.

To describe both approaches consider a discrete-time nonlinear controller expressed by the following state-space model

$$z_{k+1} = f(z_k, y_k, r_k), \qquad u_k = g(z_k, r_k), \tag{1}$$

where the y_k denote sensor measurements, the u_k actuation values, and the r_k reference signals.

The model-free approach generates the n_a future actuator commands $u_{k+1}, u_{k+2}, \ldots, u_{k+n_a}$ using polynomial extrapolation. Assuming that the measurement sequence can be approximated by a polynomial of degree q, one can use the previous $q+1$ measurements $y_{k-q}, \ldots, y_{k-1}, y_k$ to predict $n_a - 1$ future measurements $y_{k+1}, y_{k+2}, \ldots, y_{k+n_a-1}$. Feeding these to the controller (1), one obtains the desired future actuator commands $u_{k+1}, u_{k+2}, \ldots, u_{k+n_a}$. As we shall see in Section III, even a low order polynomial (linear extrapolation with $q = 1$) can be used to obtain good results.

When a plant model is available, the accuracy of the predicted measurements can be improved. Assuming a linear plant model of the form

$$x_{k+1} = Ax_k + Bu_k, \qquad y_k = Cx_k + Du_k, \tag{2}$$

if the plant's state x_k can be directly measured or estimated, one can estimate future measurements by directly solving the process model (2), which leads to

$$\hat{y}_{k+i} = CA^i \hat{x}_k + \Big(\sum_{j=0}^{i-1} CA^{i-j-1} B u_{k+j} \Big) + Du_{k+i}, \quad \forall i \in \{1, 2, \ldots, n_a - 1\}, \tag{3}$$

where $\hat{x}_k$ denotes the state estimate at time k and $u_{k+1}, u_{k+2}, \ldots, u_{k+i}$ a sequence of future control signals constructed based on the controller model (1) and previous measurement estimates obtained by (3). The use of the plant model (2) permits a more accurate estimate of future measurements and consequently a better schedule for the

Preprint submitted to 56th IEEE Conference on Decision and Control.
Received March 20, 2017.

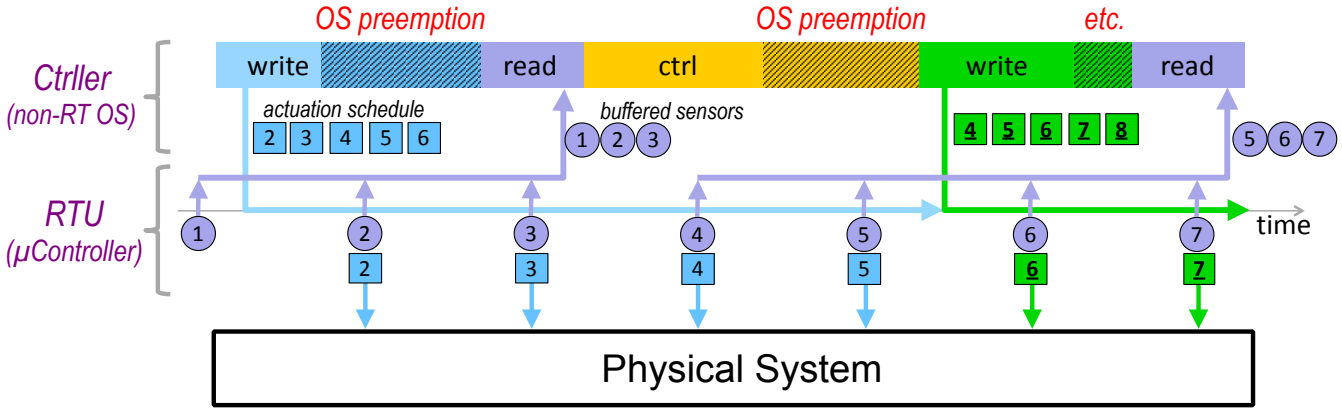

Fig. 3. The RTU buffers sensor measurements (circles) and executes buffers of time-stamped actuator commands (squares) from the controller.

future controls. However, our initial experiments indicate that this approach does not yield significant gains unless the sample time is fairly large.

Linearity of the process model in (2) was assumed solely for simplicity of presentation, as the sequence of estimated outputs can easily be generated for a nonlinear process model, provided that the process' state can be measured or estimated. Model predictive control (for either linear or nonlinear plants) is especially attractive for this type of architecture, as it automatically produces a sequence of future controls.

III. Experimental results

In this section we compare the performance of a PID controller driving a DC motor when it uses a standard file-based I/O interface versus using a real-time I/O coprocessor.

A. Hardware

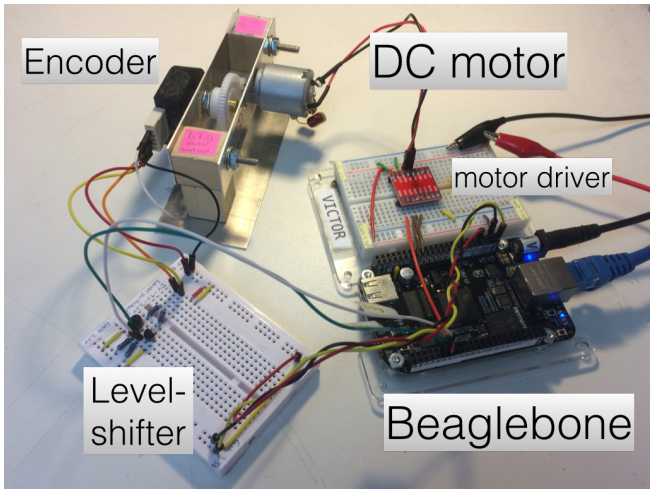

Fig. 4. Picture of the hardware setup. A Beaglebone Black drives a DC motor and measures its shaft angle using a rotary encoder.

Figure 4 shows our hardware setup. A TB6612FNG motor driver drives a hobby-grade permanent-magnet DC motor from a 5-volt power supply. The motor driver takes a 50 kHz PWM signal and three discrete 3.3 V signals which determine the motor direction. The motor shaft angle is measured by a US Digital rotary optical encoder. The encoder has 4096 counts per rotation and outputs a quadrature-encoded pulse (QEP) signal. A 5V-to-3.3V level-shifting circuit scales the QEP signal.

The controller runs on a Beaglebone Black (BBB) single-board computer [8]. The BBB has a 1-GHz processor, 512 MB RAM, HDMI video, an ethernet port, and a USB port. It ships with Debian Linux installed on its 4 GB flash memory. It is powered by a Texas Instruments Sitara AM3358BZCZ100 processor, which contains ADC, PWM, QEP, and GPIO peripherals.

For a real-time I/O coprocessor, we use one of the two "Programmable Real-Time Units" (PRUs) included in the Sitara microcontroller. Designed for real-time applications, each PRU is a 32-bit 200-MHz RISC processor core that executes independently from the main CPU, has its own 8 kB data RAM, and has full access to the peripherals on the Sitara. The CPU and PRU have access to each other's memory and can therefore exchange time-stamped sensor and actuator data quickly. The PRU is not pipelined, making its execution simpler and more deterministic: register-level instructions run in 1 cycle (5 ns), and memory instructions to the PRU's local memory take 3 cycles. The PRU's data RAM does not share a bus with the main RAM, so the PRU can read and write to it without the risk of bus contention with the main memory. A cycle counter register within the PRU allows it to track time in increments of 5 ns. For these reasons, the PRU is well-suited for use as a RTU.

We implemented the control architecture described in Section II on the PRU with circular buffer sizes of $n_a = n_s = 32$. The sensor buffer stored time-stamped QEP samples from the rotary encoder, whereas the actuator buffer stored time-stamped PWM and GPIO commands for the motor driver. To coordinate data transfer between the CPU and the PRU, we double-buffered the sensor and actuator arrays in the PRU data RAM and implemented rudimentary mutual-exclusion semaphores. Accounting for the time to sample, actuate, and communicate, our implementation on the PRU achieved

Preprint submitted to 56th IEEE Conference on Decision and Control.
Received March 20, 2017.

sampling and actuation timing accuracy of 40 μs.

B. Controller Design

A DC motor can be modeled as a series connection of a resistor, inductor, and back-EMF voltage source, resulting in the dynamic equations

$$\begin{aligned} V_m &= iR + L\dot{i} + K_1\omega \\ J\dot{\omega} &= -b\omega + K_2 i + \tau_{\text{ext}} \\ \dot{\theta} &= \omega, \end{aligned} \tag{4}$$

where i is the current through the motor, R and L are the resistance and inductance of the motor windings, θ is the motor shaft angle, ω is the angular velocity, V_m is the voltage applied across the motor, K_1 and K_2 are motor constants, b is the friction coefficient, J is the angular moment of inertia of the motor, and τ_{ext} is any external torque imposed on the motor shaft.

Equations (4) form a 3rd-order linear dynamical system. System identification was performed on the motor system using ARX on voltage and angle data to obtain the following 3rd-order discrete-time linear model of the DC motor:

$$\frac{\Theta(z)}{V(z)} = \frac{-0.5898z^{-1} - 1.121z^{-2} - 0.2757z^{-3}}{1 - 1.586z^{-1} + 0.3719z^{-2} + 0.2136z^{-3}}. \tag{5}$$

The model's sample time was $T_s \coloneqq 0.005$ s. The model was validated with additional input/output data.

A PID controller was designed using Matlab's PIDTuner with the discrete-time transfer function

$$C(z) = k_p + k_i \frac{T_s}{z-1} + k_d \frac{1}{T_f + T_s/(z-1)}, \tag{6}$$

where $k_p = -0.0304$, $k_i = -0.106$, $k_d = -8.73\text{e}{-4}$, and $T_f = 0.00405$. The parameter T_f is the time-constant of a first-order filter on the derivative term. This controller was implemented as an IIR filter in C. To produce a future actuation schedule, the two most recent angle measurements were used to linearly extrapolate measurements for the future motor angles, and the PID controller was run on the tracking error between those predicted measurements and a known reference signal. Specifically, given the sensor buffer ending with sample k, the controller computed

$$\begin{aligned} \Delta &\coloneqq \theta_k - \theta_{k-1} \\ \theta_{k+i} &\coloneqq \theta_k + \Delta i \\ v_{k+i} &\coloneqq c(\theta_{k+i}, \theta_{k+i-1}, \theta_{k+i-2}, v_{k+i-1}, v_{k+i-2}), \end{aligned}$$

for $i = 1, \ldots, n_a$, where $c(\cdot)$ is the IIR representation of the controller (6), and θ_k and v_k are the angle measurement and voltage command at the kth sample time.

C. Results

Figure 5 shows the result of the two PID controllers as they track a triangle-wave reference signal on motor shaft angle. The top two plots show the performance of the PID controller when it uses the standard I/O mechanism on the BBB, wherein the peripherals appear as normal files. The lower two plots show the response of the PID controller when it uses the PRU as a real-time I/O coprocessor. The second and fourth plots show the time between each iteration of the PID control loop running on the main processor.

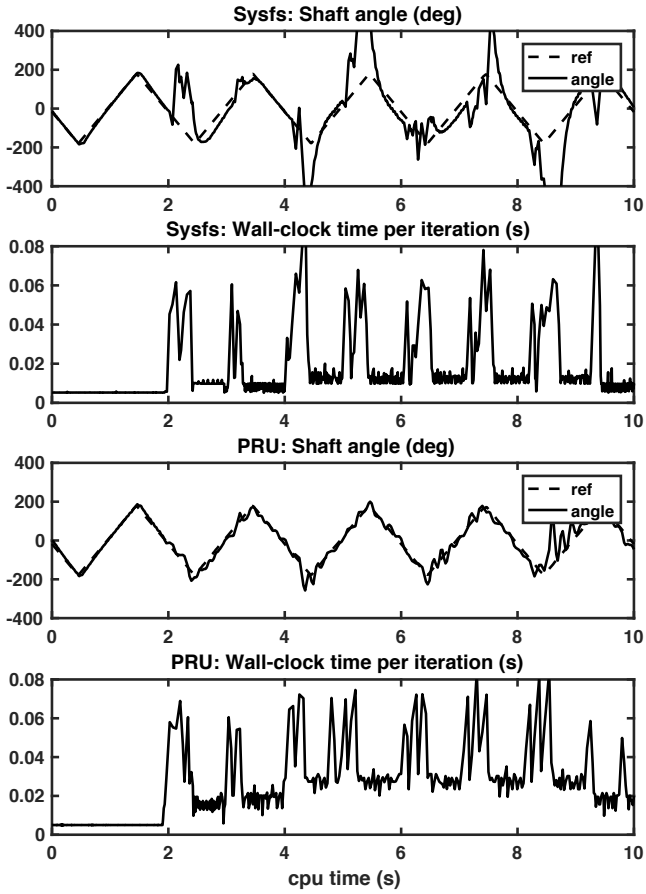

Fig. 5. Both the standard PID controller and the PRU-based PID controller have similar performance under idle ($t < 2$). However, when subjected to OS preemption ($t > 2$), the PRU out-performs the standard one.

TABLE I

RMS REFERENCE-TRACKING ERROR OF THE CONTROLLERS UNDER IDLE AND HEAVY SYSTEM LOAD.

	PID using standard I/O	PID using RTU I/O
Idle	12.0	11.3
Heavy	144	36.3

For the first two seconds, the PID controllers each run with essentially sole control of the CPU. There are no major OS preemptions during $t < 2$ and the second and fourth plots show that each iteration takes the intended sample time $T_s = 0.005$ s. We observe that the two control configurations have similar performance during $t < 2$. At $t = 2$, several higher-priority CPU-heavy tasks were spawned. The spikes in the second and fourth plots during $t > 2$ correspond to controller preemptions, sometimes lasting 10 times the sample period. Whereas the PID controller using the standard I/O interface is heavily disrupted by these preemptions, we observe that the RTU-based controller runs much more smoothly due to the PRU buffering future control signals. The root-mean-square

Preprint submitted to 56th IEEE Conference on Decision and Control.
Received March 20, 2017.

tracking errors for each controller under idle and heavily-loaded processor conditions are shown in Table I. Similar results were obtained as the priorities of the competing tasks were changed to vary the frequency and durations of the OS preemptions.

IV. Conclusion

In this paper we presented a controls architecture that pairs a real-time I/O coprocessor with a controller on a non-real-time operating system. The RTU enables sampling and actuation at precise times, even when the controller is preempted by the OS. This enables control designers to reap the benefits of an OS with minimal concern for the timing uncertainties associated with the OS task scheduler. We demonstrated the platform's utility by designing a preemption-resistant PID controller on a Beaglebone Black that uses its Programmable Real-time Unit as a real-time I/O coprocessor. The RTU-based PID controller out-performed the standard PID controller in the presence of large OS preemptions. Future directions of this work include using a more sophisticated method of forward-prediction for the actuation sequence, such as model predictive control. Also we intend to extend this architecture to multiple controllers distributed across a network.

References

[1] R Alur, K-E Arzen, John Baillieul, TA Henzinger, Dimitrios Hristu-Varsakelis, and William S Levine. *Handbook of networked and embedded control systems*. Springer Science & Business Media, 2007.

[2] Sanjeev Baskiyar and Natarajan Meghanathan. A survey of contemporary real-time operating systems. *Informatica (Slovenia)*, 29(2):233–240, 2005.

[3] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Real-Time Systems Series. Springer US, 2011.

[4] J. Chaoui, K. Cyr, S. de Gregorio, J. P. Giacalone, J. Webb, and Y. Masse. Open multimedia application platform: enabling multimedia applications in third generation wireless terminals through a combined risc/dsp architecture. In *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.01CH37221)*, volume 2, pages 1009–1012 vol.2, 2001.

[5] R. Grepl. Real-time control prototyping in matlab/simulink: Review of tools for research and education in mechatronics. In *2011 IEEE International Conference on Mechatronics*, pages 881–886, April 2011.

[6] Christoph Lameter. Numa (non-uniform memory access): An overview. *Queue*, 11(7):40:40–40:51, July 2013.

[7] Andrew McPherson and Victor Zappi. An environment for submillisecond-latency audio and sensor processing on beaglebone black. In *Audio Engineering Society Convention 138*. Audio Engineering Society, 2015.

[8] Derek Molloy. *Exploring BeagleBone: Tools and Techniques for Building with Embedded Linux*. John Wiley & Sons, 2014.

[9] E. Monmasson and M. N. Cirstea. Fpga design methodology for industrial control systems — a review. *IEEE Transactions on Industrial Electronics*, 54(4):1824–1842, Aug 2007.

[10] Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A Definition and Classification of Timing Anomalies. In Frank Mueller, editor, *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2006. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[11] Stefano Rinaldi, Paolo Ferrari, and Matteo Loda. Synchronizing low-cost probes for iec61850 transfer time estimation. In *Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS), 2016 IEEE International Symposium on*, pages 1–6. IEEE, 2016.

[12] H. Sanchez, B. Kuttanna, T. Olson, M. Alexander, G. Gerosa, R. Philip, and J. Alvarez. Thermal management system for high performance powerpc/sup tm/ microprocessors. In *Proceedings IEEE COMPCON 97. Digest of Papers*, pages 325–330, Feb 1997.

[13] Alan J. Smith. Disk cache — miss ratio analysis and design considerations. *ACM Trans. Comput. Syst.*, 3(3):161–203, August 1985.

[14] Daniel J. Sorin, Mark D. Hill, and David A. Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[15] Per Stenström, Truman Joe, and Anoop Gupta. Comparative performance evaluation of cache-coherent numa and coma architectures. *SIGARCH Comput. Archit. News*, 20(2):80–91, April 1992.

Preprint submitted to 56th IEEE Conference on Decision and Control.
Received March 20, 2017.