ENERGY EFFICIENCY FOR IOT SOFTWARE IN THE LARGE

1. INTRODUCTION

According to the International Energy Agency (IEA) [40], there were around 14 billion traditional network-connected devices in 2013 (e.g., mobile phones and smart TVs). This number is expected to increase to 50 billion by 2020 with the proliferation of IoT devices (e.g., smart bulbs and fitness wearables). IoT and traditional network-connected devices already outnumber people on the planet by a factor of two, and the amount of data traffic is expected to grow at an exponential rate in the next years. However, most of the energy to power these devices is consumed while they are in *standby mode*, i.e., when their software is neither transmitting or processing data. The annual CO_2 emissions related to standby in Australia are equivalent to those of 1 million cars. The projected growth of IoT devices together with the surprising effects of standby consumption, made network standby efficiency one of the six pillars of G20's *Energy Efficiency Action Plan*[1].

Many other organizations have also reported the importance of energy savings specifically in the context of the IoT and networked devices. In 2010, the Internet Engineering Task Force (IETF) created a working group on energy management (EMAN) with the following statement: "Energy management is becoming an additional requirement for network management systems due to several factors including the rising and fluctuating energy costs, the increased awareness of the ecological impact of operating networks and devices, and the regulation of governments on energy consumption and production" [57]. The American Council for an Energy-Efficient Economy (ACEEE) states that "The potential for new energy efficiency remains enormous, (...) looking ahead we must take a systems-based approach to dramatically scale up energy efficiency. (...) intelligent efficiency differs from component energy efficiency in that it is adaptive, anticipatory, and networked." [24] The pioneer *ENERGY STAR* program [13] states that "networked devices and networking equipment, as integrated systems, have the potential to contribute significant net energy savings". The US Environmental Protection Agency (EPA) and Information Technology Industry Council (ITI) held a workshop to explore roles for the *ENERGY STAR* and others program in promoting savings through intelligent energy efficiency [43].

With regard to concrete initiatives, IEA's *Electronic Devices and Networks* focuses specifically on the issue of networked device standby[2]. ACEEE's *Intelligent Efficiency* promotes a systematic approach to optimize the behavior of cooperating devices in order to achieve energy savings as a whole.

[1] G20's Energy Efficiency Action Plan: `https://www.iea-4e.org/projects/g20`

[2] EDNA initiative: `https://edna.iea-4e.org/`

Both approaches, device and system based, involve mostly software solutions, since energy savings are dynamic policies that depend on the application demands and device battery levels at a given moment in time. There is also a number of low-power wireless standards for the IoT infrastructure that suit different needs of range, throughput, and physical distribution [42]. For instance, *Bluetooth Low-Energy (BLE)* is a replacement for classic Bluetooth and is designed for lower data throughput in personal area networks (PANs). *6LoWPAN* adapts the IPv6 internet standard to low-power and limited-processing devices. These technologies make radio transmissions more efficient, support flexible network topologies, and reduce data traffic considerably. They also enable sleep modes that cut energy consumption to minimum levels. However, these technologies ultimately require software to control their functionalities and to wisely switch between standby and active modes in order to build an energy-efficient IoT.

1.1. **Objectives.** Effective use of low-power standby will play a fundamental role in energy efficiency for the expected 50 billion IoT devices by 2020 [40]. In order to succeed in this challenge, new solutions have to scale to the forthcoming mass of IoT software that must use standby modes effectively. This research project aims to address the software challenges, as determined by IEA, towards an energy-efficient IoT [40]:

- To ensure that devices employ the lowest possible modes of standby consumption.
- To ensure that devices remain in longest possible periods of standby time.

Given the projected scale of the IoT and the role of low-power standby towards energy efficiency, this research project has the following goals:

(1) Address energy efficiency through extensive use of standby.
(2) Target constrained embedded architectures that form the IoT.
(3) Provide standby mechanisms at the programming language level that scale to all applications.
(4) Support transparent/non-intrusive standby mechanisms that reduce barriers of adoption.

This proposal lies at the bottom of the software development layers—transparent programming language mechanisms—meaning that *all* applications would take advantage of low-power standby modes automatically, without extra programming efforts. We expect that existing energy-unaware applications will benefit from savings in the order of 50% based on IEA's estimates considering current standby technologies [40], and previous third-party work on transparent energy awareness [36].

1.2. **State of the Art.** Many energy-aware languages, extensions, and operating systems have been proposed recently. Some proposals adjust QoS (quality of service) to reduce power consumption (e.g., accuracy of computations, data sampling rates, scheduling times, etc.) [59, 25, 7, 44, 32, 34, 41, 60]. Other proposals offer mechanisms to switch behaviors depending on application demands and battery levels (e.g., disable functionalities, switch UI, etc.) [56, 17, 15, 16]. None of these initiatives are

concerned with taking advantage of standby modes when idle, but more with adapting or eliminating computations while in active modes (not addressing goal 1 of Section 1.1). There are also specialized network protocols that make devices remain in low-power standby modes for longer periods of time [4]. However, protocol-based initiatives only apply to the networked parts of applications and have to be programmed carefully to take advantage of standby modes (not addressing goals 3 and 4).

Despite the advances in energy-aware languages, embedded systems are still primarily programmed in the C language [8]. The predominance of C is associated with its portability across architectures, efficiency in terms of memory and CPU, extensive legacy codebase, and number of programmers. Therefore, the full IoT development stack typically relies on C [31, 22, 6], from high-level applications and network protocols, down to operating systems, network drivers, and SoC firmwares. However, C is a simple hardware abstraction (sometimes referred to as the *portable assembly*) and has no awareness of the surrounding environment it is embedded on. For instance, there is no dedicated vocabulary in the language to express concepts that naturally appear in IoT applications, such as time, communication with the real world, concurrency of events, and energy awareness. C is also memory unsafe, leading to all sorts of bugs such as memory leaks, buffer overflows, and wild pointers [20].

TinyOS [31] uses a C dialect [26] and partially addresses our goals. It targets constrained architectures and maximizes standby time (goals 1 and 2). TinyOS is also transparent at the level of the operating system, meaning that applications built on top of it will take advantage of standby power for free (goals 3 and 4). They claim to operate in sleep modes 44% of the time [36], which is close to IEA's estimates of 65% energy savings. However, an operating system codebase is complex and still written in C. Our goal is to provide transparent support at the language level, which even an OS implementation could take advantage (goal 3). Another issue with TinyOS is related to the adoption barrier imposed by its programming model (goal 4): applications have to be written in a callback-driven style in C, which is widely recognized as hard and error prone [3, 23, 19, 37, 39]

1.3. **Preliminary Results.** I have been working in the design of CÉU, a new reactive programming language targeting resource-constrained embedded systems, for the past 8 years [45, 50, 53, 51, 46, 47, 52, 48, 11, 54, 49]. CÉU is grounded on the synchronous concurrency model, which has been successfully adopted in the context of hard real-time systems such as avionics and automobiles industry since the 80's [9]. The synchronous model trades power for reliability and has a simpler model of time that suits most requirements of IoT applications. On the one hand, this model cannot directly express time-consuming computations, such as compression and cryptography algorithms, which are typically either absent or delegated to auxiliary chips in the context of the IoT. On the other hand, all reactions to the external world are guaranteed to be computed in bounded time, ensuring that applications always reach an idle state amenable to standby mode.

```c
digitalWrite(13, 0);
int old = 0;
while (1) {
    int cur = digitalRead(2);
    if (cur != old) {
        digitalWrite(13, cur);
        old = cur;
    }
}
```

[a] Active polling in C

```
1  input  int PIN_02;
2  output int PIN_13;
3
4  emit PIN_13(0);
5  loop do
6      var int v = await PIN_02;
7      emit PIN_13(v);
8  end
9  .
```

[b] Reactive execution in CÉU

FIGURE 1. Controlling a LED from button state changes.

In a previous work [53], we adapted CÉU to execute on top of TinyOS in the context of Wireless Sensor Networks, and developed a number of applications, protocols, and device drivers. We evaluated the expressiveness of CÉU in comparison to C and attested a reduction in source code size (around 25%) with a small increase in memory usage (around 5-10% for text and data). We also evaluated CPU responsiveness under heavy network load and attested a similar performance between CÉU and C, which implies comparable energy efficiency while in active mode. In a related work, we developed Terra, a virtual machine for CÉU targeting constrained networked devices that supports low-power remote reprogramming [11]. Virtual machine applications are an order of magnitude smaller than full binaries, reducing data traffic considerably and, consequently, energy consumption.

In a recent paper [49], we discuss in extent the design decisions behind CÉU under the perspective of synchronous languages, providing a deep comparison with the seminal language Esterel [10]. The most fundamental difference to Esterel is CÉU's event-triggered notion of time, in which the occurrence of an event (e.g., a button press) defines an indivisible instant in an application logical timeline. This characteristic makes the model of time of CÉU exclusively reactive to incoming events. As a consequence, time does not elapse during idle periods, making all applications susceptible to standby.

1.4. **A Motivating Example.** Figure 1 shows a basic program that makes a button connected to the microcontroller's pin 2 to control a LED connected to pin 13. Whenever the state of the button changes (i.e., pressed or unpressed), the state of the LED also changes (i.e., on or off).

The implementation in C in Figure 1.a first turns off the LED (ln. 1), then sets an initial state for the button (ln. 2), and then reads in a loop (ln. 3-9) its current state (ln. 4) as fast as possible. If the current state differs from the old state (ln. 5-8), the new state is copied to the LED (ln. 6), and is also set as the old state (ln. 7). This approach which actively samples the state of an external device (e.g., a button) is known as *polling* or *software-driven I/O*. On the one hand, this style does not involve dealing with the complexity of callbacks (as discussed in Section 1.2) and is widely adopted in Arduino, the most popular embedded platform (to be discussed in Section 2.1.2). On the other hand, polling wastes CPU cycles and prevents the device to enter in standby.

The implementation in CÉU in Figure 1.b uses dedicated vocabulary to handle events from I/O devices without resorting to callbacks. The pins are first declared to determine their purpose in the program, i.e., `input` for the pin connected to the button and `output` for the pin connected to the LED (ln 1-2). Then, the LED is turned off with an `emit` (ln. 4), which indicates an output operation. Then, the program enters in a loop to react to changes in the button (ln. 5-8). The `await` primitive accounts for the reactive nature of CÉU. The program literally waits for a change in the button (ln. 6) to be able to proceed and set the new state of the LED (ln 7). This way, when an application reaches an idle state, the language has precise information about which events can awake that application.

By design, all lines of execution in CÉU always reach an `await` at the end of a reaction to an event (otherwise, the application does not compile [53]). This not only allows the application to enter standby mode, but effectively use the deepest sleeping level considering all possible awaking events. While in standby, only a hardware interrupt associated with the events can awake the application, making it sleep for longest possible periods of time.

2. METHODOLOGY

2.1. **First Year.**

2.1.1. *IoT Hardware Infrastructure.* The IoT infrastructure is built on top of embedded systems such as sensor nodes, network routers, and other low-power microcontroller-based devices and appliances.

We will use off-the-shelf Arduinos [35] as the main hardware IoT platform. Most Arduinos are based on low-cost 8-bit microcontrollers from Atmel, such as the popular *ATmega328p* [5], which supports six sleeping modes which can reduce power consumption to very low levels. Depending on some configurations (e.g., frequency and voltage), an Arduino can draw from 45mA at full operation, down to 5uA in power down mode (the deepest sleeping mode). We will pursue that IoT applications operate only 50% of the time in the average, which is not unlikely [36]. Considering the negligible consumption in power down mode, this will meet our goals of reducing energy consumption by 50%.

Besides the good support for low-power standby, Arduinos are inexpensive, open source, and widely used. We will take advantage of the many environments in which Arduino is popular:

Research: There is a lot of research using Arduino in the context of the IoT (e.g., infrastructure [27], healthcare [21], home automation [38], and energy awareness [28]). In addition to the availability of support literature, the popularity of Arduino will make our own research more accessible and reproducible to other groups.

Education: Arduino is used in many University courses worldwide (e.g. [14, 12, 55, 33]). We have been using Arduino with CÉU in an undergraduate course for the past 3 years, which will allow us to evaluate our results with less experienced embedded programmers.

Hobbyists: The Arduino hobbyist community cultivates an abundance of resources such as forums, wikis, and publicly available software. It is important for us to have available software that we can adapt to CÉU to evaluate energy efficiency gains.

Non-Programmers: Arduino is also becoming popular among non-programmers and a lot of software will be produced by them in the near future. We are in the process of creating an interdisciplinary course targeting non-specialists (e.g., arts, design, architecture). This audience will be a good target to evaluate our transparent approach for energy efficiency.

Expected Contributions. Since we are adopting a third-party hardware platform with existing support for low power modes, we do not expect significant contributions.

Risks (very low: 1/5). Arduino is a cheap and popular embedded platform with widespread use in the academia. There is a considerable number of reports that claim to achieve very low power consumption with Arduino in deep standby modes.

2.1.2. *IoT Software Infrastructure.* Arduino provides a thin software layer on top of C and C++, and the typical way in which applications are built is through polling (as discussed in Section 1.3). In Arduino, even basic functionality, such as timers (`delayMicroseconds`), A/D converters (`analogRead`), and SPI (`SPI.transfer`), uses busy-wait loops that waste energy in active mode.

In order to provide automatic standby, applications need to be entirely reactive to events, and we will rebuild all of the software infrastructure from scratch in CÉU. This process will consist primarily of rewriting device drivers, which are the pieces of software that interface directly with the hardware. More concretely, at the lowest level, energy-efficient software depends on *interrupt service routines (ISRs)*, which awake the CPU from standby when an external peripheral event occurs.

Recently, we have added support for ISRs as a primitive concept of CÉU, which will allow us to rebuild the IoT software infrastructure with standby awareness from the ground up. Figure 2 is an sketch of our intended approach to achieve transparent standby modes (the lines with <. . .> are pseudo code). The application in Figure 2.a requests, every hour (ln. 7-11), an A/D conversion (ln. 8) and awaits its completion (ln. 9) to perform some action (collapsed in ln. 10). Note that the application uses abstract input/output events to interface with the A/D converter (ln. 3-4) and does not manage energy explicitly. Figure 2.b would be the associated driver with the A/D converter, which implements the input/output events declared as abstractions in the application code. The output request (ln. 3-7) configures the A/D peripheral (ln. 4) and enables interrupts in the microcontroller (ln. 5). It also needs to set the deepest mode supported by the A/D peripheral (ln. 6) so that the language uses it as soon as the program reaches the idle state. The input definition (ln. 9-13) implements the ISR for the A/D converter. When the conversion is done, the peripheral awakes the CPU, and this code executes immediately in interrupt context. The code disables A/D interrupts (ln. 10), reads the A/D converter data register with the converted value (ln. 11), and returns this data to the application (ln. 12). A

```
// application.ceu

output none ADC_REQUEST;
input  int  ADC_DONE;
#include "adc.ceu" // driver implementation

every 1h do
    emit ADC_REQUEST;
    var int v = await ADC_DONE;
    <performs-some-action>;
end

.
```

[a] Application: no explicit energy management

```
// adc.ceu

output none ADC_REQUEST do
    <configures-ADC>;
    <enables-ADC-interrupts>;
    <sets-the-deepest-standby-mode>;
end

input int ADC_DONE [ADC_vect_num] do
    <disables-ADC-interrupts>;
    var int v = <reads-adc-data>;
    return v;
end
```

[b] Driver: explicit energy management

FIGURE 2. ISRs in CÉU

`return` in interrupt context will enqueue the input event `ADC_DONE`, which will awake the application code associated with it (ln. 9 in Figure 2.a).

With this approach, the application code remains similar to the one would write for Arduino (apart from minor syntax mismatches). However, instead of wasting cycles in busy-wait loops, the applications will enter the deepest possible standby mode when idle.

Expected Contributions.

(1) **An environment-aware system language:** CÉU's dedicated vocabulary to deal with events raises the programming abstraction to a level closer to the domain of IoT, providing more safety and expressiveness to programmers. These results have already been partially published in the past [53], when we implemented a radio device driver on top of TinyOS. Nevertheless, the novel support for ISRs would complete the whole development cycle, from the application down to the hardware, without operating system support.

(2) **An energy-aware system language:** The method described in this section makes all applications subject to standby modes transparently. Only device drivers, which are typically provided by vendors, will require explicit energy management. In our case, we will need to rewrite each driver only once, as part of the software infrastructure, and all applications built on top of them would benefit from energy efficient automatically.

Risks (high: 4/5). As far as we know, abstract ISR support at the language level has not been tried in the past. The challenge is to choose the right level of abstraction that is, at the same time, powerful to support the range of devices and portable to work in multiple platforms. Since CÉU integrates seamlessly with C [49], it is always possible to circumvent abstractions, at least in early stages of development, until the appropriate level is found.

```
pinMode(13, OUTPUT);
while (1) {
    digitalWrite(13, 1);
    delay(1000);
    digitalWrite(13, 0);
    delay(1000);
}
```

[a] Blinking a LED in C

```
1  output int PIN_13;
2  loop do
3      emit PIN_13(1);
4      await 1s;
5      emit PIN_13(0);
6      await 1s;
7  end
```

[b] Blinking a LED in CÉU

FIGURE 3. Blinking a LED connected to pin 13 every second.

From the engineering perspective, rewriting drivers is a laborious endeavour. As a rough reference, about half of the 15 million lines of the Linux codebase consists of device drivers. With the tight integration between CÉU and C, however, significant parts of existing drivers can be reused.

We already have some evidence of the feasibility of programming device drivers in CÉU based on a project of this year's *Google SoC*[3]. In the period of 3 months, an undergraduate student studied and implemented drivers for an A/D and SPI peripherals using the recent support for ISRs in CÉU.

2.1.3. *IoT Applications.* In order to evaluate the gains in energy efficiency with the proposed infrastructure, we will need to evaluate the consumption of realistic applications. The Arduino community has an abundance of open-source projects which can be rewritten in CÉU to take advantage of transparent standby modes (as illustrated in Figure 2). Then, we will be able to compare the original and rewritten versions in terms of energy consumption to draw conclusions about the effectiveness of transparent standby modes.

We will start with simple applications, such as the *Blink* that ships with the Arduino distribution and blinks an LED every second. The code is reproduced in Figure 3.a and a very similar version in CÉU appears in Figure 3.b. The example is in active mode for a negligible time just to set the I/O ports (ln. 3 and 5), remaining idle most of the time. We expect this application to consume much less energy in CÉU, since the `await` statements (ln. 4 and 6) will put the device into standby automatically, while the version in C uses a busy-wait loop inside the `delay` statements (ln. 4 and 6).

The most realistic scenarios for the IoT use radio communication extensively. In this context, we will evaluate from simple handmade ad-hoc protocols to more complex energy-aware protocols (as discussed in Section 1.2) and see to what extent our proposal would effectively contribute to energy savings. The *RadioHead* project[4] supports dozens of RF (radio frequency) chipsets and provides examples that explore mesh topologies as well as low-power energy modes. We plan to support at least two chipsets: *nRF24L01* [2] and *RFM69HCW* [1], both of which have good support for low power consumption.

[3]GSoC project: `http://www.lua.inf.puc-rio.br/gsoc/ideas2017.html#ceu`

[4]RadioHead project: `http://www.airspayce.com/mikem/arduino/RadioHead/`

In the long term, we expect to make the case for developers to rewrite their applications in CÉU to take advantage of standby modes for free. Towards this direction, we will evaluate IoT applications rewritten in CÉU based on the following criteria:

Time to rewrite: This criteria relates to the incentives to rewrite existing applications to CÉU. It is a tradeoff between the expected rewriting times and gains in energy efficiency as compared to standard Arduino code.

Coding "aesthetics": To lower the adoption barrier, it is also important that the proposed programming style is sufficiently familiar, and, at least, as easy to write applications.

Energy consumption: More importantly, rewritten applications should have significant gains in energy efficiency to justify a complete application rewrite.

Expected Contributions.

(1) **A realistic programming alternative for the IoT in the Arduino:** CÉU is a 8-year project and has a mature open-source implementation that is publicly available[5]. With the adaptation to the context of energy-efficient IoT, CÉU may cross the academic fences in the short term and become a practical alternative for the Arduino community.

(2) **Energy-efficient IoT applications:** We expect substantial gains in energy efficiency for all applications that do not manage energy explicitly. Most Arduino APIs use busy-wait polling to wait for input, which are very energy inefficient, and could take advantage of transparent standby.

Risks (medium: 3/5). We do not know precisely the practical gains in energy efficiency in the standby modes of Arduino. Most reports that compare theoretical (from official datasheets) and measured consumption claim good performance but are non peer-reviewed research. For instance, a specialized company[6] provides Arduino-based IoT devices and a cross-platform library dedicated to standby modes that is widely used by the community.

2.2. **The Three Years to Follow.**

2.2.1. *Systematic Energy Awareness.* As discussed in Section 1, systematic energy awareness considers the IoT network as a whole, with cooperating devices trying to maximize energy efficiency globally. In this context, *adaptive computing* [29] is the capability of the IoT to adapt energy consumption dynamically, based on application demands and levels of power supply during execution. Ultimately, the goal is still that each device maximizes its idle periods to be more susceptible to standby modes. Hence, the research of the first year is a prerequisite for this phase.

[5]CÉU repository: `https://github.com/fsantanna/ceu`

[6]RocketScream: `http://www.rocketscream.com`

```
1  input none BUTTON_PRESSED;              // input for every button press
2  event bool enter_low_power_mode;        // event to toggle power mode
3  par do
4      <mandatory-functionality>           // omitted code with basic functionality
5  with
6      pause/if enter_low_power_mode do    // the nested block is suspended in low mode
7          <energy-costly-functionality>   // omitted code with costly functionality
8      end
9  with
10     loop do                             // toggles power mode on every button press
11         await BUTTON_PRESSED;
12         emit enter_low_power_mode(true);
13         await BUTTON_PRESSED;
14         emit enter_low_power_mode(false);
15     end
16 end
```

FIGURE 4. Adaptive application that changes behavior depending on the power mode.

We will continue to investigate how language mechanisms can aid energy efficiency, but now in the context of adaptive computing. Figure 4 is a sketch of an application that adapts its behavior under low-power mode by suspending parts of its functionality (the lines with `<...>` are pseudo code). The application relies on the `par` construct of CÉU (ln. 3-16) which creates three concurrent lines of execution (ln. 4, 6-8, and 10-15). The first line of execution (collapsed in ln. 4) implements the mandatory functionality of the application, which must execute regardless of the power mode. The second line of execution (ln. 6-8) implements extra functionality (collapsed in ln. 7) that can be suspended in low-power mode without affecting the main functionality. The `pause/if` construct of CÉU (ln. 6) suspends the nested costly functionality whenever the event `enter_low_power_mode` is set to *true* and resumes when set to *false*. The third line of execution is responsible for implementing the adaptive manager by interacting with the external world. In this case, every time the button is pressed (ln. 11 and 13), the device switches the power mode by triggering the event `enter_low_power_mode` (ln. 12 and 14).

The example suggests that `BUTTON_PRESSED` represents a simple button connected to the device. However, the advantage of using the abstract concept of an input event is that the actual implementation may vary. For instance, `BUTTON_PRESSED` might be a "virtual button" associated to a remote device (or set of devices). We will take advantage of the abstract concepts of input and output events to be able to specify IoT interactions at a higher level.

Note that adaptive computations must be programmed explicitly, since they are application dependent and have particular requirements. Hence, since it is not possible to offer transparent language mechanisms (such as in our proposal for standby modes), we will investigate *non-intrusive language mechanisms*. For instance, note in the example that the `<energy-costly-functionality>` (ln. 7) did not suffer any internal modifications to be suspended, but was simply enclosed by the `pause/if`

construct. This way, making an application adaptive to embrace energy efficiency will require less programming efforts. Furthermore, the implementation of the application functionalities will not be intermixed with energy awareness, making the code more modular.

The evaluation process for systematic energy awareness will be similar to the one adopted for device-based awareness (as discussed in Section 2.1.3): take existing energy-aware IoT protocols and applications for Arduino, and rewrite them using non-intrusive mechanisms. Then, employ the same evaluation criteria (i.e., time to rewrite, coding aesthetics, and energy consumption).

Expected Contributions.

(1) **Higher-level programming abstractions:** Similarly to the IoT software infrastructure and applications (discussed in Sections 2.1.2 and 2.1.3), we expect that systematic energy awareness will take advantage of the non-intrusive mechanisms that CÉU provides and will be implemented at a higher level of abstraction.

(2) **Adaptive and energy-efficient IoT applications:** We expect to achieve the same energy efficiency as that of hand-tweaked applications but with less programming efforts.

Risks (medium: 3/5). On the one hand, hand-tweaked applications always have a theoretical higher efficiency than applications based on richer abstractions. On the other hand, having to deal with the complexity of hand-tweaked applications makes them less amenable to adaptive computations. In some situations, we may face this tradeoff and sacrifice adaptiveness to achieve satisfactory energy efficiency.

From the engineering perspective, rewriting complex applications takes a lot of time. However, considering the period of 4 years, time estimates can be reconsidered periodically to employ more human resources into the project.

2.2.2. *Complex IoT Architectures and Applications.* The niche of constrained embedded systems, which includes Arduino, covers the substantial (and increasing) portion of IoT applications. They typically do not require significant computing resources but are sensitive to physical dimensions and energy consumption. However, the IoT also consists of more traditional networked devices, such as routers, servers, and smartphones. As of 2016, there were 3.9 billion smartphone subscriptions worldwide, and this number is expected to reach 6.8 billions by 2022 [30].

Smartphones rely on much more complex architectures than microcontroller-based embedded systems (e.g., 32-bit CPUs with memory management unit). They typically rely on an operating system, a complete TCP/IP stack, and can execute multiple applications at the same time. In addition to reactive applications, typical of the IoT, smartphones also perform active computations, such as audio/image/video processing and cryptographic functions. Nevertheless, they are an important piece of the IoT, serving as a common human interface to process, visualize, and actuate on the network.

Smartphones have similar restrictions in battery consumption and could also take advantage of the techniques we propose for constrained embedded systems. In order to transpose the barrier from constrained devices to smartphones for the IoT, we will take a similar path as presented in Section 2.1:

Hardware Infrastructure: We will use the BeagleBone Black [18], which shares similar goals with Arduino, providing a cheap and open-source platform but which is suitable for rich applications, such as graphical user interfaces, multimedia, and games. It is based on the Texas Instruments (TI) Sitara ARM architecture [58], featuring a 1GHz CPU, with 512MB RAM and 2GB flash. Like happens with Arduino, the BeagleBoard community offers an abundance of open projects and publicly available software.

Software Infrastructure: In order to ensure automatic standby for applications, all software infrastructure, mostly device drivers, needs to be rebuilt from scratch using ISRs in CÉU. However, smartphones have much more complex peripherals, such as WiFi adapters and graphics processors. To reduce the barrier adoption, TI provides *StarterWare*[7], a free software development package with open support for the basic Sitara peripheral drivers.

Applications: In addition to IoT applications, typical smartphone applications, such as instant messaging and internet browsing, can also maximize energy efficiency through standby. We will rewrite from simple applications, such as a graphical clock, to more complex networked applications such as a web browser:

- **xclock** displays the time in analog or digital and the source code is around 5k lines in C.
- **Dillo**[8] is a minimalistic web browser with the source code around 50k lines in C.

Expected Contributions.

(1) **Energy-efficient smartphones:** The transition from simple mobile phones to smartphones resulted in a degrading effect on battery lives. Phones used to have autonomy of weeks and now require daily recharges. Most of the time, however, smartphones are sitting in our pockets wasting energy. We expect to increase the battery autonomy considerably while keeping the functionality of a modern smartphone.

(2) **Towards an environment- and energy-aware operating system:** The whole software technology of Smartphones is built on top of bloated and antiquated personal computer operating systems like Windows and UNIX (e.g., macOS and Android). These operating systems were not designed with mobile systems and energy awareness in mind. Our proposal is a step towards a new environment- and energy-aware operating system.

Risks (very high: 5/5). Modern smartphone hardware architectures are much more complex than of constrained embedded systems. For this reason, making an application to be energy-aware from the

[7]TI Sitara: `http://www.ti.com/tool/starterware-sitara`

[8]Dillo web browser: `https://www.dillo.org/`

ground up, starting from a hardware interrupt, is an enduring challenge. Nonetheless, the proposed language infrastructure is the same, and all other required pieces such as drivers are publicly available.

3. TIMELINE

Figure 5 shows the project timeline for the full period of 4 years. The two lines in black refer to the *constrained* and *complex* IoT architectures phases (Sections 2.1/2.2.1 and Section 2.2.2, respectively). They only intersect during the second semester of the second year, when the constrained phase is about to terminate and the complex phase kicks off. At the end of the first year, we expect to address energy efficiency for the constrained IoT architectures, with the hardware and software infrastructure set, and the basic and networked applications completed (Section 2.1). The second year covers systematic energy awareness for the constrained IoT (Section 2.2.1) and marks the beginning of the complex IoT phase (Section 2.2.2), which extends to the end of the period of 4 years.

The small tasks in red are related to hardware acquisition and setup of existing IoT architectures. The tasks in yellow refer to the software infrastructure, which will be in continuous iterations during the whole project cycle. They require the most experienced researchers and consist of designing the language mechanisms, employing automatic standby, and implementing the most essential drivers. The tasks in green refer to the applications built on top of the software architecture. They have delimited development cycles and require less experienced researchers.

4. RESOURCES

For the period of 3 years, we will require a budget of R$345k:

- R$300k for staff payment (87%),
- R$20k for equipment (6%), and
- R$25k for publication and travel costs (7%).

4.1. **The Team.** Most of the resources will be employed in research staff consisting of undergraduate, MSc, and PhD students. The quality and long-term commitment of the students is paramount for the success of the project. For this reason, we want to offer a salary compatible with the industry, which is considerably larger than typical research scholarships.

Proponent: The proponent will work in all project fronts: managing the resources and staff, writing papers and supervising students, and working on the technical challenges of the project.

Advisors: The proponent is an associate professor at UERJ and a member of the LabLua[9] at PUC-Rio. The advisors are experienced researchers from UERJ and PUC-Rio in the field of programming languages, distributed systems, and wireless sensor networks. They will support

[9]LabLua: `http://www.lua.inf.puc-rio.br/`.

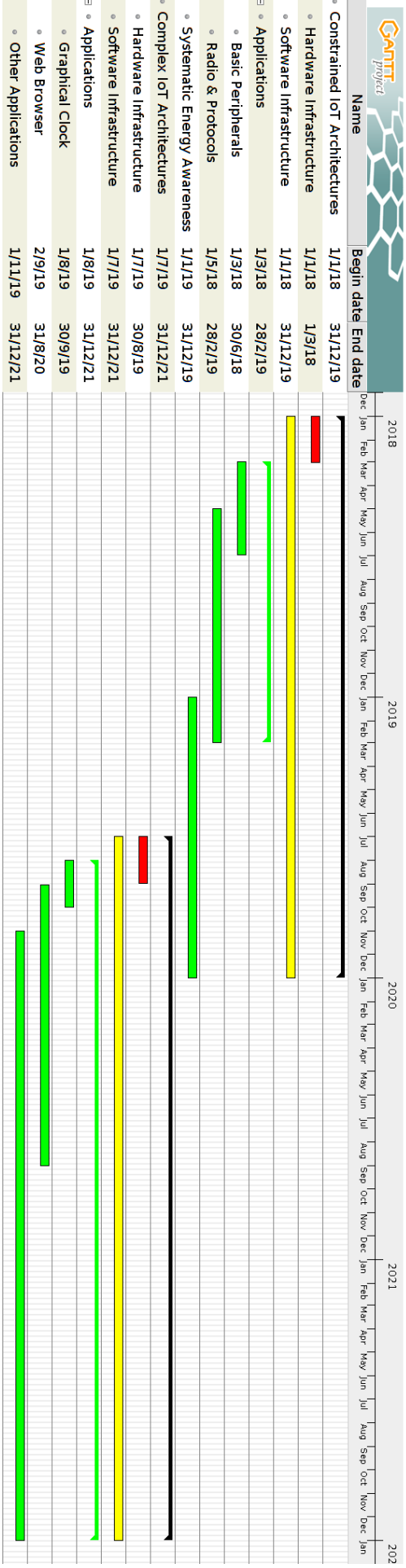

FIGURE 5. Project timeline

the academic duties, such as prospecting and supervising students, writing papers, and helping on the general project guidelines.

Research staff: At the beginning of the first year, we will appoint two students to join the project full time: one early-stage PhD student and one MSc student or late-stage undergraduate student. Every six months, we plan to add a new member to the staff until we reach a limit of 6 students (consisting of 2 or 3 PhD students). The PhD students will work with the software infrastructure and the other students with the applications. We also expect that the PhD students will work on academic papers and take advantage of the project in their thesis. Considering an average of 4 students during the whole project with and an average salary of R$2k, we expect to spend around R$300k with payments.

4.2. **Infrastructure and Equipment.** The required equipment is divided between support for the research staff (printer and laptops) and the project embedded systems (Arduino, Beagle Black, etc.):

EQUIPMENT	PRICE	QTD	TOTAL
Laser printer	3000	1	3.000
Laptop	4000	3	12.000
Arduino kits	200	5	1.000
Sensors & radios	-	-	2.000
Beagle Black	600	2	1.200
Beagle Display	400	2	800
			R$20k

We expect that half of these resources will be acquired at the beginning of the project. The other half will be acquired gradually during the course of the 3 years.

4.3. **Publication and Travel Costs.** Our goal is to publish our results on international journals in the field of embedded systems and sensor networks, such as *ACM TECS* and *ACM TOSN*. However, the process to publish a paper in a journal may take longer than a year after the submission. For this reason, we also intend to publish partial results in conferences.

During the 3 years, we plan to attend two Brazilian conferences (e.g., *SBRC*), and one prestigious international conference (e.g., *ACM SenSys*), These venues are important to cultivate future cooperation with other research groups. For the Brazilian conferences, we will support 2 participants, and for the international conference, only 1 participant. Using the CNPq travel grants as a reference, attending a national conference costs around R$3k (including travel and accommodation for 3 days), while a international conference costs around US$4k (including travel and accommodation for 5 days). Considering the two Brazilian conferences with two participants each, and one international conference with one participant, the total travel costs for the 3-year period will be around R$25k.

REFERENCES

[1] Rfm69hcw datasheet. Technical report, HopeRF Electronic, 2006.

[2] nrf24l01 datasheet. Technical report, Nordic Semiconductor, 2007.

[3] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Proceedings of ATEC'02*, pages 289–302. USENIX Association, 2002.

[4] G. Anastasi, M. Conti, M. Di Francesco, and A. Passarella. Energy conservation in wireless sensor networks: A survey. *Ad Hoc Netw.*, 7(3):537–568, May 2009.

[5] Atmel. *ATmega328P Datasheet*, 2011.

[6] 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*, pages 79–80. IEEE, 2013.

[7] W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 198–209, New York, NY, USA, 2010. ACM.

[8] M. Barr. Real men program in C. *Embedded Systems Design*, 22(7):3, 2009.

[9] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. D. Simone. The synchronous languages twelve years later. In *Proceedings of the IEEE*, volume 91, pages 64–83, Jan 2003.

[10] F. Boussinot and R. De Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, Sep 1991.

[11] A. Branco, F. Sant'anna, R. Ierusalimschy, N. Rodriguez, and S. Rossetto. Terra: Flexibility and safety in wireless sensor networks. *ACM Trans. Sen. Netw.*, 11(4):59:1–59:27, Sept. 2015.

[12] J. D. Brock, R. F. Bruce, and S. L. Reiser. Using arduino for introductory programming courses. *J. Comput. Sci. Coll.*, 25(2):129–130, Dec. 2009.

[13] R. Brown, C. Webber, and J. G. Koomey. Status and future directions of the ENERGY STAR program. *Energy*, 27(5):505–520, 2002.

[14] L. Buechley, M. Eisenberg, J. Catchen, and A. Crockett. The lilypad arduino: Using computational textiles to investigate engagement, aesthetics, and diversity in computer science education. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 423–432, New York, NY, USA, 2008. ACM.

[15] A. Canino. Gradual mode types for energy-aware programming. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH Companion 2015, pages 79–80, New York, NY, USA, 2015. ACM.

[16] A. Canino and Y. D. Liu. Proactive and adaptive energy-aware programming with mixed typechecking. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 217–232, New York, NY, USA, 2017. ACM.

[17] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu. Energy types. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 831–850, New York, NY, USA, 2012. ACM.

[18] G. Coley. Beaglebone black system reference manual. Technical report, BeagleBoard.org, 2013.

[19] M. de Icaza. Callbacks as our generations' go to statement. `http://tirania.org/blog/archive/2013/Aug-15.html` (accessed in Aug-2014), 2013.

[20] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. *SIGPLAN Not.*, 38(7):69–80, June 2003.

[21] C. Doukas and I. Maglogiannis. Bringing iot and cloud computing towards pervasive healthcare. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, pages 922–926. IEEE, 2012.

[22] Dunkels et al. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of LCN'04*, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[23] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of SenSys'06*, pages 29–42. ACM, 2006.

[24] R. N. Elliott, M. Molina, and D. Trombley. A defining framework for intelligent efficiency. Technical Report E125, American Council for an Energy-Efficient Economy (ACEEE), 2012.

[25] J. Flinn and M. Satyanarayanan. Managing battery lifetime with energy-aware adaptation. *ACM Trans. Comput. Syst.*, 22(2):137–179, May 2004.

[26] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of PLDI'03*, pages 1–11, 2003.

[27] V. Georgitzikis, O. Akribopoulos, and I. Chatzigiannakis. Controlling physical objects via the internet using the arduino platform over 802.15. 4 networks. *IEEE Latin America Transactions*, 10(3):1686–1689, 2012.

[28] K. Gomez, R. Riggio, T. Rasheed, D. Miorandi, and F. Granelli. Energino: A hardware and software solution for energy consumption monitoring. In *Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks (WiOpt), 2012 10th International Symposium on*, pages 311–317. IEEE, 2012.

[29] J. Henkel and L. Bauer. What is adaptive computing? *SIGDA Newsl.*, 40(5):1–1, May 2010.

[30] N. Heuveldop. Ericsson mobility report. Technical report, Ericsson, AB, 2017.

[31] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *SIGPLAN Notices*, 35:93–104, November 2000.

[32] H. Hoffmann. Jouleguard: Energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 198–214, New York, NY, USA, 2015. ACM.

[33] P. Jamieson. Arduino for teaching embedded systems. are computer scientists and engineering educators missing the boat? *Proc. FECS*, 289294, 2010.

[34] A. Kansal, S. Saponas, A. B. Brush, K. S. McKinley, T. Mytkowicz, and R. Ziola. The latency, accuracy, and battery (lab) abstraction: Programmer productivity and energy efficiency for continuous mobile context sensing. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 661–676, New York, NY, USA, 2013. ACM.

[35] D. Kushner. The making of arduino. *IEEE Spectrum*, 26, 2011.

[36] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. Tinyos: An operating system for sensor networks. *Ambient intelligence*, 35:115–148, 2005.

[37] I. Maier, T. Rompf, and M. Odersky. Deprecating the observer pattern. Technical report, 2010.

[38] K. Mandula, R. Parupalli, C. A. Murty, E. Magesh, and R. Lunagariya. Mobile based home automation using internet of things (iot). In *Control, Instrumentation, Communication and Computational Technologies (ICCICCT), 2015 International Conference on*, pages 340–343. IEEE, 2015.

[39] E. Meijer. Reactive extensions (rx): curing your asynchronous programming blues. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 11:1–11:1, New York, NY, USA, 2010. ACM.

[40] OECD/IEA. More data less energy—Making network standby more efficient in billions of connected devices. Technical report, International Energy Agency, 2014.

[41] J. Park, H. Esmaeilzadeh, X. Zhang, M. Naik, and W. Harris. Flexjava: Language support for safe and modular approximate programming. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 745–757, New York, NY, USA, 2015. ACM.

[42] G. Reiter. Wireless connectivity for the internet of things. Technical report, Texas Instruments, 2014.

[43] E. A. Rogers, R. N. Elliott, S. Kwatra, D. Trombley, and V. Nadadur. Intelligent efficiency: opportunities, barriers, and solutions. Technical Report E13J, American Council for an Energy-Efficient Economy (ACEEE), 2013.

[44] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. *SIGPLAN Not.*, 46(6):164–174, June 2011.

[45] F. Sant'Anna. Céu: A reactive language for wireless sensor networks. `http://www.cse.ust.hk/~lingu/SenSys11DC/`, 2011.

[46] F. Sant'Anna. *Safe System-level Concurrency on Resource-Constrained Nodes with Céu*. PhD thesis, PUC–Rio, 2013.

[47] F. Sant'Anna et al. Structured reactive programming with céu. Workshop on Reactive and Event-based Languages & Systems (REBLS'14), 2014.

[48] F. Sant'Anna et al. Reactive traversal of recursive data types. Workshop on Reactive and Event-based Languages & Systems (REBLS'15), 2015.

[49] F. Sant'anna, R. Ierusalimschy, N. Rodriguez, S. Rossetto, and A. Branco. The design and implementation of the synchronous language cÉu. *ACM Trans. Embed. Comput. Syst.*, 16(4):98:1–98:26, July 2017.

[50] F. Sant'Anna, N. Rodriguez, and R. Ierusalimschy. Céu: Embedded, Safe, and Reactive Programming. Technical Report 12/12, PUC-Rio, 2012.

[51] F. Sant'Anna, N. Rodriguez, and R. Ierusalimschy. Advanced control reactivity for embedded systems. Workshop on Reactivity, Events and Modularity (REM'13), 2013.

[52] F. Sant'Anna, N. Rodriguez, and R. Ierusalimschy. Structured Synchronous Reactive Programming with Céu. In *Proceedings of Modularity'15*, 2015.

[53] F. Sant'Anna, N. Rodriguez, R. Ierusalimschy, O. Landsiedel, and P. Tsigas. Safe System-level Concurrency on Resource-Constrained Nodes. In *Proceedings of SenSys'13*. ACM, 2013.

[54] R. Santos, G. Lima, F. Sant'Anna, and N. Rodriguez. Céu-Media: Local Inter-Media Synchronization Using Céu. In *Proceedings of WebMedia'16*, pages 143–150, New York, NY, USA, 2016. ACM.

[55] J. Sarik and I. Kymissis. Lab kits using the arduino prototyping platform. In *Frontiers in Education Conference (FIE), 2010 IEEE*, pages T3C–1. IEEE, 2010.

[56] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: A language and runtime system for perpetual systems. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 161–174, New York, NY, USA, 2007. ACM.

[57] E. Tychon, B. Schoening, M. Chandramouli, and B. Nordman. Energy management (EMAN) applicability statement. Technical report, IETF, 2011.

[58] S. Venkateswaran. AM335x SitaraTM processors reference manual. Technical report, Texas Instruments, 2014.

[59] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Ecosystem: Managing energy as a first class operating system resource. *SIGARCH Comput. Archit. News*, 30(5):123–132, Oct. 2002.

[60] Y. Zhu and V. J. Reddi. Greenweb: Language extensions for energy-efficient mobile web computing. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 145–160, New York, NY, USA, 2016. ACM.