Building Real-Time Embedded Applications on QduinoMC: A Web-connected 3D Printer Case Study

Zhuoqun Cheng
Boston University
czq@cs.bu.edu

Richard West
Boston University
richwest@cs.bu.edu

Ying Ye
Boston University
yingy@cs.bu.edu

***Abstract*—Single Board Computers (SBCs) are now emerging with multiple cores, ADCs, GPIOs, PWM channels, integrated graphics, and several serial bus interfaces. The low power consumption, small form factor and I/O interface capabilities of SBCs with sensors and actuators makes them ideal in embedded and real-time applications. However, most SBCs run non-real-time operating systems based on Linux and Windows, and do not provide a user-friendly API for application development. This paper presents QduinoMC, a multicore extension to the popular Arduino programming environment, which runs on the Quest real-time operating system. QduinoMC is an extension of our earlier single-core, real-time, multithreaded Qduino API. We show the utility of QduinoMC by applying it to a specific application: a web-connected 3D printer. This differs from existing 3D printers, which run relatively simple firmware and lack operating system support to spool multiple jobs, or interoperate with other devices (e.g., in a print farm). We show how QduinoMC empowers devices with the capabilities to run new services without impacting their timing guarantees. While it is possible to modify existing operating systems to provide suitable timing guarantees, the effort to do so is cumbersome and does not provide the ease of programming afforded by QduinoMC.**

I. Introduction

The Internet-of-Things (IoT) is leading to a revolution in areas such as manufacturing, robotics, driverless cars, autonomous drones, and intelligent home automation systems. For IoT devices to interoperate with one another they need to be able to connect and exchange data. Consider, for example, a web-connected 3D printer with the capability to receive and spool remote printing requests via a webserver. A farm of these printers might then operate together to serve multiple remote user requests, as part of a distributed manufacturing facility. However, current consumer-grade 3D printers are not yet able to spool requests like a 2D printer, let alone coordinate their operations across a network.

Many consumer-grade 3D printers are based on relatively simple microcontrollers such as the Arduino Mega. These controllers incorporate several stepper motor drivers, general-purpose I/O pins (GPIOs) and analog-to-digital converters (ADCs), for motion, extruder, heat-bed and fan control. Examples include the RepRap printing platforms that run Arduino-based firmware such as Marlin. While these platforms provide low-latency I/O with sensors and actuators, they are unable to run a real-time printing control program along with web services, to interoperate with other devices or remote users.

To provide greater capabilities for IoT applications, numerous single board computers (SBCs) are now emerging. These SBCs often feature multiple cores, ADCs, GPIOs, pulse-width modulation (PWM), integrated graphics, networking, and several serial bus interfaces. Examples include the Intel Edison, Minnowboard MAX (MinnowMax) and Joule, which feature multicore Atom x86-based processors. Yocto and other variants of Linux are typically targeted at these platforms, to manage the complexity of their hardware. However, Linux is not ideal for real-time embedded application development, due to both unpredictable, non-negligible system overheads and the lack of a uniform, intuitive programming interface.

To take full advantage of these new hardware platforms requires the use of an operating system and a suitable programming interface. Ideally, the operating system needs to be real-time, to satisfy the requirements of many robotics, 3D printing, drone, and smart devices. Additionally, the programming interface needs to be simple and easy to use, with the ability to specify timing requirements for GPIOs, PWM signals, tasks and interrupts.

In this paper, we present QduinoMC, a programming interface and runtime environment for real-time, multithreading applications on multicore architectures. QduinoMC adopts the simplicity of the Arduino API, with support to take advantage of hardware-level parallelism in a predictable manner. While being backward-compatible with the existing Arduino API, QduinoMC provides several key interface extensions to enable: (1) the construction of multiple time-constrained loops, (2) temporal isolation between loops, (3) the assignment of loops to cores, (4) the mapping and masking of interrupts for specific cores, (5) real-time communication via semi-/fully-asynchronous channels between loops, and (6) the time-bounded delivery of interrupts to user-level services.

We show the utility of QduinoMC by applying it to the management of a web-connected 3D printer that we built. Our web-connected 3D printer is based on a Printrbot Simple Metal, with a replacement controller that uses a MinnowMax SBC running QduinoMC. Results show how our 3D printer is able to perform multiple tasks, while ensuring their temporal isolation. In contrast, the same tasks running on an embedded Yocto Linux 3D printer controller are not able to achieve the same timing guarantees. Consequently, QduinoMC is able to produce successfully printed objects within certain time bounds while a system running Yocto Linux is not. When Linux is able to successfully print a 3D object, it takes far longer than an equivalent system based on QduinoMC. In manufacturing, the time to produce items is critical to productivity, making QduinoMC far more suitable for smart manufacturing devices.

Contributions of this paper include: (1) an analysis and case-study of the real-time issues in a 3D printer, (2) QduinoMC, an API and runtime environment for real-time embedded applications, and (3) the design and evaluation of a web-connected 3D printer on a custom printer controller.

The rest of this paper is organized as follows: Section II provides background to the timing requirements for a web-connected 3D printer. Section III describes the implementation of, and problems with, a prototype 3D printer running Linux. Section IV follows with an overview of QduinoMC, including its real-time properties, programming interface, and support for multicore platforms. Comparisons between our printer running on QduinoMC to the ones on equivalent systems is given in Section V. Related work is discussed in Section VI, followed by conclusions and future work in Section VII.

II. Background

In this section, we describe the operation and timing requirements of a popular class of current 3D printers. We then briefly describe the added features and system challenges to support web-connected 3D printers.

A. *Mechatronics in FDM 3D Printers*

3D printers are classified by the filament materials they use and the way layers of filament are deposited to create parts. Most of the 3D printers in the consumer market utilize fused deposition modeling (FDM), which produces objects by extruding small beads of material that harden immediately to form layers on a movable bed. These printers have 3 axes of motion, with stepper motors to control the bed's movement along the X and Y axes, and the extruder's movement along the Z axis (Figure 1). An additional stepper motor feeds the filament through the extruder to maintain the correct bead deposition rate.

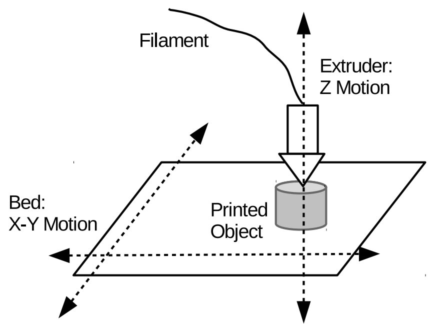

Fig. 1: A Conceptual View of a 3D Printer

Motor Control in 3D Printers. In a 3D printer, stepper motors combine with linear motion systems, such as pulleys and belts, to control the position and speed of the bed and extruder. Stepper motors are able to make precise angular movements. A stepper motor divides its full rotation into a number of equal steps. A microcontroller and driver circuit commands the motor to rotate and hold at one of these steps without a feedback sensor. Motor steps are triggered by a series of digital pulses sent to a GPIO pin, which feeds a driver circuit (e.g., Pololu 4988) that produces directional current within the motor windings. Electromagnets are switched on and off to attract teeth on the motor's shaft, causing it to rotate. This makes the shaft rotate one step for each pulse. Higher rotational accuracy is achieved by using driver circuits that generate microsteps. Similarly, rotational speed is affected by the frequency at which digital pulses are generated.

Control of the stepper motor's acceleration and deceleration is needed to ensure smooth linear movement. The time delay between the stepper motor pulses must be calculated, so that the motor's operation follows the trapezoidal speed ramp (example shown in Figure 2) as closely as possible.

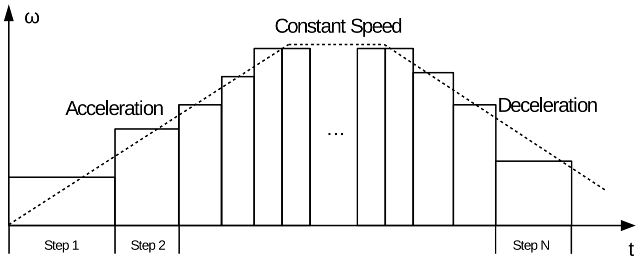

Fig. 2: An Example Speed Ramp for a Stepper Motor

B. *Timing Requirements*

For a better understanding of the timing issues, we use a Printrbot Simple Metal 3D printer as an example. The Printrbot Simple Metal uses one of its four NEMA (National Electrical Manufacturers Association) 17 stepper motors with a belt driven system to move the bed along the X axis. The NEMA 17 has 200 full steps per revolution, denoted as s. Each step is divided by $\alpha = 16$ microsteps, to achieve more precise motor rotation with less vibration. Thus, $s \cdot \alpha = 200 \times 16 = 3200$ digital pulses are needed per revolution. The Printrbot uses a GT2 belt driven system, with a belt tooth pitch, $b = 2\ mm$, and pulley tooth count, $p = 20$. This means one rotation of the stepper motor will be translated to $b \cdot p = 2 \times 20 = 40\ mm$ linear movement of the bed. For Θ radians of rotation, the relationship between linear motion speed, v, and pulse frequency, f, is as follows:

$$v = \frac{\Delta S}{\Delta t} = \frac{\frac{\Delta\Theta}{2\pi} \cdot b \cdot p}{\Delta t} = \frac{\frac{\Delta t \cdot f \cdot \frac{2\pi}{s \cdot \alpha}}{2\pi} \cdot b \cdot p}{\Delta t} = \frac{bp}{s\alpha} f \qquad (1)$$

After the extruder is heated, we assume melted filament continuously flows out of the extruder tip hole at speed γ when material needs to be deposited. Within a time period of Δt, using an extruder tip with diameter d, filament of volume $\Delta V = \gamma \Delta t \cdot (\frac{d}{2})^2 \pi$ will be extruded. We assume that during this process, the pulse train frequency is steady at f. Using Equation 1, the relative displacement of the bed to the extruder is $\Delta S = v\Delta t = \frac{bp}{s\alpha} f \Delta t$. So filament of volume ΔV is distributed on a rectangle of area $\Delta A = \Delta S d$. This reveals the relationship between layer thickness, H and pulse frequency, f:

$$H = \frac{\Delta V}{\Delta A} = \frac{\gamma \Delta t \cdot (\frac{d}{2})^2 \pi}{\frac{bp}{s\alpha} f \Delta t \cdot d} = K \cdot \frac{1}{f} \qquad (2)$$

where $K = \frac{\pi}{4} \cdot \frac{s\alpha d}{bp} \cdot \gamma$.

Equation 2 indicates that, for a given printer configuration and filament material, an unstable pulse train will cause uneven thickness of a printed layer. If the task that generates the pulse experiences a delay δt, due to temporal interference from other tasks or delays within the control system, the pulse frequency will drop to $f\prime = \frac{1}{\frac{1}{f} + \delta t}$. According to Equation 2, this will result in a vertical gap of height ΔH,

$$\Delta H = K(\frac{1}{f\prime} - \frac{1}{f}) = K\delta t \qquad (3)$$

between adjacent layers and therefore undermine the structural soundness of the printed object. This example is illustrated in Figure 3 where the delay δt happens at the $(n+1)$th step.

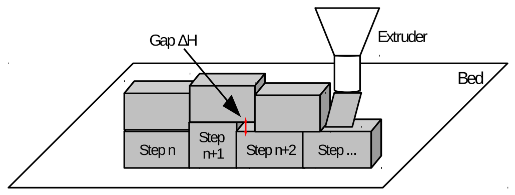

Fig. 3: Structural Deficiency due to Unstable Pulse Train

C. A Web-connected 3D printer

Traditional 3D printers are tethered to a PC via a USB link, to receive manual configurations or low-level commands (G-codes [1]) for one printing job. In some cases, they are able to read G-codes from a local SD card. In contrast, a web-connected 3D printer provides an interface to interact with either end users or other printers on the web. It is also capable of spooling multiple job requests, which is a common feature of 2D printers. Web-connected 3D printers form the basis for more sophisticated farms of manufacturing devices that coordinate their operation.

A 3D printer is required to translate G-codes into peripheral I/O control sequences. More advanced features might include running a local compute-intensive *slicer* engine to translate 3D modeling files into 2D sequences of G-codes, for each layer of a print job. At the same time, time-critical I/O signaling operations are necessary to drive stepper motors, fans and heaters. For a high volume of remote requests, interrupts from the disk and the network interface controller start to consume a large portion of CPU cycles. This poses challenges to balance system resources among several types of tasks: those performing latency-sensitive versus high-bandwidth I/O operations, and those involved in G-code processing.

III. Implementation on Linux

This section describes the web-connected 3D printer we built on Linux, including both hardware and software setup. It then identifies the problems we observed during building and testing this prototype.

A. Hardware

While we reused the mechanical parts of the Printrbot Simple Metal 3D printer, we replaced the original Printrboard controller with our custom controller. Most traditional 3D printers, including the Printrbot, are equipped with the AVR ATmega microcontroller, operating at speeds up to 20 MHz, and run firmware on the bare metal. In such an environment, however, it is difficult if not impossible to run a webserver in parallel with the printing control software. Therefore, our custom 3D printer controller is based on the much more powerful Intel MinnowMax board. The MinnowMax is equipped with a 64-bit dual-core Atom 1.33 GHz processor, 2 GB memory and 86 GPIOs, of which some are configurable as I2C, SPI, UART and PWM pins.

We interfaced the MinnowMax to a RepRap Arduino Mega Polulo Shield (RAMPS) and various analog circuits to level-shift the 3.3V GPIO pins to appropriate values, to control the Printrbot's motors, fan and extruder heater. Our custom controller uses four Pololu 4988 stepper motor drivers and an ADS7828 8-channel 12-bit I2C analog-to-digital converter (ADC), to monitor extruder temperature readings. A snapshot of the controller board is shown in Figure 14 and the controller circuit diagram is shown in Figure 18. Further information is available on our project webpage [2].

B. Software

We ran Yocto Linux 4.4.13 with the PREEMPT_RT patch on the MinnowMax. A lighttpd daemon runs in the background to receive remotely submitted printing jobs, and a custom spooler queues and feeds jobs to the printing control program, which is a customized version of Marlin.

Marlin is a firmware for RepRap single-processor electronics, supporting RAMPS, RAMBo, Ultimaker, BQ, and several other Arduino-based 3D printers. It has one Arduino *loop* function and two interrupt handlers for a pair of hardware timers, illustrated in Figure 4. In each iteration of the *loop*, a G-code is read from the serial bus and is processed. Though there are hundreds of different G-codes, the most common are G0 and G1, which specify the speed and position for the extruder to move. Positional coordinates are then translated into each stepper motor's direction, number of steps and angular speed (according to the speed ramp). This is the core algorithm of Marlin and consumes the most CPU cycles. As the last stage, the *loop* packages the stepper motor motion parameters into a data block and adds it to a finite capacity queue, after which the main loop repeats.

The block queue is shared between the *loop* and an interrupt handler, which is bound to a hardware timer in one-shot mode. When the timer fires, the interrupt handler is invoked to check the remaining steps to execute on each axis, and their respective rate. It then, accordingly, sends one pulse to each stepper motor that should be driven and programs the timer for when the next pulse should occur. When the handler completes all the steps in a block, it consumes a new one from the queue if it is not empty.

Marlin has a second interrupt handler bound to a hardware timer in periodic mode. Every 8 milliseconds it samples the extruder temperature, and adjusts the fan speed and heater settings using PID feedback control. Typical PLA plastic filament, for example, requires a temperature of about 208-210 degrees Celcius, and the PID controller is used to keep the target and actual temperature within a small error margin.

We ported Marlin to run as a multithreaded Linux application. The *loop* function and the two interrupt handlers are each converted to run as an individual thread. Proper locking is applied to data shared among threads. GPIO and I2C operations are performed via the *mraa* library from Intel's IoT Dev Kit, to replace AVR I/O instructions. Instead of programming hardware timers directly, we rely on *nanosleep* to perform timed operations. We also refactored the program structure by moving the PID controller to the thread that runs the original Timer2 interrupt handler, from the thread that runs the original *loop* function. In the original Marlin firmware, the PID control code is in the *loop* function, because its execution takes a relatively long time and renders the system unresponsive if invoked in an interrupt handler. The new Marlin application-level code is optimized for the real-time preemption patch by carefully setting its scheduling priority, and locking all pages into memory [1].

[1]G-code is a numerical control programming language for computer-aided manufacturing.

[2]https://www.cs.bu.edu/~richwest/smartprint3d.php

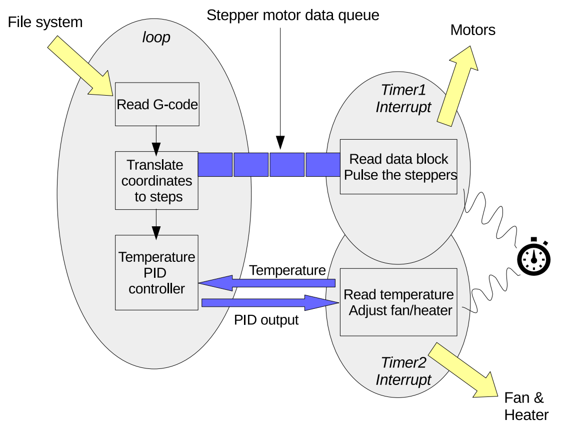
File system
loop
Stepper motor data queue
Motors
Read G-code
Timer1 Interrupt
Translate coordinates to steps
Read data block Pulse the steppers
Temperature PID controller
Temperature
PID output
Read temperature Adjust fan/heater
Timer2 Interrupt
Fan & Heater

Fig. 4: The Structure of Marlin

C. *Observations*

We carefully tuned the 3D printer and it successfully printed out our test objects [3]. This was done under the circumstances that the webserver was disabled during printing. We then proceeded to enable the webserver and start submitting G-code files while printing was active. We used a script to automatically generate submission requests. The submission intervals were random numbers uniformly selected from 100 to 10000 milliseconds. The file sizes were randomly selected from a log-uniform distribution, ranging from 50 KB to 150 MB [4]. Each file was then submitted to the printer via an HTTP POST request. We were able to observe evident jitter of the extruder relative to the bed. Backlash noise from the stepper motor gearing and belt drive was evident for an active job when a file larger than 15 MB was being transferred.

TABLE I: Case Descriptions

Case #	Description
Case 1	Webserver/Spooler Disabled. No Job Submitted.
Case 2	Webserver/Spooler Enabled. Jobs Submitted from the Script.

In order to quantify our observations, we created two micro-benchmarks to examine the interference by the webserver and spooler on: (1) the timely execution of the temperature control thread, and (2) the stepper motor control thread. In the first experiment, we logged the temperature readings from the thermistor under the two cases in Table I. The sample period was set to 1 second. The extruder was heated to 209 degrees Celcius, and kept at that value during printing. Results are plotted in Figure 5. It can be seen that, despite the continuous reception of submitted files, the PID controller maintained the temperature close to the setpoint in both cases. The 8 ms period of the temperature control thread is large enough to hide delay variations caused by Linux.

In the second experiment, we measured the frequency of the pulse train that drives the stepper motors. However, when a 3D printer is printing, the pulse train changes its frequency in different G-code commands and phases in the speed ramp. It is hard to tell if an observed frequency change is actually caused by interference. Therefore, we wrote a micro-benchmark program to drive a stepper motor at a constant speed. Listing 1 shows our benchmark code. Variable *period*

[3]A 3mm cube, fan shroud, Ultimaker robot, and the object in Figure 15.

[4]It is realistic to have a G-code file exceeding 100 MB, if the object to print is relatively large, requires thin layers and dense infill.

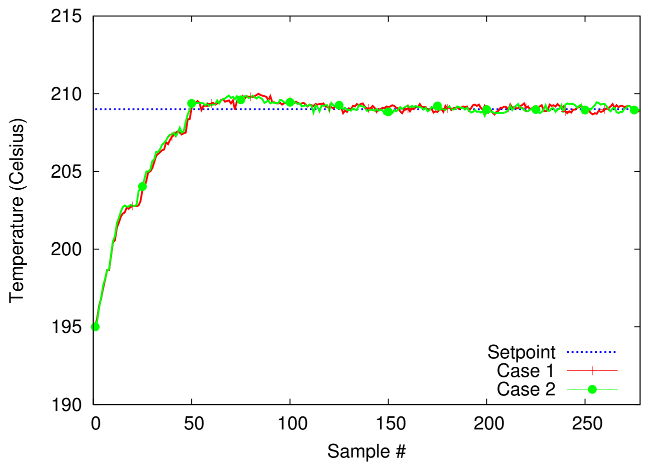

Fig. 5: Temperature Readings

was set to 100 microseconds, to generate a 10kHz pulse. Using Equation 1, this yields an expected linear speed of 125 mm/s. When the webserver is disabled, an oscilloscope showed a pulse wave of variable frequency, ranging from 7.75 to 8.02 kHz, shown in Figure 10. After we started the script to submit jobs, the frequency showed more fluctuation, even dropping to as low as 6.42 kHz.

Listing 1: Loop to Measure Frequency

```
struct timespec period =
    {.tv_sec = 0, .tv_nsec = 100000};
for (;;) {
  nanosleep(&period, NULL);
  /* write 1 to gpio6 */
  mraa_gpio_write(GPIO6, HIGH);
  /* write 0 to gpio6 */
  mraa_gpio_write(GPIO6, LOW);
}
```

D. *Operating System Control*

The second experiment reveals two problems we faced when generating a precise pulse train on the Linux platform. First, the frequency range is too low to drive the motors at their expected speed. Second, the stepper motor control thread is subject to interference from other tasks, causing jerks in the stepper motor's rotation, due to backlash. In this section, we take a look into the causes of these two problems.

We reran the benchmark shown in Listing 1 with the timestamp counter recorded at various stages within the program and the kernel, to determine the root cause of these costs. Results are shown in Table II. All times are averaged over 1,000 samples, and shown in nanoseconds (ns). The average time of one iteration is 126,422 ns, resulting in a 7.91 kHz pulse train. Except for the times spent sleeping, and setting the GPIO value register, the others are operating system overheads, which fall into four categories:

- *nanosleep Kernel Crossing* – from user space to kernel and back, which accounts for $\sim$0.5% of the execution time.
- *The hrtimer Subsystem* – for high-resolution kernel timers. The `nanosleep` call uses the hrtimer subsystem to access the highest precision hardware timing mechanism, either local APIC timers or the high precision event timer (HPET).
- *Context Switch Overhead* – to put a task to sleep, run the scheduler, and wake up the next task.

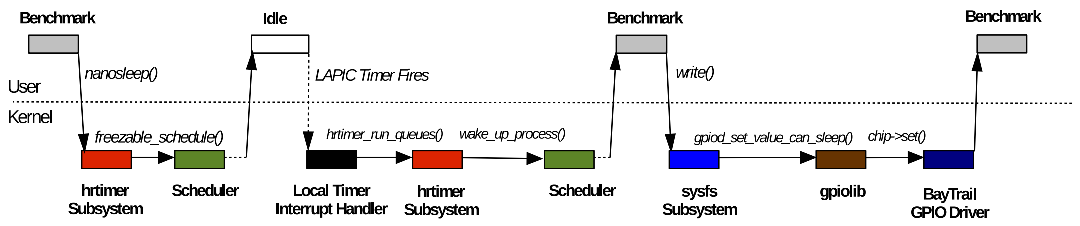

Fig. 6: Call Graph for Listing 1

- *The GPIO Framework* – which interfaces GPIO pins with user-space programs using *sysfs*. This includes the general GPIO framework (gpiolib) and the actual GPIO controller driver for the Minnow MAX (the BayTrail GPIO driver). This accounts for roughly 12.5% of the execution time.

TABLE II: Kernel Overheads (in nanoseconds)

	Costs	Percentage
nanosleep Kernel Crossing	587	*0.5%*
Timer Framework	2420	*1.9%*
In sleep	100000	*79.1%*
Context Switch	7744	*6.1%*
sysfs Framework	10592	*8.4%*
gpiolib Framework	1038	*0.8%*
GPIO Controller Driver	4146	*3.3%*
Total	126422	*100%*

GPIO Framework Overhead. As shown in Table II, the biggest overhead stems from GPIO operations. Intel development boards, such as Galileo, Edison, MinnowMax and Up, all have PCI-based GPIO controllers with memory-mapped device registers. On top of the GPIO controller drivers, Yocto Linux uses a uniform GPIO sysfs interface for userspace. The sysfs interface provides a simple way to manually check the status of an input pin or write values to output pins, from the command line or a shell script. However, an autonomous control program suffers from this convenience, by paying the cost of going through convoluted kernel control paths before reaching the actual GPIO driver to set the value register, as shown in Figure 6. Instead, a simple userspace GPIO driver is all that the control system really needs.

Context Switch Overhead. The *nanosleep* system call is invoked whenever a Linux task wishes to relinquish the processor and sleep for a precise period of time. This call sets a high-resolution timer (hrtimer) to fire at some future point in time, when the task will be woken up and allowed to run again on the processor. As shown in Figure 6, *nanosleep* programs the Local APIC timer and calls into the kernel scheduler. In a tickless kernel, the next LAPIC timer interrupt will check the timer and call the scheduler to wake the task up. The cost of going through the hrtimer subsystem, (re-)programming the LAPIC timer, handling the timer interrupt and running the scheduler adds up to more than 10 microseconds.

Timing Unpredictability. There is a trend in the operating system community to optimize away microsecond-level latency, with a focus on minimizing the costs of network and storage operations in datacenter applications [2]. However, unlike disk and network interface controllers that use direct-memory access (DMA) to transfer high-bandwidth (bulk) data, GPIOs are used for latency- and jitter-sensitive control of sensors and actuators. As explained earlier, a fluctuating pulse train results in jerky stepper motor movements in 3D printer control. This is exacerbated by the lack of temporal isolation between Linux processes, and the interference from interrupts on process execution.

E. Further Optimizations

We recognize that Linux can be further optimized for latency-sensitive real-time applications, such as by using the *cset* utility to isolate a CPU, making use of the SCHED_DEADLINE scheduling class, or enabling CONFIG_NO_HZ_FULL to disable timer interrupts on certain cores. However, all these techniques involve system utilities or kernel settings without a uniform programming interface. Significant kernel and real-time expertise is required to take advantage of these features, and thus raises barriers to embedded application development.

IV. Implementation on QduinoMC

Rather than focusing on the optimization of Linux to meet the timing requirements of a web-connected 3D printer, we decided to investigate a programming approach applicable to a wide variety of time-critical applications on emerging SBCs. We started with our earlier work on Qduino [3], a programming interface and runtime environment and extended it to QduinoMC. QduinoMC provides additional support for multicore architectures. In this section, we first describe the real-time task model that characterizes a web-connected 3D printer. We then briefly summarize the key features of Qduino, followed by a detailed description of the extensions made in QduinoMC. Finally, we describe the implementation of the 3D printer on this new platform.

A. Task Model and Scheduling

Tasks are scheduled using Rate-Monotonic Scheduling (RMS) [4], and admission control involves both RMS utilization bound and completion time tests [5]. The Sporadic Server [6] model is used for main tasks and Priority Inheritance Bandwidth preserving Servers (PIBS) [7] for threaded bottom halves. Bottom halves are the deferrable (i.e., schedulable) parts of interrupt handling.

Variable Period Task. We model the web-connected 3D printer control program as a set of n real-time tasks $T = \{\tau_1, \tau_2, ..., \tau_n\}$. Each task τ is characterized by its worst-case execution time (WCET) C_i and period T_i. For periodic tasks, the C_i and T_i are fixed during execution. However, the pulse-generating task changes its period according to current step number and the pre-calculated speed ramp, as illustrated in Figure 2. Given a speed ramp, the period of the pulse-generating task follows the pattern below.

Suppose the stepper motor is in the acceleration phase, with angular acceleration $\dot{\omega}$ based on a changing velocity ω. The pulse-generating task generates the nth pulse at time t_n. The angular displacement θ_n, relative to θ_0, at time t_n is:

$$\theta_n = \frac{1}{2}\dot{\omega}{t_n}^2 = n\phi \tag{4}$$

where ϕ is the angular displacement of each step, and θ_0 is assumed to be 0. t_n is then expressed as:

$$t_n = \sqrt{\frac{2n\phi}{\dot{\omega}}} \tag{5}$$

It follows that the period, T_n, between the nth pulse and $(n+1)$th pulse is:

$$T_n = t_{n+1} - t_n = \sqrt{\frac{2\phi}{\dot{\omega}}}(\sqrt{n+1} - \sqrt{n}) \tag{6}$$

A Taylor series 2^{nd} order approximation on Equation 6 gives:

$$T_n = T_{n-1}(1 - \frac{2}{4n+1}) \tag{7}$$

Equation 7 confirms that the period decreases in the acceleration phase. Conversely, it is easy to see that the period increases in the deceleration phase. Therefore, the sum of each task's utilization reaches its peak when the printer's linear motion system is running at its maximum speed. Thus, for a given configuration of a 3D printer, we apply a schedulability test on the control software using each task τ_i's parameters, C_i and T_i, assuming a maximum print speed. For example, the default maximum speed on the X axis is set to 125 mm/s in the Marlin firmware. Using Equation 1, and parameters given in Section II-B, the period of the pulse-generating task is calculated to be 100 microseconds.

I/O-Intensive Task. The webserver and spooler interact with underlying disk and network interface controllers that use direct-memory access (DMA) to transfer high-bandwidth (bulk) data. During a high volume of job submissions, the rate of interrupts has the potential to interfere with the timely execution of tasks that perform latency- and jitter-sensitive I/O operations for sensor and actuator control.

To temporally isolate interrupts from other main tasks, we execute interrupt bottom halves in a deferrable thread context, and the cycles for interrupt handling are charged to the bottom half thread's budget. This way, the handling of an interrupt does not steal CPU cycles from a currently running, potentially time-critical task. Instead of using the Sporadic Server model for both main tasks and bottom half threads, an I/O event is serviced by a PIBS. As with a Sporadic Server, PIBS uses replenishments but instead of a list there is only a single replenishment. If a Sporadic Server were used, it would need to maintain a replenishment list, to keep track of when in time each part of its budget was available. This is essential for correct timing behavior. However, short-lived ISRs lead to significant fragmentation of replenishment list budgets, requiring frequent reprogramming of timers (e.g., LAPIC x86 one-shot timers) to denote when budget fragments for a VCPU become available. A PIBS also differs from a Sporadic Server in that it will not execute again until the next replenishment time, regardless of whether it has utilized its entire budget or not.

It is shown in [8], by using PIBS for interrupt bottom half threads, the scheduling overheads are reduced due to reduced context switching and timer reprogramming. While a system of Sporadic Servers and PIBS has a slightly lower schedulability than a system of only Sporadic Servers from a theoretical point of view, in practice an implementation of both scheduling policies results in a system of Sporadic Servers and PIBS outperforming a system of only Sporadic Servers.

B. Qduino

Qduino is a predictable, multithreaded Arduino system built on the Quest real-time operating system. Quest operates on 32-bit x86 architectures and has support for kernel threads including threaded interrupt handlers, POSIX threads, and a network protocol stack based on lightweight IP (lwIP) [9].

Quest features a two-level scheduling hierarchy, with threads mapped to virtual CPUs (VCPUs) and VCPUs mapped to physical CPUs (PCPUs). In effect, VCPUs are resource containers [10] for threads that are assigned to them. They account for budget usage in specific windows of real-time. Each VCPU is specified a processor capacity reserve [11] consisting of a budget capacity, C, and period, T. 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 is passed when creating new VCPUs. All VCPUs assigned to the same core are scheduled using Rate-Monotonic Scheduling (RMS), where the VCPU with the smallest period has the highest priority. An admission controller runs RMS utilization bound and completion time tests on each core. Quest's scheduling subsystem provides temporal isolation between tasks and I/O events using two different classes of VCPUs: (1) Main VCPUs for conventional tasks, and (2) I/O VCPUs for interrupt bottom halves. Main VCPUs are implemented as Sporadic Servers and I/O VCPUs implement a PIBS scheme. Quest enables I/O VCPUs to be bound to Main VCPUs for the purposes of handling I/O requests from specific devices, or device classes, such as a network, USB, or GPIO class.

Qduino builds on top of Quest and provides users with an extended Arduino API which, while backward-compatible with the original API, supports real-time multithreaded sketches and event handling. While only one *loop()* function is allowed in the standard Arduino API, Qduino allows up to 32 loops, which is translated by the C preprocessor to a thread creation call, with the loop function being the thread routine. Thus, each loop is viewed as a schedulable entity by the underlying Quest scheduler. When one loop blocks on I/O, it yields execution to another loop instead of leaving the CPU idle. By default, each thread associated with a loop creates a new Main VCPU and binds itself to it, before repeatedly performing the *loop()* function. The budget and period of the VCPU are determined by the second and third *loop* arguments, whose time units default to milliseconds but are configurable in microseconds or clock cycles. A program written in Qduino should have separate loops for tasks with different timing requirements, if the temporal isolation between them is necessary.

Qduino supports inter-loop communication using semi-asynchronous ring buffers and Simpson's four-slot fully-asynchronous communication channels [12], [13]. Ring buffers allow lossless exchange of data through shared memory between producers and consumers. Communication imposes no delays on the sender or receiver, except when the buffer is full or empty. For completely asynchronous communication, four-slot channels are possible. These guarantee the freshness and coherence of data exchanges, ensuring the most recently written data object is always available to the reader, without being partially updated by interleaved reads and writes. This is particularly important in sensor-data processing, where a consumer cares more for the most recent (freshest) sensor reading than a full history of values.

C. Web-connected 3D Printer on Qduino

To support our custom 3D printer controller, we ported Qduino to the MinnowMax board, which required updating the ACPI driver to correctly parse tables in a newer version of the ACPI firmware. We modified the existing PCI device manager to recognize and handle the nonstandard PCI configuration space of the GPIO controller. We implemented the drivers for the on-board GPIO, I2C and NIC controller. We ported the multithreaded Marlin on Linux to a Qduino multi-loop sketch, with each thread being a loop. Interrupts from I2C and NIC are handled in respective I/O VCPUs. The stepper motor data queue is implemented using the semi-asynchronous ring buffers, and status variables shared between loops are exchanged using four-slot channels. A simple webserver and a spooler were implemented as Quest native processes, running as background tasks to the main printer control code.

A practical question to ask is how to decide the appropriate budget and period for a loop. For control applications, a loop period should be set to the sampling interval for an I/O signal. The budget should be sufficient to complete all computational requirements within one loop iteration. In some cases, this may require offline profiling, to determine how long it takes the instructions within a loop to execute. In hard real-time systems, worst-case execution time analysis is necessary. For less time-critical situations, a user is able to define loop timing constraints more loosely.

We switched the time units of all VCPU parameters to microseconds, from their default millisecond values. The temperature control loop period was set to 8 milliseconds, and the stepper motor control loop period was set to 100 microseconds, as derived in Section IV-A. We profiled the worst-case execution time of both loops, using a 1 millisecond budget (C) for the temperature loop and a 10 microseconds budget for the stepper loop. The G-code reading and translation loop was less time-critical but more CPU-intensive. Consequently, we set the budget and period for this loop to 30 milliseconds, and 100 milliseconds, respectively. The webserver and the spooler operated as non-real-time tasks, and were assigned to a default VCPU with $C = 10$ ms and $T = 100$ ms. All other low-priority system tasks used this default VCPU. The I/O VCPUs for I2C and network controller interrupts were both set to 1 ms budget and 100 ms period.

After careful tuning, the printer successfully printed out the three test objects. Again, we proceeded to do the two quantitative experiments described in Section III-C. Under both cases in Table I, the temperature control loop effectively maintained the extruder temperature around 209 degrees. In Case 1, the stepper motor control loop generated a relatively stable pulse train at 9.009 kHz (Figure 11), only occasionally dropping down to 8.975 kHz. However, in Case 2, when jobs were submitted using the script in III-C, the frequency started fluctuating, dropping to as low as 8.236 kHz. While the pulse train on Qduino has higher and more stable frequency than Linux, it still reveals one problem that Qduino has with handling tasks of strict timing requirements: the processing of top half (non-deferrable part of interrupts) and the following invocation of the scheduler impose non-negligible overheads when the system experiences a high volume of interrupts. The overheads affects the timely execution of latency-sensitive tasks running on the same core.

TABLE III: New APIs added in QduinoMC

Function Signatures	**Category**
`loop(loopID, budget, period [,coreID])`	Structure
`interruptsVcpu(device, budget, period [,coreID]), noInterrupts(device, coreID), noTimer(coreID), reTimer(coreID)`	Interrupt
`delayBusyMicroseconds(ms), delayBusyNanoseconds(ns)`	Time

D. QduinoMC: Leveraging Hardware Parallellism

In Qduino, the CPU affinity and interrupt routing is managed by the underlying kernel. This leads to possible problems when two or more time-critical loops share the same core that handles frequent interupts, while a non-real-time task occupies another core with only occasional hardware interrupts. This motivated us to develop QduinoMC, an extension to Qduino. QduinoMC provides easy-to-use APIs that allow developers to better specify application-level timing requirements on multicore architectures.

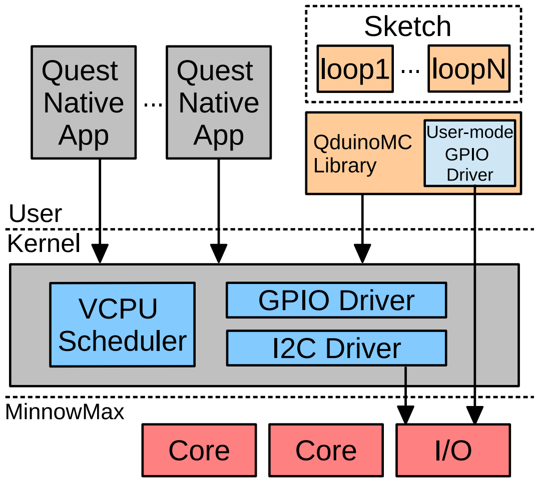

Fig. 7: QduinoMC Architecture Overview

In QduinoMC, the *loop* function allows the specification of multiple time-budgeted loops within the same program, which can be restricted to specific cores. The *Interrupts* API category is extended to support I/O bandwidth control as well as interrupt routing. Combining these two APIs, it is easy to provide a core with strict isolation. We further added two new time functions, *delayBusyNanoseconds()* and *delayBusyMicroseconds()*. These read the time stamp counter and wait until the specified time instance is passed. They are intended to be used for time-driven event loops. Within a task loop, GPIOs are manipulated via digital and analog I/O functions. On MinnowMax, the GPIO controller is a PCI-based device with memory-mapped registers. We wrote a user-mode GPIO driver by mapping the registers to a memory region with user-level access rights. QduinoMC's GPIO functions are wrappers around the user-mode driver functions. This way, the kernel crossing overheads are significantly reduced for loops that perform intensive GPIO operations. Newly added APIs are listed in Table III.

Loop Pinning. QduinoMC provides extensions to the *loop()* and *interruptsVcpu()* APIs, to pin a loop and a bottom half interrupt handler, respectively, on specific cores. Loops with different CPU utilizations may be load balanced across cores to guarantee their collective timing requirements. Interrupts are then delivered only to those cores that require them.

Additionally, loops with high-precision timing requirements can be assigned to dedicated cores, to avoid scheduling and context-switching overheads between separate threads. SBCs such as the Intel Up board have four cores, with the likelihood that this will increase further on future SBCs. It is therefore not impractical to dedicate a core to high-precision, time-critical tasks such as those operating at the nano- and micro-second resolution.

Interrupts. As stated earlier, Quest is capable of scheduling interrupt handlers as time-budgeted threads, to avoid interference with other tasks. Each threaded interrupt handler is bound to an I/O VCPU dedicated to that I/O device. The I/O VCPU budget prevents a high volume of interrupts being handled indefinitely, at the cost of other tasks. We exploit this feature by providing APIs for tuning the budget and period of I/O VCPUs so that a system designer can balance CPU time between CPU- and I/O-intensive tasks. The *interruptsVcpu()* function takes an I/O device type as the first parameter. Currently, QduinoMC supports GPIO, I2C and SPI, although we are considering additional support for UARTs and NICs. The second and third parameters specify the budget and period, respectively, of the I/O VCPU dedicated to that device.

QduinoMC also allows I/O device interrupts to be routed to a specified core. *interruptsVcpu()* accepts an optional fourth parameter to specify which core is associated with the threaded interrupt handler. To further isolate a core, hardware interrupts of a specified I/O device can be disabled on a specified core, by calling *noInterrupts()*. It internally modifies the I/O APIC registers to mask a core from the delivery destination of the interrupt line of the specified device.

noTimer() is used to disable the local APIC timer interrupts on a specified core. This is useful when a real-time task does not want to be interrupted, to allow another task on the same core to execute. As a consequence, all scheduling and context-switching overheads are eliminated. Interrupts from the local APIC timer are reactivated by a call to *reTimer()*.

In summary, we have developed QduinoMC to address problems that neither Qduino nor Linux-based embedded systems has completely solved. Pros of QduinoMC include:

- **Simple programming interface** – The QduinoMC API is based on the Arduino API, which is a popular programming model for physical computing. It provides a uniform and intuitive way to interact with I/O devices involving sensors and actuators, with the underlying OS being transparent to users.
- **Timing guarantees** – QduinoMC is designed for embedded applications comprising multiple tasks with different timing requirements on multicore platforms. QduinoMC supports the specification of real-time loops, as well as latency-sensitive I/O interrupt handling.

E. Marlin on QduinoMC

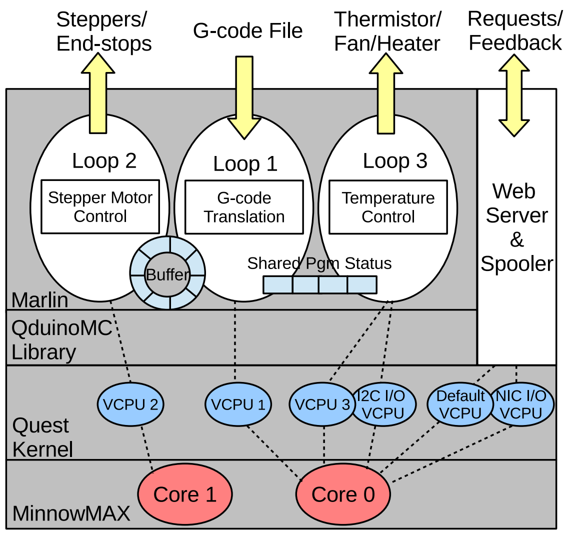

Fig. 8: Marlin on QduinoMC

We refactored the Qduino version of Marlin to take advantage of new features in QduinoMC. The new Marlin code maintained the three loops, shown in Figure 8. Apart from Marlin, a webserver and a spooler ran natively on the Quest OS, sharing the default Main VCPU. Therefore, four Main VCPUs existed in total in the system.

The second loop for the stepper motors had the most critical timing requirement. The other tasks associated with different VCPUs had weaker real-time requirements. Thus, we assigned *Loop 2* to a dedicated core on which all unnecessary interrupts were disabled. By calling *noTimer(1)* in the setup phase, the timer interrupt was disabled on Core 1 so that Loop 2 was free from the interruption of the scheduler. Interrupts were generated by the I2C controller as part of communication with the ADC, which recorded readings from a thermistor to monitor extruder temperature. A call to *noInterrupts(ALL, 1)* was used to suppress all interrupts, including I2C and NIC interrupts, on Core 1. The threaded bottom half of the I2C and NIC interrupt handler, running on respective I/O VCPUs, were automatically pinned to Core 0 by calling *noInterrupts(ALL, 1)*. Outside of the Marlin sketch, the webserver and the spooler ran on the default Main VCPU, which was pinned to Core 0. After these setup stages, Core 1 was able to run Loop 2 in isolation of all other loops and interrupt events.

V. Evaluation

This section evaluates the implementation of the web-connected 3D printer on the QduinoMC platform, in terms of its overheads, predictability and usability. Comparison tests involve an unmodified off-the-shelf Printrbot 3D printer, with a Printrboard controller based on an Atmel AT90USB1286, operating at 16 MHz. The Printrboard runs Marlin as firmware, lacking the capabilities of an operating system with web connectivity. Thus, experiments on the Printrbot are conducted *without* remote job submissions. Additional comparisons are shown for Linux (Section III-C) and Qduino (Section IV-B) running on our custom MinnowMax controller.

Temperature Control. We measured the temperature during printing under the two cases shown in Table I. The results are plotted against those of the Printrboard. As shown in

Figure 9, the execution of the PID controller in QduinoMC Marlin does not suffer interference by the webserver or the spooler. The PID controller maintains the temperature close to the setpoint, as effectively as the Printrboard.

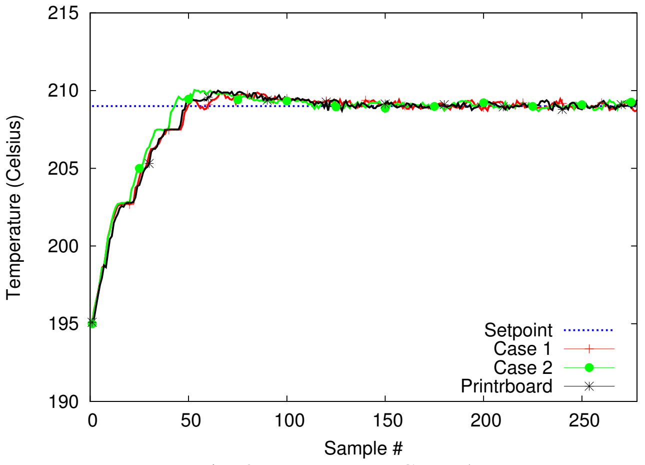

Fig. 9: Temperature Control

The 10 kHz Pulse Train. We also performed the 10 kHz pulse train experiment on QduinoMC. The benchmark sketch is shown in Listing 2. Due to the hard real-time nature of this single loop sketch, we pinned the loop to Core 1 and disabled all interrupts on that core in the setup phase. The C and T parameters were both set to 100, to indicate the task's dedication to the whole core. Oscilloscope readings yielded a pulse train with a stable frequency of 9.569 kHz (Figure 12). We then enabled the webserver and spooler in the default Main VCPU, that is pinned to Core 0 by default. After we started the script described in Section III-C to submit jobs, the pulse frequency stayed at 9.569 kHz.

Listing 2: The 10 kHz Pulse Train Sketch

```
int GPIO6 = 6;
void setup() {
  pinMode(GPIO6, OUTPUT);
  noInterrupts(ALL, 1);
}
void loop(1, 100, 100, 1) {
  delayBusyNanoseconds(100000);
  digitalWrite(GPIO6, 1);
  digitalWrite(GPIO6, 0);
}
```

The Printrboard has four hardware timers, which are programmable with a maximum frequency as fast as the system clock. We modified the Marlin firmware to set the Timer1 prescaler to 8, resulting in a 2 MHz timer. We then put Timer1 in the Clear Timer on Compare (CTC) mode and programmed the Output Compare Register to 200. Under this setting, the timer incremented the counter by 1 every 500 nanoseconds. When the counter reached 200, it fired an interrupt. The interrupt handler did nothing but generate a pulse on the GPIO6 pin, and then reset Timer1. An oscilloscope showed a stable pulse frequency of 9.96 kHz (Figure 13) on GPIO6, which was very close to the theoretical 10 kHz[5]. However, this was without any background tasks running, and served as a reference for the performance of all other scenarios.

Experiments show that while the pulse frequency on QduinoMC was slightly lower than the one on the Printboard, it was more than 21% higher than the one on Yocto Linux, and 6.2% higher than with Qduino. More importantly, the pulse train on QduinoMC was more stable, especially when the system load was high. While QduinoMC was able to maintain a pulse frequency close to that of the Printrboard, it did so while handling background web requests.

Example Test Object. We developed a 3D test object to work as a wheel encoder for a mobile robot, as shown in Figure 15. We compared the performance of QduinoMC against Linux to print the object, while our job submission script was actively communicating with the webserver. Given the inability of Linux to maintain a high stepper pulse rate, the whole print process ran at a slower speed to produce a successfully printed object, as shown in Figure 16. The Linux object was still not printed to the same quality as with QduinoMC, and took more than one hour to complete. With Linux, jitter and delays in stepper motor speeds for both printer movement and extrusion caused jerky operation and fine strands of unwanted filament being deposited. In contrast, QduinoMC printed the object (Figure 17) in about thirty minutes, which was similar to the Printrboard without the ability to support web requests. QduinoMC is able to empower a 3D printer with additional services without impacting manufacturing time, which is critical in a production system.

Platform Usability. One of the goals of QduinoMC is to provide easy-to-use APIs for application development. The pulse train generation sketch for QduinoMC in Listing 2 required only 10 lines, while the Yocto Linux code in Listing 1 needed 35 lines [6]. We also observed a 10% source code reduction in Marlin, with Linux Marlin having 3,674 lines of code, and QduinoMC Marlin having 3,342. The minimal system overheads incurred by QduinoMC compared to those of the Printrboard firmware are compensated by its ability to work in conjunction with a richer set of embedded application services (e.g., web services). QduinoMC is a platform on which to develop more sophisticated IoT applications.

VI. Related Work

Contiki [14] is an operating system for embedded devices. The Contiki programming model is based on protothreads [15] that shares features of both multithreading and event-driven programming. Applications are written in a subset of the C programming language. RIOT OS [16] is another operating system designed for embedded devices. It provides full multithreading and real-time abilities. RIOT allows application programming in C and C++. Both Contiki and RIOT are aimed at developing applications on resource-constrained wireless networking devices. However, QduinoMC is a system designed for physical computing on more powerful multicore platforms. It provides a simpler application programming interface for developing embedded applications that perform both CPU-intensive and latency-sensitive tasks.

ERIKA Enterprise is an OSEK/VDX real-time kernel [17] that supports time-sensitive applications on a wide range of microcontrollers and multicore platforms. ERIKA supports EDF scheduling of engine control applications [18]. Similar to the web-connected 3D printer, engine control applications are typically characterized by a set of periodic tasks and adaptive

[5]The Printrboard uses 5V logic levels.

[6]Scheduling priority settings, memory locking and error checking are not shown in Listing 1.

variable-rate (AVR) tasks whose activation time is determined by the current engine rotation speed. A detailed analysis of scheduling AVR tasks is given in [19]. Apart from the OSEK-compliant API, ARTE [20] is an Arduino extension built upon ERIKA to support multitasking real-time applications. While ARTE is designed for single-core hardware devices (e.g., Arduino Uno and Arduino Due), QduinoMC exploits multicore architectures to improve performance and temporal isolation amongst tasks with hard real-time requirements.

To the best of the authors' knowledge, there is no published work on building intelligent 3D printers. However, Replicape is an open source add-on board for BeagleBone and BeagleBone Black, to enable 3D-printing. It hosts a standard Linux distribution (Ångström/Debian) for running the G-code daemon, while real-time stepper timings are handled by the two on-chip Programmable Real-time Units (PRU) present on BeagleBone. It also supports a web interface for reading and writing configuration files. However, PRUs have to be programmed in a special assembly language. Our 3D printer does not require any special hardware to guarantee the real-time stepper timings, and the firmware is written using simple QduinoMC APIs. In addition, our web interface is not only used for configuration files but also for G-code transmission so that users can request 3D printing services remotely.

VII. Conclusion and Future Work

This paper presents QduinoMC, a platform to ease the process of developing applications with critical timing requirements on emerging multicore SBCs. We also describe a case study based on the development of a web-connected 3D printer, which has the ability to spool print requests via a webserver without being tethered to a remote computer. We analyze the real-time issues in a 3D printer, and compare the performance of a prototype controller running Linux, Qduino, and QduinoMC. Experiments show that stepper motor control requires precise timing at the microsecond resolution and the underlying scheduling and context switching overheads on Linux and Qduino are prohibitive to accurate task timing at this granularity. Even with the real-time preemption patch and carefully tuned thread priorities, the Linux Marlin 3D printing code was unable to maintain print speeds on par with either the original Marlin firmware on the Printrboard, or Marlin running on QduinoMC in the presence of web requests.

QduinoMC leverages multicore architectures and extends Qduino by enabling: (1) the assignment of loops to cores, and (2) the mapping and masking of interrupts for specific cores. With QduinoMC, we pinned the stepper control logic on a separate core without interference from interrupts or scheduling overheads. The printer built on QduinoMC displays lower overheads and better predictability.

Future work will extend QduinoMC with security features based on our Quest-V separation kernel. This way, untrusted users will be able to communicate and exchange data with secure IoT devices, without compromising the timing and integrity of safety- and timing-critical operations.

Acknowledgment

Thanks to the shepherd and reviewers for improvements to this paper. This work is supported by a gift from Intel Corporation, and the National Science Foundation (NSF) 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.

References

[1] P. E. McKenney, "'Real Time' vs. 'Real Fast': How to Choose?" in *Ottawa Linux Symposium (July 2008), pp. v2*, 2008, pp. 57–65.

[2] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout, "It's Time for Low Latency," in *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, ser. HotOS'13. Berkeley, CA, USA: USENIX Association, 2011, pp. 11–11.

[3] Z. Cheng, Y. Li, and R. West, "Qduino: A Multithreaded Arduino System for Embedded Computing," in *Proceedings of the 36th IEEE Real-Time Systems Symposium*, 2015.

[4] 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.

[5] J. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," in *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, 1989.

[6] B. Sprunt, "Aperiodic Task Scheduling for Real-Time Systems, Ph.D. Dissertation, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburg, PA, August 1990."

[7] M. Danish, Y. Li, and R. West, "Virtual-CPU Scheduling in the Quest Operating System," in *Proceedings of the 17th Real-Time and Embedded Technology and Applications Symposium*, 2011.

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

[9] "lwIP: http://savannah.nongnu.org/projects/lwip/."

[10] G. Banga, P. Druschel, and J. C. Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," in *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, 1999.

[11] C. Mercer, S. Savage, and H. Tokuda, "Processor Capacity Reservation for Multimedia Operating Systems," in *IEEE International Conference on Multimedia Computing and Systems*. IEEE, May 1994.

[12] H. Simpson, "Four-slot Fully Asynchronous Communication Mechanism," *IEEE Computers and Digital Techniques*, vol. 137, pp. 17–30, January 1990.

[13] J. Rushby, "Model Checking Simpsons Four-slot Fully Asynchronous Communication Mechanism," *Computer Science Laboratory–SRI International, Tech. Rep. Issued*, 2002.

[14] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors," in *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*. IEEE, 2004, pp. 455–462.

[15] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying Event-driven Programming of Memory-constrained Embedded Systems," in *Proceedings of the 4th international conference on Embedded networked sensor systems*. Acm, 2006, pp. 29–42.

[16] E. Baccelli, O. Hahm, M. Gunes, M. Wahlisch, and T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*. IEEE, 2013, pp. 79–80.

[17] P. Gai, E. Bini, G. Lipari, M. Di Natale, and L. Abeni, "Architecture for a Portable Open Source Real Time Kernel Environment," in *Proceedings of the Second Real-Time Linux Workshop and Hands on Real-Time Linux Tutorial*, 2000.

[18] V. Apuzzo, A. Biondi, and G. Buttazzo, "OSEK-Like Kernel Support for Engine Control Applications under EDF Scheduling," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016, pp. 1–11.

[19] G. C. Buttazzo, E. Bini, and D. Buttle, "Rate-adaptive Tasks: Model, Analysis, and Design Issues," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2014, pp. 1–6.

[20] P. Buonocunto, A. Biondi, M. Pagani, M. Marinoni, and G. Buttazzo, "ARTE: Arduino Real-time Extension for Programming Multitasking Applications," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2016.

VIII. Appendix

A. The 10 kHz Pulse Train Experiment

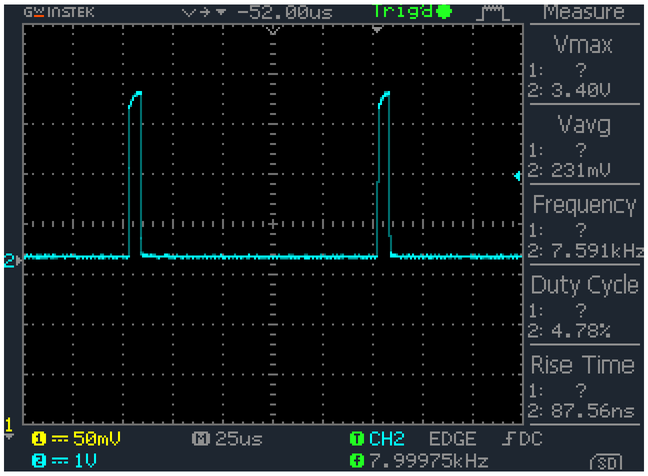

Fig. 10: Target 10kHz Pulse on Yocto Linux (Actually 7.591kHz)

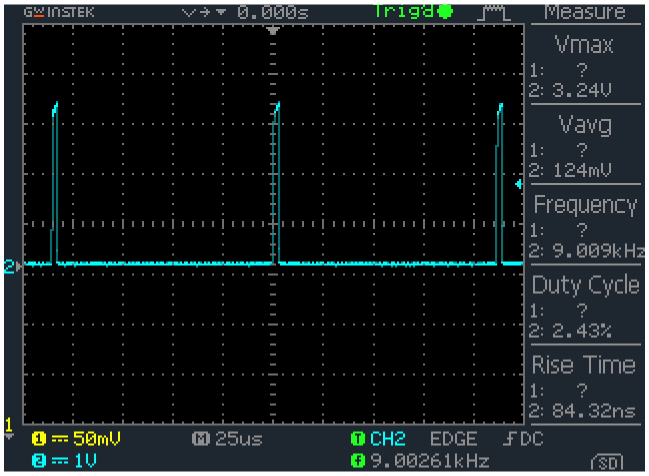

Fig. 11: Target 10kHz Pulse on Qduino (Actually 9.009kHz)

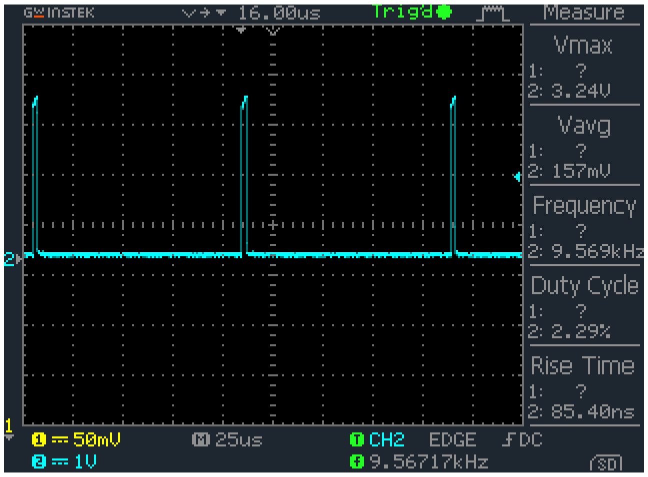

Fig. 12: Target 10kHz Pulse on QduinoMC (Actually 9.569kHz)

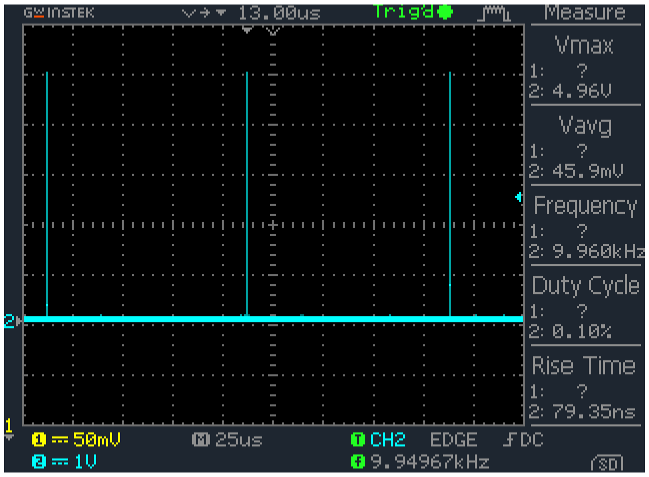

Fig. 13: 10kHz Pulse on the Printrboard (without a webserver)

B. Prototype 3D Printer Controller

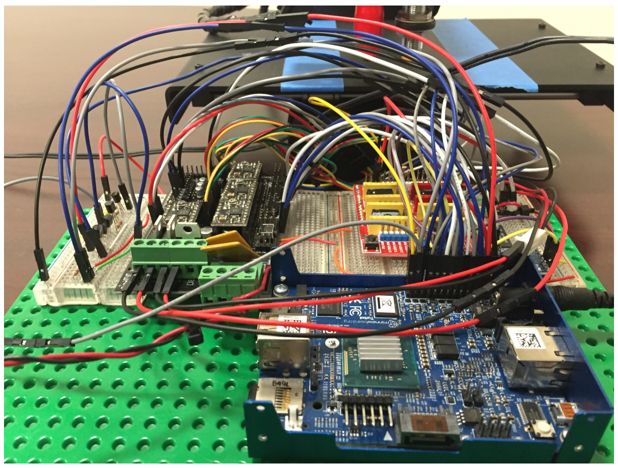

Fig. 14: MinnowMax-based Controller for the Printrbot

C. Example 3D Object

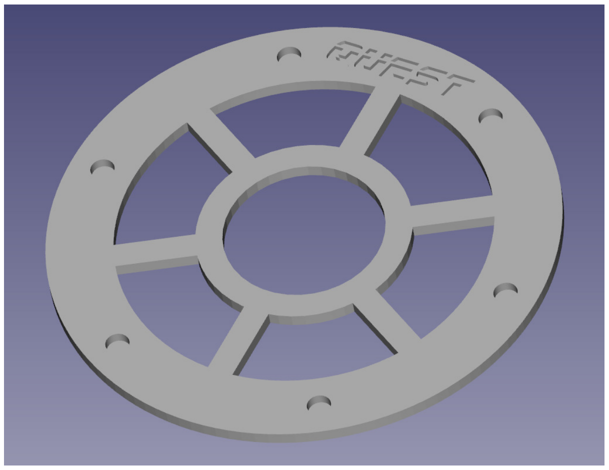

Fig. 15: 3D Image STL File for Test Object

D. Printed Test Objects

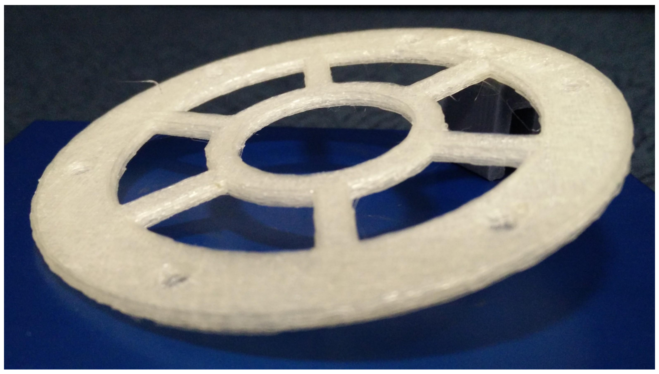
Fig. 16: Linux (More than twice the print time of QduinoMC)

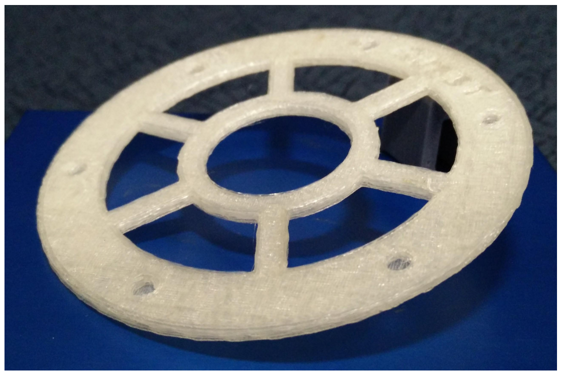
Fig. 17: QduinoMC (Faster and better quality than Linux)

E. MinnowMax-based 3D Printer Controller

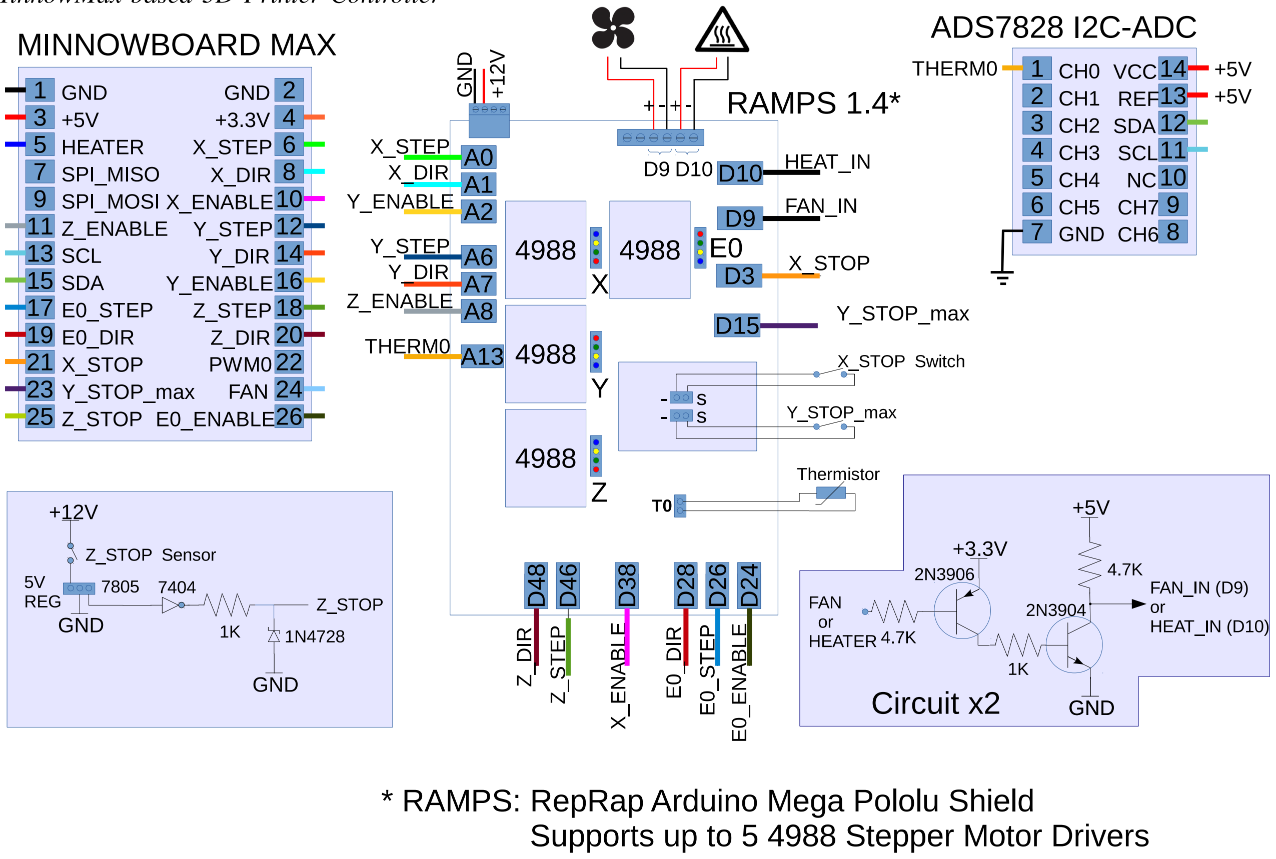

Fig. 18: 3D Printer Controller Circuit Diagram