Qduino: A Multithreaded Arduino System for Embedded Computing

Zhuoqun Cheng
Boston University
Computer Science Dept.
czq@cs.bu.edu

Ye Li
Boston University
Computer Science Dept.
liye@cs.bu.edu

Richard West
Boston University
Computer Science Dept.
richwest@cs.bu.edu

***Abstract*—Arduino is an open source platform that offers a clear and simple environment for physical computing. It is now widely used in modern robotics and Internet of Things (IoT) applications, due in part to its low-cost, ease of programming, and rapid prototyping capabilities. Sensors and actuators can easily be connected to the analog and digital I/O pins of an Arduino device, which features an on-board microcontroller programmed using the Arduino API. The increasing complexity of physical computing applications has now led to a series of Arduino-compatible devices with faster processors, increased flash storage, larger memories and more complicated I/O architectures. The Intel Galileo, for example, is designed to support the Arduino API on top of a Linux system, code-named Clanton. However, the standard API is restricted to the capabilities found on less powerful devices, lacking support for multithreaded programs, or specification of real-time requirements. In this paper, we present Qduino, a system developed for Arduino compatible boards. Qduino provides an extended Arduino API which, while backward-compatible with the original API, supports real-time multithreaded sketches and event handling. Experiments show the performance gains of Qduino compared to Clanton Linux.**

I. Introduction

Arduino [1] is a popular open-source platform for embedded computing. Its success is mainly due to the simplicity of its programming interface, the comprehensive library support, and the availability of numerous extension shields. Arduino offers a straightforward programming interface to create sketches (programs) that interact with the environment using sensors and actuators.

Traditionally, Arduino boards are equipped with the AVR ATmega microcontrollers, operating at speeds up to 20 MHz. The relatively low processing capabilities, limited SRAM and flash capacity, restricts traditional Arduino boards to applications with fairly simple logic and I/O capabilities. While these devices are adequate for basic sensing and control, they are insufficient for the high processing demands of many robotics or Internet of Things (IoT) applications, which now run digital image processing, location and 3D mapping algorithms. Consequently, many Arduino compatible boards with more powerful hardware specifications are now emerging. Examples include the Intel Galileo and Edison boards, the Minnowboard MAX, and the 86Duino, amongst others. These new boards have inherited the simplicity of the original Arduino API, and provide the standard Arduino GPIO hardware interface, which makes them compatible with most of the existing Arduino extension shields. However, they all feature a much more powerful processor and more complex I/O architecture. To cope with the complexity of the architecture, most of the advanced Arduino compatible platforms are shipped with an embedded Linux operating system and execute Arduino sketches as Linux processes.

Modern robotics and IoT applications often interact with complex I/O peripheral devices and require the handling of multiple threads of control. The standard Arduino API was designed for programs running directly on 8-bit AVR microcontrollers. It provides the interface to setup a single thread of execution with a `loop()` function. This is insufficient for use in multi-threaded applications, which require processing to continue while other threads wait on I/O operations. Additionally, single-threaded applications may under-utilize system resources and fail to take advantage of the parallelism provided by the hardware. For example, the Intel Edison board now has dual cores, and future devices may support significantly more cores. Hence, we believe that a preemptive multithreading interface should be added to the standard Arduino API.

Even though programming techniques such as *event loops* and *coroutines* can be used in a single-threaded environment to achieve cooperative scheduling, they only allow coarse-grained multithreading which often complicates the application design and cannot provide true parallelism on a multicore platform. To implement true preemptive multithreading, a process/thread model and a corresponding scheduling framework are required. However, because of the real-time nature of many embedded applications running on Arduino platforms, a multithreading interface for the Arduino API must also guarantee timing *predictability* of different control flows in a program sketch. To achieve this requires *temporal isolation* between threads, so that they do not interfere with one another's progress. Unfortunately, traditional operating systems designed for general applications fail to provide adequate predictability. As a result, even though a multithreading Arduino API extension can be trivially implemented under Linux with Pthreads, our experiments (shown in Section III) demonstrate the lack of predictable sketch behavior of the approach. This problem becomes even more obvious when asynchronous system events such as device interrupts have the capability to interfere with thread execution.

In this paper, we present Qduino, an operating system and programming environment that provides support for real-time, multithreading extensions to the Arduino API. Qduino is built on top of the Quest [2] real-time kernel, which runs on multicore x86 platforms and Arduino-compatible devices such as the Intel Galileo. The contributions of Qduino include:

- An extension to the standard Arduino API, which is easy to use and allows the creation of multithreaded sketches, as well as synchronization and communication between threads.

- Real-time features that provide temporal isolation between different threads and asynchronous system events such as device interrupts.
- An event handling framework that offers predictable event delivery for I/O handling in an Arduino sketch.
- A platform with smaller memory footprint and improved performance for Arduino sketches as compared to embedded computing platforms based on Linux.
- Backward compatibility that allows the execution of legacy Arduino sketches.

The rest of this paper is organized as follows: Section II describes the Qduino architecture. We introduce the basic kernel utilities of Quest and explain how standard Arduino APIs are implemented in Qduino, using the Intel Galileo as an example. This is then followed by a detailed discussion of the design and implementation of the proposed API extensions. In Section III, we evaluate the performance and effectiveness of the API extensions by comparing Qduino with the Clanton Linux distribution shipped with the Intel Galileo board. We show the situations under which Qduino outperforms Clanton Linux. Related work is then presented in Section IV, followed by conclusions and future work.

II. Qduino Architecture

Qduino is a predictable, multithreaded Arduino system built on our Quest real-time operating system [2]. Quest is a standalone system designed around three main goals: *safety*, *predictability*, and *efficiency*. It currently operates on 32-bit x86 architectures, and leverages hardware MMU support to provide page-based memory protection to processes and threads. As with UNIX-like systems, segmentation is used to separate the kernel from user-space. Quest is an SMP system, operating on multicore and multiprocessor platforms. It has support for kernel threads, POSIX threads, and a network protocol stack based on *lightweight IP* (lwIP) [3]. The source tree is more than 200 thousand lines of code, including drivers and lwIP. However, the core kernel code is approximately 11 thousand lines. The system features a novel hierarchical VCPU scheduling framework that ensures temporal isolation between system events (e.g., interrupts) and conventional tasks.

Porting the Quest kernel to an Arduino compatible device such as the Intel Galileo is relatively straight-forward. The Galileo features a 32-bit x86 Quark processor, and uses a customized GRUB bootloader that leverages the on-board EFI firmware. We had to modify the Galileo's legacy GRUB code to include multiboot support, which is required by Quest.

Arduino sketches in Qduino are executed as user processes in the Quest operating system. A Qduino sketch allows multiple `loop()` constructs to be declared, with each loop assigned to a separate thread in a single process. We developed the Qduino libraries for Quest user processes in order to support Arduino APIs on platforms such as the Galileo. The actual API implementations require various device drivers (including an I^2C bus controller driver, an SPI bus controller driver, and a GPIO controller driver) in the Quest kernel to control the Galileo hardware GPIO interface.

An overview of the Qduino system architecture is shown in Figure 1. Even though Qduino currently only supports the Intel Galileo board, we are looking into other x86 SoCs such as the Intel Edison module and the Minnowboard MAX.

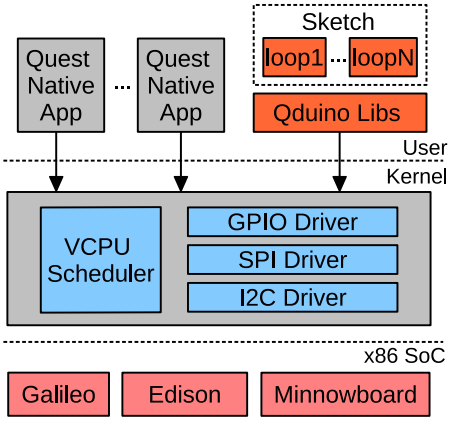

Fig. 1. Qduino Architecture Overview

As stated earlier, Quest features a novel VCPU scheduler that guarantees predictable behavior of system events and application tasks. This scheduling framework is essential to providing temporal isolation between multiple loops in an Arduino sketch, and predictable interrupt handling.

A. VCPU Scheduling Framework

For use in real-time systems, Quest must perform certain tasks by their deadlines. The system does not require tasks to specify deadlines but instead ensures that the execution of one task does not interfere with the timely execution of others. For example, Quest is capable of scheduling interrupt handlers as threads, so they do not unduly interfere with the execution of higher-priority tasks. While Quest's scheduling framework is described elsewhere [4], we briefly explain how it provides temporal isolation between tasks and system events. This is the basis for real-time tasks with specific resource requirements to be executed in bounded time, while allowing non-real-time tasks to execute with specific priorities.

In Quest, *virtual CPUs* (VCPUs) form the fundamental abstraction for scheduling and temporal isolation of the system. The concept of a VCPU is similar to that in virtual machines [5], [6], where a hypervisor provides the illusion of multiple *physical CPUs* (PCPUs) [1] represented as VCPUs to each of the guest virtual machines. VCPUs exist as kernel objects to simplify the management of resource budgets for potentially many software threads. Quest uses a hierarchical approach in which VCPUs are scheduled on PCPUs and threads are scheduled on VCPUs. Each VCPU acts as a resource container [7] for scheduling and accounting decisions on behalf of its assigned software threads.

In common with bandwidth preserving servers [8], [9], [10], each VCPU, V, has a maximum compute time budget, C_{max}, available in a time period, T_V. V is constrained to use no more than the fraction $U_V = \frac{C_{max}}{T_V}$ of a physical processor (PCPU) in any window of real-time, T_V, while running at its normal (foreground) priority. To avoid situations where PCPUs are idle when there are threads awaiting service, a

[1] We define a PCPU to be either a conventional CPU, a processing core, or a hardware thread.

VCPU that has expired its budget may operate at a lower (background) priority. All background priorities are set below those of foreground priorities to ensure VCPUs with expired budgets do not adversely affect those with available budgets.

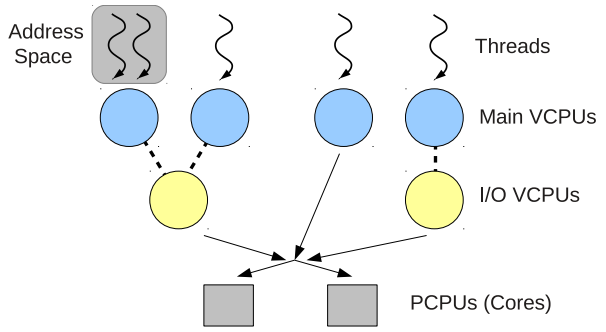

Fig. 2. VCPU Scheduling Hierarchy

Quest defines two classes of VCPUs as shown in Figure 2: (1) *Main VCPUs* are used to schedule and track the PCPU usage of conventional software threads, while (2) *I/O VCPUs* are used to account for, and schedule the execution of, interrupt handlers for I/O devices. This distinction allows for interrupts from I/O devices to be scheduled as threads, which may be deferred execution when threads associated with higher priority VCPUs having available budgets are runnable. Quest allows I/O VCPUs to be specified for certain devices, or for certain tasks that issue I/O requests, thereby allowing interrupts to be handled at different priorities and with different CPU shares than conventional tasks associated with Main VCPUs.

By default, each Main VCPU acts like a Sporadic Server [11], [12], with a budget and replenishment period. Each I/O VCPU, V_j, has a dynamically calculated budget and period, based on a specified utilization bound, U_j. An I/O VCPU's service constraints are a function of those of the Main VCPU bound to it, which is currently running a thread requiring I/O processing. In Quest, every I/O operation is associated with an accountable thread. This approach simplifies the budget management of I/O VCPUs, which have to deal with potentially many short-lived interrupt handlers whose execution times are far less than those of process timeslices.

Local APIC timers are programmed to replenish VCPU budgets as they are consumed during thread execution. Sporadic Servers enable a system to be treated as a collection of equivalent periodic tasks scheduled by a rate-monotonic scheduler (RMS) [13]. This is significant, given I/O events can occur at arbitrary (aperiodic) times, potentially triggering the wakeup of blocked tasks (again, at arbitrary times) having higher priority than those currently running. RMS analysis can be applied (See Equation 1 below), to ensure each VCPU is guaranteed its share of CPU time, U_V.

Temporal Isolation. In Quest, VCPUs are mapped to a separate scheduling queue for each PCPU. Under this arrangement, our default policies for Main and I/O VCPU scheduling allow us to guarantee temporal isolation if the Liu-Layland utilization bound is satisfied [13]. For a single PCPU with n Main VCPUs and m I/O VCPUs we have:

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

Here, C_i and T_i are the budget capacity and period of Main VCPU V_i, and U_j is the utilization factor of I/O VCPU V_j [4]. This bound can be improved with dynamic priority scheduling of VCPUs (e.g., using earliest deadline first scheduling) but this adds more overhead to the scheduler. This is because: (1) dynamic priorities require more complex queue management, and (2) Quest uses local APIC timers, programmed for one-shot operation, to trigger an interrupt in time for the next event to be processed; more frequent reprogramming of timers may be necessary if priorities change.

Quest admission control uses Equation 1 to decide whether to allow the creation of a new VCPU. In overload, static priority scheduling has the advantage that the highest priority subset of VCPUs capable of meeting their timing requirements will not be affected by lower priority VCPUs. This is not the case with dynamic priority scheduling, where overload can cause all VCPUs to fail to maintain their correct PCPU shares. Similarly, hypervisor scheduling using policies such as Borrowed Virtual Time (BVT) [14] cannot guarantee temporal isolation between VCPUs over specific real-time windows.

B. Arduino API Support

The Arduino language reference [15] specifies 40 functions and various libraries (e.g. WiFi, Servo, etc.) available for all Arduino-compatible platforms. Table I lists all the functions in different categories. On the traditional Arduino boards (e.g. UNO, Duemilanove), all the GPIO pins are connected directly to the microcontroller. To implement the Arduino digital I/O APIs, the software just needs to write 0s and 1s to certain memory registers. PWM output (`analogWrite()`) is emulated using digital I/O with the help of a hardware timer, and an ADC in the microcontroller can be used to support `analogRead()`.

TABLE I. ARDUINO STANDARD API

Function Name	Category
`loop, setup`	Structure
`pinMode, digitalWrite, digitalRead`	Digital I/O
`analogWrite, analogRead, analogReference`	Analog I/O
`tone, noTone, shiftOut, shiftIn, pulseIn`	Advanced I/O
`millis, micros, delay, delayMicroseconds`	Time
`min, max, abs, constrain, map, pow, sqrt`	Math
`sin, cos, tan`	Trigonometry
`randomSeed, random`	Random Numbers
`lowByte, highByte, bitRead, bitWrite, bitSet, bitClear, bit`	Bits and Bytes
`attachInterrupt, detachInterrupt`	External Interrupts
`interrupts, noInterrupt`	Interrupts

On the Intel Galileo board, however, the GPIO fabric is mostly controlled by a GPIO Expander featuring the Cypress CY8C9540A chipset connected to the I^2C bus. The I^2C bus controller itself is a PCI device that can be probed and programmed by the processor. Similarly, the Galileo board also

features an AD7298 ADC device connected to an SPI bus controller on the PCI bus to support analog input. Consequently, to support the Arduino APIs on the Galileo board, we developed drivers for the PCI bus, I^2C bus controller, the Cypress GPIO Expander chipset, the SPI bus controller, and the ADC. The Galileo board I/O infrastructure overview is shown in Figure 3.

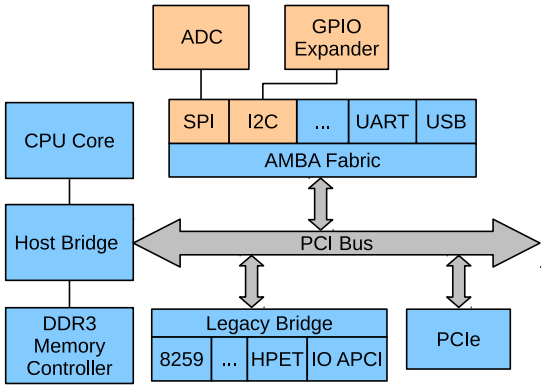

Fig. 3. Intel Galileo I/O Infrastructure Overview

In Clanton Linux, Intel and other device vendors provide all the Linux drivers needed and all the GPIO pins have been exposed to user-space through the Linux `sysfs` interface. All the Arduino API functions are included in the Arduino IDE as shared libraries, and interface with the Linux `sysfs` for GPIO operations. Arduino sketches are then converted into Linux user processes for execution.

In Qduino, we developed all the device drivers in the Quest kernel and exposed the GPIO interface to sketches running as user processes through system calls. Currently, we have implemented most of the frequently used functions along with the Serial and Servo libraries with the exception of some advanced I/O interfaces. Most of the API implementations are wrappers around the GPIO system calls and are included in a user library called `libqduino`. Similar to Clanton Linux, an Arduino sketch for Qduino will be pre-processed and turned into a C program, and then linked with `libqduino`. In this way, a sketch can be converted into a normal Quest user process and loaded for execution. Experiments in Section III show that the standard Arduino API implemented in Qduino outperforms the one for Clanton Linux.

C. Arduino API Extension

In this section, we discuss the Arduino API extensions in Qduino, to support real-time, multi-threaded applications.

Multithreaded Sketch. The standard Arduino API offers two structure functions: `setup()` and `loop()`. The `setup()` function is called when a sketch starts and usually contains code for initialization. After calling the `setup()` function, the `loop()` function repeatedly performs a series of tasks. While only one `loop()` function is allowed in the standard API, Qduino allows up to 32 `loop()` functions. Each `loop()` function is assigned to a Quest thread and scheduled by the Quest scheduler.

Multiple loop support in Qduino makes it easier to write sketches with parallel tasks. A simple example might be to process sensor input data from one I/O pin while another I/O pin is used for output, perhaps to control an actuator. If the input and output processing require separate rates for reading and writing data, a single timed loop might be inadequate. The loop will have a certain period, which might satisfy one, but not necessarily both, of the input and output rates. A similar example is shown for blinking LEDs in the standard Arduino API [16], suggesting users to do time accounting on their own. This places the burden of scheduling on users, making code overly complex and vulnerable to mistakes when the number of tasks increases. With the multi-loop feature, separate tasks with different delays between I/O operations can be assigned to different loops, with the assurance that their delay settings will not affect other tasks.

Furthermore, binding loops to threads also provides the possibility to interleave the execution of independent tasks. This is particularly beneficial when blocking I/O operations frequently occur. In a single-threaded sketch, a blocking I/O operation in one task would prevent other tasks in the same sketch from executing. In Section III, we experiment with a multithreaded sketch which has both CPU- and I/O-intensive tasks. Significant performance improvement is observed over a single-threaded sketch containing the same tasks.

Communication and Synchronization. One benefit of binding loops to threads rather than processes is that communication is vastly simplified. Communication between loops, on Qduino, can be done via global variables, which are automatically shared by all the loops within one sketch. However, unrestricted use of shared variables is unreliable and unsafe due to multiple update problems. Therefore, spinlocks are made available for use in Qduino. To hide the complexity of explicit synchronization and to maintain the simplicity of Arduino programming, we further provide two asynchronous communication facilities: a four-slot [17] channel and a ring buffer. Simpson's four-slot fully asynchronous communication mechanism allows a single reader and writer to access a shared memory region in such a way that the reader always accesses the most recent data stored by the writer, and neither entity need wait for the other [18]. Thus, data is always *fresh*, even though some may be over-written and, hence, lost. Four-slot asynchronous communication is widely used in real-time systems to guarantee that actuators always read the latest data from sensors. We also provide a single-reader, single-writer ring buffer FIFO for applications that want historical data values to be preserved.

Temporal Isolation. Each thread bound to a loop in Qduino is associated with a separate VCPU. As explained in the Section II-A, Quest partitions CPU resources precisely between tasks and thereby ensures temporal isolation between them. By making use of these properties provided by Quest, Qduino guarantees that the execution of one loop will not interfere with the timely execution of others.

As also mentioned earlier, Quest is capable of scheduling interrupt handlers as time-budgeted threads, to avoid interference with other tasks. We exploit this feature by creating an I/O VCPU to handle interrupt bottom halves associated with the GPIO expander. The I/O VCPU budget prevents a high volume of interrupts being handled indefinitely, at the cost of other tasks. By careful tuning of I/O and Main VCPU budgets, it is possible for a system designer to balance CPU time between CPU- and I/O-intensive tasks. Effectively, when setting the I/O

VCPU capacity to 0, GPIO interrupt handling is disabled. It is possible in Qduino to establish separate I/O VCPUs for different devices (or GPIOs), depending on the underlying hardware. For situations where the system is configured to only have one I/O VCPU for all devices, there is a many-to-one mapping of Main VCPUs (one for each Qduino loop thread) to the I/O VCPU.

Although this paper focuses on single sketches with multiple threads, we are considering the support for separate co-existent sketches in different processes. One idea is to enable process-level control of interrupt delivery by allowing a sketch's Main VCPU(s) to be unbound from an I/O VCPU. Similarly, when a sketch wants to unblock interrupt delivery, its Main VCPU(s) can be rebound to the I/O VCPU. This way, the I/O VCPU budget (possibly set by a system administrator) can be made available to other sketches that still wish to receive interrupts. In Clanton Linux, it is not possible for sketches, which run in user-space processes, to disable interrupts, as this could affect the entire system.

In Section III, we show a 3-loop sketch to demonstrate that no loop experiences interference from other loops when all of them are performing CPU-intensive tasks. We also show that, while the number of interrupts are effectively controlled by adjusting the I/O VCPU parameters, the performance of a co-existing CPU-intensive loop is always isolated from interrupts.

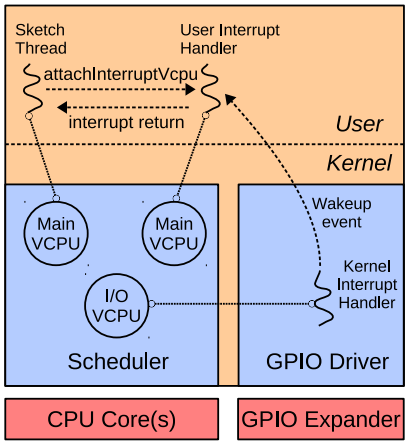

Fig. 4. Qduino GPIO Interrupt Handling Mechanism

Predictable Events. On the Galileo, there is currently one I/O VCPU for all GPIOs. When an interrupt is raised on a GPIO pin, the *top half* (non-deferrable part) of the GPIO interrupt handler will wakeup a thread associated with the I/O VCPU. The I/O VCPU is removed from a wait queue and added to a ready queue where it can be scheduled. When granted execution, the I/O VCPU thread runs at kernel-level and serves as the *bottom half* (deferrable handler) for GPIO interrupts. The kernel bottom half associated with the I/O VCPU queries the GPIO pin number that triggered the interrupt. This information can then be used to invoke a specific *user-level* interrupt handler in a Qduino sketch.

Qduino provides an `attachInterruptVcpu()` function, to associate an interrupt handler with a user-level thread that is bound to a time-budgeted Main VCPU. A user-level handler becomes eligible for execution when its Main VCPU (with non-zero budget) is moved to the ready queue by a wakeup event from the bottom half kernel thread. Figure 4 illustrates the GPIO interrupt handling mechanism in Qduino.

On Clanton Linux, a GPIO interrupt is delivered to a user-level process as a POSIX signal. There is no guaranteed delay between the occurrence of the GPIO pin change and the execution of the user-level handler, since it depends on when the process is scheduled. By comparison, Qduino ensures that the time interval between the reception of the hardware interrupt and the invocation of user-level handler is bounded by its worst-case delay (*WCD*).

The WCD happens when the bottom half is invoked at the moment when the associated I/O VCPU has just depleted its budget. Let C_{io} and T_{io} denote I/O VCPU's budget and period. According to Quest's I/O VCPU scheduling algorithm, it takes $T_{io} - C_{io}$ time until the I/O VCPU is replenished and is able to run the thread that issues a wakeup event. The wakeup event could be delivered to the Main VCPU of a user-level ISR at the critical instant when it, too, has just depleted its budget. The worst-case delay for the Main VCPU, V_h to resume execution is $T_h - C_h$, where C_h and T_h are the budget and period, respectively. Finally, the WCD has to consider the time to execute the bottom half, which can be obtained by pre-profiling. Let δ_{bh} denote the required CPU time of the bottom half, and Δ_{bh} denote the wall-clock time to execute the bottom half. We then have:

$$\Delta_{bh} = (T_{io} - C_{io}) + \left\lceil \frac{\delta_{bh}}{C_{io}} - 1 \right\rceil \cdot T_{io} + \delta_{bh} \bmod C_{io} \quad (2)$$

It follows that the WCD is:

$$\Delta_{WCD} = (T_h - C_h) + \Delta_{bh} = (T_h - C_h) + (T_{io} - C_{io}) + \left\lceil \frac{\delta_{bh}}{C_{io}} - 1 \right\rceil \cdot T_{io} + \delta_{bh} \bmod C_{io} \quad (3)$$

New APIs. If a new Arduino API is to be adopted by the community it must not require the modification of existing sketches and it must maintain the simplicity that made the original API so successful. Qduino maintains backward compatibility with the original API, while introducing a set of new functions as described in Table II. Values of C and T are, by default, specified in milliseconds, although Qduino can be configured to accept their specification in different time units.

TABLE II. NEW APIS

Function Signatures	Category
`loop(loop_id, C, T)`	Structure
`interruptsVcpu(C, T)`, `attachInterruptVcpu(pin, ISR, mode, C, T)`	Interrupt
`spinlockInit(lock)`, `spinlockLock(lock)`, `spinlockUnlock(lock)`	Spinlock
`channelWrite(channel, item)`, `item channelRead(channel)`	Four-slot
`ringbufInit(buffer, size)`, `ringbufWrite(buffer, item)`, `ringbufRead(buffer, item)`	Ring buffer

Qduino requires real-time loops to be specified with loop identifiers and VCPU parameters. For backward compatibility, Qduino also supports the standard `loop()` function. `attachInterruptVcpu()` extends the standard `attachInterrupt()` function by requiring the specification of Main VCPU timing constraints for a user-level

ISR. `interruptsVcpu()` is the API to control the I/O VCPU associated with the kernel thread serving as the bottom half of a GPIO interrupt. Though not listed in Table II, `noInterrupts()` and `interrupts()` disable and re-enable interrupts, respectively. These two functions are currently implemented as wrappers around the `interruptsVcpu()` function. `noInterrupts()` sets the I/O VCPU budget to zero so that the kernel thread dedicated to a bottom half is never executed. Finally, `interrupts()` simply restores the I/O VCPU budget cleared by `noInterrupts()`. In a future multi-sketch system, we plan to enable and disable interrupt delivery to individual sketches by binding/unbinding Main and I/O VCPUs as described earlier.

Example Sketch for Autonomous Vehicle. Listing 1 (in the Appendix) presents a sample sketch written with Qduino's new API. It is for a rover equipped with an HC-SR04 ultrasonic sensor and the Intel Galileo board. The sketch contains two loops: (1) a sensing loop detects the rover's distance to an obstacle, and (2) an actuation loop controls the motors. The sensing loop communicates the measured distance to the actuation loop via a four-slot channel. If a distance less than 1 meter is detected, the rover will back off and turn right to avoid a collision. This sample sketch only serves as a proof-of-concept. A more realistic autonomous vehicle, however, might be equipped with many more sensors and actuators. For example, a vehicle might use rotary encoders to measure speed, a PID control to stabilize movement, LIDAR sensors to compute localization and mapping values, and other audio-visual sensors to warn of potential collisions. Each task can be arranged into separate loops or interrupt handlers with appropriate VCPUs.

In this example, a four-slot communication channel is not entirely necessary, if sensor data is stored in global variables accessible to both loop threads. However, if sensor data is larger than the architecture word size (e.g., 64-bits on a 32-bit architecture) multiple memory fetches might see inconsistent updates to the values without using explicit synchronization. Such synchronization could unduly affect the timing of both sensing and actuation loops, which can otherwise proceed independently using four-slot communication.

III. Experimental Evaluation

We conducted a series of experiments to investigate the performance of the standard Arduino API and the effectiveness of our API extensions in the Qduino environment. All experiments used a first generation Intel Galileo board with GPIO logic level set to 3.3V [2]. We compared Qduino to Clanton Linux 3.8.7, which is shipped with the Intel Galileo board. The Linux sketches are created and uploaded with the Intel Arduino IDE v1.0.0. Sketches running on Qduino are built using Quest's toolchain and loaded through the Qduino shell. Quest's toolchain is based on GCC 4.7.2, with the same optimization flag (`-Os`) as the Intel Arduino IDE. All clock cycle timing measurements used the Quark processor's TimeStamp Counter.

[2]This is achieved by modifying the `IOREF` jumper on the board. The default logic level is set to 5V.

The Standard Arduino I/O API. To evaluate the efficiency of the standard Arduino API implementation in Qduino, we compared the performance of `digitalWrite()`, `digitalRead()`, and the maximum interrupt frequency with `attachInterrupt()` between Clanton Linux and Qduino. For the `digitalWrite()`, we toggled digital pin 13 for 4000 times, while for the `digitalRead()`, we read the value on pin 13 without delay for 4000 times. We tested both functions on Clanton Linux and Qduino and recorded the average CPU cycles needed to perform a single operation. The results shown in Figure 5 demonstrate that our implementation of basic GPIO operations in Qduino does not incur any additional overhead compared to Clanton Linux. Moreover, the `digitalWrite()` in Qduino is more efficient than in Clanton Linux.

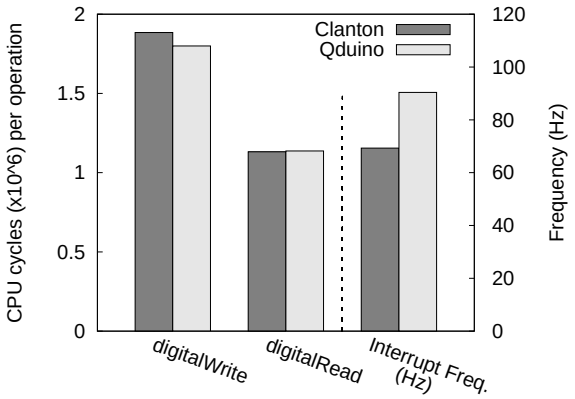

Fig. 5. Arduino API Performance Comparison

In the next experiment, we wrote a sketch that registered an interrupt service routine (ISR) for a pin change event on digital pin 2, using `attachInterrupt()`, and then toggled the pin setting 4000 times. By tuning the delay between each pin change (using `digitalWrite()`), we recorded the minimum delay that guarantees the reception of all the interrupts. From this, we calculated the corresponding maximum interrupt frequency for both Linux and Qduino. The results are also shown in Figure 5. We observed that Qduino is able to handle a higher rate of interrupts via its `attachInterrupt()` implementation.

We next used an oscilloscope to test the effectiveness of analog I/O in Qduino. Figure 6 is a screenshot of `analogWrite(pin,120)` running on Clanton Linux and Qduino. The information in the right column shows that both platforms have almost identical maximum and average voltage, and the same frequency and calculated duty cycles.

Multithreaded Sketch. We next constructed a sketch with a mixture of CPU and I/O operations. For the CPU workload, we constructed a `findPrime` benchmark that calculates all prime numbers smaller than 80000. For I/O, we issued 2000 `digitalWrite()` requests. In the single-loop version, both CPU and I/O operations are combined in one `loop()` function. In the multithreaded version, we have two loops: one runs `findPrime` and the other issues the `digitalWrite()` requests. Table III lists all four experimental cases. We conducted this group of experiments on both Qduino and Clanton Linux. For Qduino, the single-loop case uses a Main VCPU with $C = 498mS$ and $T = 500mS$ [3]. In the multi-loop version, we

[3]Unless stated otherwise, all VCPU parameters are in milliseconds.

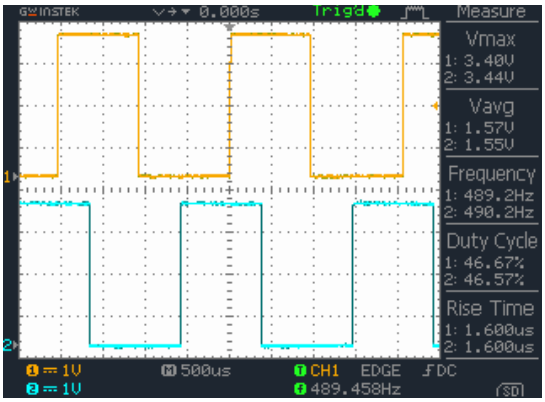

Fig. 6. analogWrite Performance Comparison

TABLE III. CASE DESCRIPTIONS

Case #	Description
Case 1	single-loop `digitalWrite()`
Case 2	single-loop findPrime
Case 3	single-loop `digitalWrite()`+findPrime
Case 4	multi-loop `digitalWrite()`+findPrime

established a Main VCPU with $C = 495mS$ and $T = 500mS$ to run `findPrime`. For the I/O operations, we assigned an I/O VCPU with $3/500$ fraction of CPU time. In both cases, the leftover CPU time is reserved for the shell so that the sketch can be loaded. When running on Clanton, the multithreaded sketch uses the Pthread library.

Results in Figure 7 show that the multithreaded sketch achieves approximately 28% performance increase over the single-loop version on Clanton Linux, and 31% increase over a single-loop version on Qduino. The multithreaded sketches are both only slightly slower than running findPrime alone. This is because `digitalWrite()` spends most of its time blocking on I/O commands from the I^2C bus.

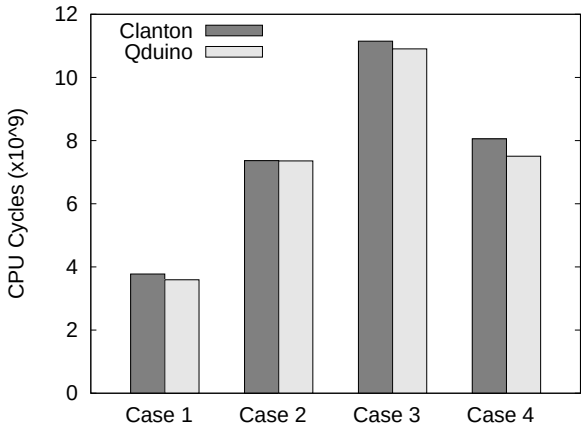

Fig. 7. Multithreaded Sketch Benchmarks

Predictability. We conducted two groups of experiments to verify the predictability of loop execution and event delivery in Qduino. In the first group, we wrote a series of multi-loop sketches, in which one foreground loop per sketch repeatedly increments a counter. At the end of the foreground loop's period the counter value is recorded and then reset to 0. Additionally, two or more background loops per sketch serve as potential interference sources by performing CPU-intensive tasks. Table IV shows the VCPU parameters for each foreground loop, and the number of background loops in each case. The background loops in each case equally consume all remaining CPU capacity not used by the foreground loop. The lack of hardware performance counters on the Quark processor meant that we periodically sampled a counter value to track each foreground loop's progress in real-time.

TABLE IV. EXPERIMENTAL SETUP

Case #	VCPU Parameters	# of Background Loops
Case 1	50/100	2
Case 2	50/100	4
Case 3	70/100	2
Case 4	70/100	4
Case 5	90/100	2
Case 6	90/100	4

For comparison, the same experiment was run on Clanton Linux using a separate POSIX thread within a single process for each of the loops. The standard `loop()` function runs in a separate thread and does the same work as the foreground loop in Qduino's setup. Clanton's lack of real-time support meant that the threads were not able to be specified with time constraints. Consequently, we only varied the number of background threads according to the cases in Table IV.

Figure 8 shows the value of the foreground loop counter in each case (with 2 or 4 background threads). The counter is incremented to about the same value in every period for a given VCPU constraint using Qduino. This is due to the guaranteed execution time of the loop within each VCPU period. However, due to the lack of predictability in the Linux scheduler, the progress of the `loop()` function is variable, as seen by the spikes in counter values above and below the average. It is also sensitive to the number of background threads, which are not temporally isolated from the foreground loop.

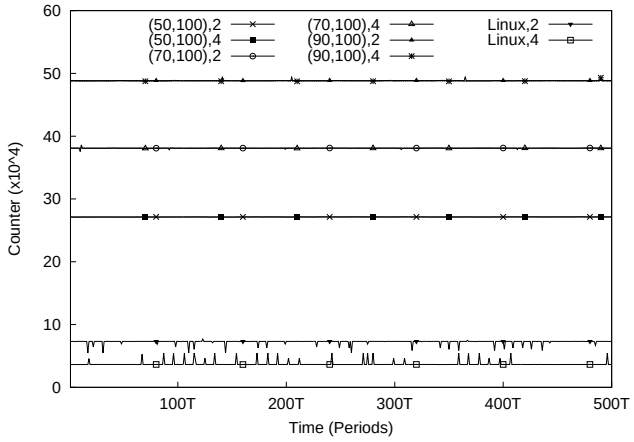

Fig. 8. Predictable Loop Execution

In the second group of experiments, we tested the predictability of Qduino's event delivery framework. We used two Intel Galileo boards. Board A's pin 13 was connected to Board B's pin 2. Board A ran Clanton Linux and flipped pin 13 in fast mode with a random delay, ranging from 0 to 2.3 milliseconds. It thus generated interrupts on Board B's pin 2 with a variable frequency from 477kHz to approximately 434Hz [4]. On Board

[4]Interrupt rate is: $1000/(1000/f + \delta) \mid \delta = [0, 2.3ms], f = 477KHz$ in fast mode.

B, we ran a sketch that attaches an interrupt handler to pin 2 using the `attachInterruptVcpu()` function with different VCPU parameters in each case. We also, in each case, adjusted the I/O VCPU parameters using the `interruptsVcpu()` function. The parameter combinations of the I/O VCPU bound to the bottom-half kernel thread, and the Main VCPU associated with the user-level interrupt handler, are shown in Table V. We instrumented the Quest kernel to measure the

TABLE V. VCPU PARAMETERS

Case #	I/O VCPU	Main VCPU
Case 1	2/10	3/10
Case 2	1/10	3/10
Case 3	3/10	3/10
Case 4	2/10	2/10
Case 5	2/10	3/20

event delivery time, which is the time interval between the invocation of the top half and the invocation of the user-level interrupt handler. We also measured the execution time of the bottom half to be 2.33 milliseconds without any interruption from external interrupts or CPU scheduling. We calculated the predicted worst-case delay (WCD) for event delivery using Equation 3 in Section II-C. Figure 9 compares the predicted WCD's with the observed event delivery times under different VCPU combinations. As can be seen, the observed value is always within the prediction bounds.

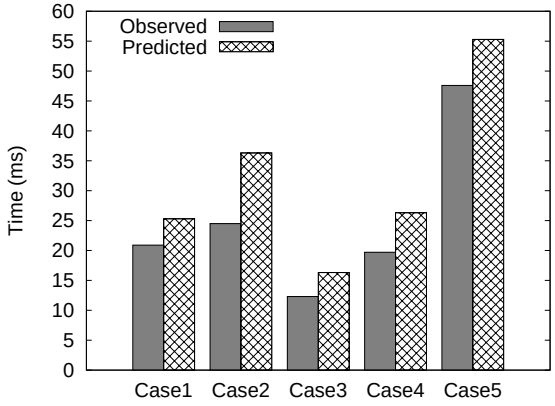

Fig. 9. Event Delivery Time

Temporal Isolation. Loops in Qduino sketches are guaranteed to be temporally isolated from other loops and asynchronous system events, e.g. interrupts. We conducted another set of experiments to verify temporal isolation.

We first wrote a sketch with 3 loops, each running `findPrime` with different VCPU parameters, as shown in Table VI. We then split the 3-loop sketch into three single-loop

TABLE VI. VCPU PARAMETERS

Loop #	Loop 1	Loop 2	Loop 3
VCPU parameters	40/100	20/100	10/100

sketches. Each contains one of the three loops respectively. We ran each single-loop sketch with the same VCPU parameters it used in the 3-loop version. We compared each loop's execution time in the 3-loop sketch to that in the corresponding single-loop sketch (averaged over 5 runs). The results in Figure 10 show that Qduino maintains temporal isolation between loops.

In a further experiment, we investigated the use of I/O VCPUs in Qduino. We used a similar setup to the predictable

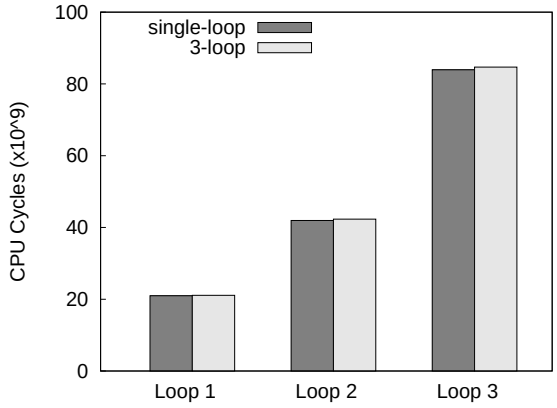

Fig. 10. Temporal Isolation between Loops

event delivery experiment, except that Board A toggles pin 13 repeatedly, without any delay, and interrupts generated on B's pin 2 thus have a frequency of 220Hz. On Board B, we ran `findPrime` using a Main VCPU with parameters $70mS/100mS$. The sketch also attaches an interrupt handler to pin 2, which counts the number of interrupts received during the execution of `findPrime`. Five cases were studied, all using Qduino, as shown in Table VII. I/O VCPU parameters are adjusted via the `interruptsVcpu()` function before the interrupt handler is attached. Case 1 serves as the base case to show the execution time when external interrupts are disabled[5]. Case 5 is a special case, using a kernel-level interrupt handler, which is not associated with any I/O VCPU. This case is intended to show the interrupt processing interference when an I/O VCPU is not used. Results are presented in

TABLE VII. EXPERIMENTAL SETUP

Case #	I/O VCPU	External Interrupts
Case 1	10/100	OFF
Case 2	0/100	ON
Case 3	5/100	ON
Case 4	10/100	ON
Case 5	Disabled	ON

Figure 11. For each cluster of bars, the bar on the left shows the execution time of the loop in CPU cycles. As can be seen, when the I/O VCPU is enabled, the loop has approximately the same execution time with the base case where external interrupts are turned off, thereby demonstrating the expected temporal isolation between loops and interrupts. The bar on the right represents the number of interrupts received. Though the loop has guaranteed execution time whatever I/O VCPU parameters are used, the number of interrupts received varies accordingly. The larger the I/O VCPU budget is, the more interrupts the sketch receives. It demonstrates that interrupts can be flexibly controlled by the budget of the I/O VCPU. Users can effectively disable external interrupt delivery by setting the I/O VCPU budget to 0, as shown in Case 2.

We performed the same experiment with Clanton Linux. In this case, `findPrime`'s performance degrades to 30.4% of its peak value while 2402 interrupts are received. For comparison, we divided the CPU cycles between the Main VCPU and the I/O VCPU in Qduino to achieve the similar performance drop for `findPrime`. We found that when using a $40mS/100mS$ I/O VCPU and a $48mS/100mS$ Main VCPU, 5369 interrupts are received when the performance drop is about 34%.

[5]We disabled interrupts by removing the wire that connects the two boards.

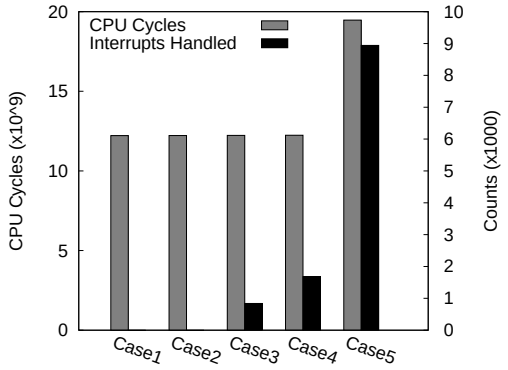

Fig. 11. Temporal Isolation between Loops and Interrupts

Autonomous Vehicle Application. Apart from microbenchmarks, we also created a simple collision avoidance application for an autonomous vehicle, as described earlier at the end of Section II-C. The sketch code for Qduino is shown in Listing 1 while the single-loop Clanton version is shown in Listing 2. We measured the time interval between two consecutive calls to the motor actuation code when there was a change in the distance to an obstacle as observed by the sensing logic. Figure 12 shows that with the multi-loop sketch in Qduino the time interval is stable at about 103ms. This includes an explicit 100ms to keep the motor settings at their current values, plus several digital I/O operations to subsequently change the motor values. In Clanton, the time interval varies from 383ms to 591ms. The Linux delay is a combination of the same 100ms programmed delay we used in Qduino plus the time to do one iteration of the sensing and actuation code.

Note that in both Linux and Qduino there is a 200ms sampling delay in the sensing code, to avoid the ultrasonic trigger pulse being incorrectly detected as an echo signal. However, in Linux this delay is included in a single loop for both sensing and actuation. Although Pthreads could be used to separate the sensing and actuation code, Clanton does not allow multiple threads to simultaneously access the I/O subsystem, because it serializes access to `sysfs`. In Clanton, the digital I/O pins on the Arduino are exposed to user-space code via the `sysfs` filesystem interface. Consequently, when the safe distance to an obstacle is set to 1m, an autonomous vehicle running Qduino can move at about 9.7m/s (>21 mph), while still having time to react before a collision. This compares to only 1.7m/s (3.8 mph) using a Clanton single-loop sketch.

For completeness, we include results in Figure 12 for a single-loop sketch in Qduino, which shows similar jitter to the same sketch running in Clanton Linux. However, Qduino has slightly less overhead because digital I/O is more efficient. The I/O operations in Qduino are system calls as opposed to being built on top of a filesystem abstraction in Linux. Finally, we include results for a Clanton sketch that uses an interrupt handler to time the ultrasonic signal within each iteration of the main motor-controlling loop. We also removed the `delay(200)` instruction at the cost of false echoes, to minimize the time spent in the sensing code. Even so, a Clanton sketch with interrupts still incurs more overhead than a multi-loop Qduino sketch. With multiple loops, the actuation code is not serialized with the sensing code.

Finally, we measured the memory footprint of the sketches and kernels on both platforms. Table VIII shows that Qduino has a smaller sketch and kernel memory footprint than Clanton. We believe that this is fairly beneficial to embedded applications running under relatively strict memory constraints.

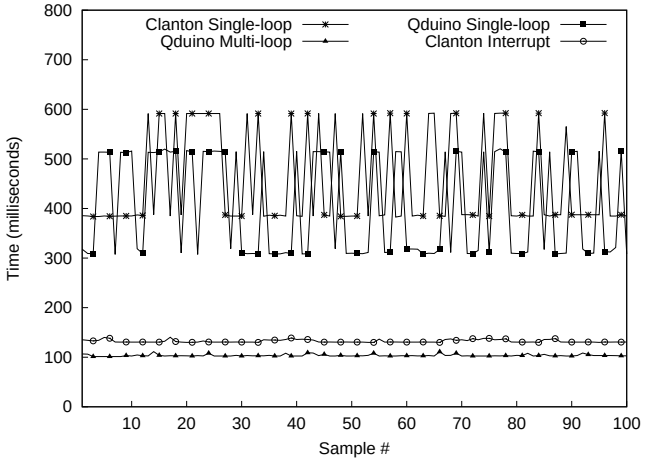

Fig. 12. Time between consecutive motor actuations

TABLE VIII. MEMORY FOOTPRINT

	Text (byte)	Data (byte)
Qduino kernel	953358	321516
Clanton kernel	4390436	336104
Qduino rover sketch	4832	2360
Clanton rover sketch	26249	27652

IV. RELATED WORK

Contiki [19] is a small footprint operating system for use with Internet of Things (IoT) devices. It supports per-process preemptive multi-threading by linking applications with a *protothread* library. Protothreads [20] function as stackless, lightweight threads and are cooperatively scheduled. This means that any protothread that fails to yield control back to the kernel will inevitably lock up the system. RIOT OS [21] is another multi-threaded operating system designed for IoT devices. RIOT enforces constant periods for kernel tasks to fulfill strong real-time requirements, but user-level threads are scheduled by a minimized scheduler without real-time guarantees. Both Contiki and RIOT aim to bridge the gap between OSes for wireless sensor networks and traditional fully-fledged OSes. However, Qduino is a system that focuses more on physical computing with hard real-time requirements.

The Arduino Yun [22] is a hardware approach to real-time and multi-threaded computing. Yun has an ATmega32u4 microcontroller for Arduino sketches requiring real-time performance, and an Atheros AR9331 SoC running a Linux based OS for more complex multi-threaded applications. A bridging library is required for communication between applications on the two chips. In contrast, Qduino makes it possible to create Arduino sketches with both real-time and multi-threading support on a single SoC. The communication between tasks is much more efficient and the programming interface is cleaner.

RT-Arduino [23] is a software-based Arduino extension that provides real-time multitasking support. It is built upon the OSEK/VDX certified ERIKA Enterprise RTOS [24]. Arduino loops are mapped to OSEK-tasks that are statically configured at compile-time. By comparison, Qduino provides the basis for Arduino sketches with multiple loops and interrupt handlers

to be associated with multi-threaded processes. This approach makes it possible to support real-time and parallel thread execution on multicore architectures.

Qduino is built on the assumption that the underlying OS provides temporal isolation between threads. Linux now has support for a `SCHED_DEADLINE` real-time class, with CPU reservations [9], [25], [26] for tasks based on the Constant Bandwidth Server [8]. This is similar to resource reserves in Linux/RK [27]. Qduino uses Quest's VCPU scheduling framework, which provides temporal isolation between both tasks and system events, such as interrupts. Quest uses a novel approach to dynamically calculate the budgets for handling short-lived and highly-frequent interrupts. These types of system events have been shown to severely fragment the replenishment lists of other types of bandwidth preserving scheduling algorithms, making their effective CPU utilizations lower than desired [4]. Consequently, Quest's scheduling framework is ideally suited to support real-time CPU and I/O processing [28] in Qduino.

V. Conclusion and Future Work

In this paper, we describe Qduino, an extension to the Arduino API for the Quest real-time operating system. Qduino is designed for Arduino-compatible devices with greater capabilities than those based on the Atmel MegaAVR. Qduino leverages Quest's VCPU scheduling approach to provide processor reservations over specific windows of time, for both tasks and I/O events. It also provides support for multithreading by allowing Arduino sketches to specify multiple loops, each with their own timing requirements. Experiments show that Qduino has similar performance efficiency to a Clanton Linux implementation on the Intel Galileo. However, it is shown to offer greater predictability and temporal isolation between separate threads. In particular, I/O operations and main CPU processing are guaranteed to not interfere with one another in Qduino, but this is not ensured in Linux. Multi-threaded real-time programming is made simple with the Qduino API.

Qduino has a smaller memory footprint than Clanton Linux, making it suitable for embedded real-time systems in areas such as robotics and the Internet of Things. Future work will investigate the predictability and efficiency of Qduino on multicore processors and networks of embedded devices.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1117025. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Arduino Homepage: http://arduino.cc.

[2] "Quest Operating System: http://www.questos.org."

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

[4] 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, pp. 169–179.

[5] K. Adams and O. Agesen, "A Comparison of Software and Hardware Techniques for x86 Virtualization," in *Proceedings of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 2–13.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003, pp. 164–177.

[7] 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.

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

[9] Z. Deng, J. W. S. Liu, and J. Sun, "A Scheme for Scheduling Hard Real-Time Applications in Open System Environment," in *Proceedings of the 9th Euromicro Workshop on Real-Time Systems*, 1997.

[10] M. Spuri and G. Buttazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems," *Real-Time Systems*, vol. 10, pp. 179–210, 1996.

[11] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time Systems," *Real-Time Systems Journal*, vol. 1, no. 1, pp. 27–60, 1989.

[12] M. Stanovich, T. P. Baker, A. I. Wang, and M. G. Harbour, "Defects of the POSIX Sporadic Server and How to Correct Them," in *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2010.

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

[14] K. J. Duda and D. R. Cheriton, "Borrowed-Virtual-Time (BVT) Scheduling: Supporting Latency-Sensitive Threads in a General-Purpose Scheduler," in *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, 1999, pp. 261–276.

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

[16] Blink Without Delay: http://arduino.cc/en/Tutorial/BlinkWithoutDelay.

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

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

[19] 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.

[20] 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.

[21] 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.

[22] Arduino Yun: http://arduino.cc/en/Main/ArduinoBoardYun?from=Products.ArduinoYUN.

[23] P. L. Pasquale Buonocunto, Alessandro Biondi, "Real-Time Multitasking in Arduino," in *WiP session of the 9th IEEE International Symposium on Industrial Embedded Systems*, 2014.

[24] ERIKA Enterprise: http://erika.tuxfamily.org/drupal/.

[25] M. Spuri and G. Buttazzo, "Efficient Aperiodic Service under Earliest Deadline Scheduling," in *IEEE Real-Time Systems Symposium*, December 1994.

[26] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environment," *IEEE Transactions on Computers*, vol. 44, no. 1, pp. 73–91, January 1995.

[27] S. Oikawa and R. Rajkumar, "Linux/RK: A Portable Resource Kernel in Linux," in *Proceedings of the 19th IEEE Real-Time Systems Symposium*, 1998.

[28] Y. Zhang and R. West, "Process-aware Interrupt Scheduling and Accounting," in *Proceedings of the 27th IEEE Real Time Systems Symposium*, December 2006.

VI. Appendix

Listing 1. Qduino Autonomous Vehicle Sketch

```
#include <pin.h>
#include <stdio.h>
#include <ardutime.h>   // Qduino timing fns
#include <arducomm.h>   // Qduino comms APIs
#include <arduAdvIO.h>  // Qduino I/O APIs
#include <arduthread.h> // Qduino multi-loop API

const int leftmotorpin1 = 8;
const int leftmotorpin2 = 9;
const int rightmotorpin1 = 6;
const int rightmotorpin2 = 7;
const int trigPin = 11;
const int echoPin = 2;
channel distChannel;

void setup() {
  pinMode(leftmotorpin1, OUTPUT);
  pinMode(leftmotorpin2, OUTPUT);
  pinMode(rightmotorpin1, OUTPUT);
  pinMode(rightmotorpin2, OUTPUT);
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, FAST_INPUT);
  distChannel = channelInit();
}

//go forward
void nodanger() {
  digitalWrite(leftmotorpin1, HIGH);
  digitalWrite(leftmotorpin2, LOW);
  digitalWrite(rightmotorpin1, HIGH);
  digitalWrite(rightmotorpin2, LOW);
}

//reverse
void backup() {
 digitalWrite(leftmotorpin1, LOW);
 digitalWrite(leftmotorpin2, HIGH);
 digitalWrite(rightmotorpin1, LOW);
 digitalWrite(rightmotorpin2, HIGH);
}

//turn left
void body_lturn() {
 digitalWrite(leftmotorpin1, LOW);
 digitalWrite(leftmotorpin2, HIGH);
 digitalWrite(rightmotorpin1, HIGH);
 digitalWrite(rightmotorpin2, LOW);
}

//turn right
void body_rturn() {
 digitalWrite(leftmotorpin1, HIGH);
 digitalWrite(leftmotorpin2, LOW);
 digitalWrite(rightmotorpin1, LOW);
 digitalWrite(rightmotorpin2, HIGH);
}

void loop(1, 20, 100) {
  //read the distance from the channel
  long distance = channelRead(distChannel);
  //do actuation
  if (distance > 100) {
    nodanger();
  } else {
    backoff();
    body_rturn();
  }
  // Keep motors in current state
  // for fixed period
  delay(100);
}

void loop(2, 70, 100) {
  long duration;
  //do sensing
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);
  duration = pulseIn(echoPin, HIGH);
  //write the distance to the channel
  channelWrite(distChannel,
         (duration / 2) / 29.1);
  // Prevent sensor trigger signal being
  // falsely detected as echo using delay
  delay(200);
}
```

Listing 2. Clanton Linux Arduino Autonomous Vehicle Sketch

```
const int leftmotorpin1 = 8;
const int leftmotorpin2 = 9;
const int rightmotorpin1 = 6;
const int rightmotorpin2 = 7;

const int trigPin = 3;
const int echoPin = 2;
long distance;

void setup() {
  // Setup code is run once for initialization
  Serial.begin(9600);
  pinMode(leftmotorpin1, OUTPUT);
  pinMode(leftmotorpin2, OUTPUT);
  pinMode(rightmotorpin1, OUTPUT);
  pinMode(rightmotorpin2, OUTPUT);
  pinMode(trigPin, OUTPUT_FAST);
  pinMode(echoPin, INPUT_FAST);
}

//go forward
void nodanger() {
 digitalWrite(leftmotorpin1, HIGH);
 digitalWrite(leftmotorpin2, LOW);
 digitalWrite(rightmotorpin1, HIGH);
 digitalWrite(rightmotorpin2, LOW);
}

//back off
void backoff() {
 digitalWrite(leftmotorpin1, LOW);
 digitalWrite(leftmotorpin2, HIGH);
 digitalWrite(rightmotorpin1, LOW);
 digitalWrite(rightmotorpin2, HIGH);
}

//turn left
void body_lturn() {
 digitalWrite(leftmotorpin1, LOW);
 digitalWrite(leftmotorpin2, HIGH);
 digitalWrite(rightmotorpin1, HIGH);
 digitalWrite(rightmotorpin2, LOW);
}

//turn right
void body_rturn() {
 digitalWrite(leftmotorpin1, HIGH);
 digitalWrite(leftmotorpin2, LOW);
 digitalWrite(rightmotorpin1, LOW);
 digitalWrite(rightmotorpin2, HIGH);
}

void sensing()
{
  long duration;
  fastDigitalWrite(TrigPin, LOW);
  delayMicroseconds(2);
```

```
    fastDigitalWrite(TrigPin, HIGH);
    delayMicroseconds(10);
    fastDigitalWrite(TrigPin, LOW);
    duration = pulseIn(EchoPin, HIGH);
    distance = (duration / 2) / 29.1;
    // Prevent sensor trigger signal being
    // falsely detected as echo using delay
    delay(200);
  }

  void loop() {
    //do sensing
    sensing();

    //do actuation
    if (distance > 100)
      nodanger();
    else {
      backoff();
      body_rturn();
    }
    // Keep motors in current state
    // for fixed period
    delay(100);
  }
```

In the above listings, the lines of code for each sketch are almost identical if we discount comments and blank lines. There are 63 lines of code for the Qduino sketch, discounting the included header files, compared to 62 lines of code for the Clanton Linux Arduino sketch. When developing Arduino sketches for Clanton, the build environment automatically adds the header files. Notwithstanding, Qduino's support for multithreaded real-time loops is made simple. The Qduino API additionally provides a convenient way to exchange data between loop threads in real-time. Currently, Qduino supports either Simpson's four-slot (non-blocking) asynchronous communication, a semi-asynchronous ring buffer, or globally-shared data which is potentially guarded by spinlocks on multicore platforms. In general, the code complexity and ease of programmability in Qduino compares favorably to the conventional Arduino API.

NB: We do not include the sketch for the "Clanton Interrupt" code in Figure 12 due to space constraints. However, the code is 67 lines, and it is generally more complicated to program with interrupt handlers, whose timing is not guaranteed, than with multiple real-time loops.

Listing 3. The Qduino GNU C Macro for Multiple Real-Time Loops

```
#define loop(i,c,t)                                \
  extern void _loop##i();                          \
  void loop##i () {                                \
    struct sched_param s_params = {.type =
        MAIN_VCPU, .C = c, .T = t};                \
    int new_vcpu = vcpu_create(&s_params);         \
    vcpu_bind_task(new_vcpu);                      \
    while (1) _loop##i ();                         \
  }                                                \
  void loop##i##_init() {                          \
    pthread_create(&thread[i], NULL,
        (void *)loop##i, NULL);                    \
  }                                                \
  void _loop##i ()
```

Listing 4. The `main` Body for all Qduino Sketches

```
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <vcpu.h>

#define MAX_THREAD_NUM    32
pthread_t thread[MAX_THREAD_NUM];

#define WEAK_LOOP(i)           \
  extern void loop##i##_init() \
    __attribute__((weak))

//up to 32 loops
WEAK_LOOP(1);
WEAK_LOOP(2);
....
WEAK_LOOP(31);
WEAK_LOOP(32);
extern int loop() __attribute__((weak));

int main()
{
  int i, res, new_vcpu;
  struct sched_param s_params = {.type =
      MAIN_VCPU, .C = 80, .T = 100};

  extern void setup();
  setup();
  /* for backward compatibility
   * support conventional Arduino loop fn */
  if (loop) {
    /* create new vcpu */
    new_vcpu = vcpu_create(&s_params);
    vcpu_bind_task(new_vcpu);

    while(1) loop();
  } else {/* Support Qduino real-time loops */
    /* call user defined loop_init() to start
        thread */
    if (loop1_init) loop1_init();
    if (loop2_init) loop2_init();
    ....
    if (loop31_init) loop31_init();
    if (loop32_init) loop32_init();

    /* waiting for threads to finish */
    for (i = 0; i < MAX_THREAD_NUM; i++)
      if (thread[i] != 0)
        res = waitpid(thread[i]);
  }
  return 0;
}
```

Listings 3 shows the Qduino GNU C macro to define multiple real-time loops. Additionally, Listing 4 describes the `main` entry code, which is automatically linked with all Qduino sketches. The Qduino programming environment focuses on several objectives: (1) backward compatibility with the original Arduino API, supporting conventional `setup()` and `loop()` semantics, (2) ease of programming, and (3) support for multiple real-time loops, each assigned to time-budgeted threads of control. For multi-loop Qduino sketches, Listing 3 shows the template code to bind a loop to a Quest VCPU, with a specified budget and period. The loop initialization code in Listing 4 takes advantage of the GNU C `weak` attribute, to allow specification of up to 32 threaded real-time loops. The maximum value of 32 can easily be extended in Qduino if so desired. The `weak` attribute enables a user to define only those loops needed in a sketch without having to manually execute all the initialization code to bind each loop thread to a VCPU.

Source code for Qduino is available upon request. We hope to make a public repository available as soon as possible.