Hybrid Apps: Apps for the Internet of Things

Daniel Yunge, Philipp Kindt, Michael Balszun, Samarjit Chakraborty
Technische Universität München (TUM) - Institute for Real-Time Computer Systems (RCS)
Munich, Germany
Email: {yunge,kindt,balszun,chakraborty}@rcs.ei.tum.de

***Abstract*—Smartphones have become popular mainly because of the large variety of applications they can run. In contrast, most devices in the phone's environment – e.g., household appliances or environmental sensors – are much less flexible because their functionality is hardcoded at the design time. In order to realize the vision of the Internet of Things (IoT), where all devices communicate with each other to realize joint tasks, it is necessary that these devices are able to extend and adapt their functionalities on-the-fly based on their surrounding. To realize smart functionalities for IoT devices, we propose "hybrid Apps", the concept of Smartphone "Apps" applied to small embedded systems. In contrast with current packaged "smart home" solutions, where all appliances have to be changed to their smart counterparts at the same time, hybrid Apps permit an incremental and hence feasible deployment of the IoT vision. In this paper we discuss the challenges and opportunities associated with this approach. We argue code interpretation as a candidate reprogramming method for IoT devices and analyzed its feasibility with real-world measurements of key parameters such as computational and energy overhead. While in general, code interpretation incurs a large energy-overhead, we show that for typical IoT applications executed every few seconds, it is as low as 1%.**

***Keywords*—Code Interpreter; Virtual Machines; Reprogrammability; Wireless Sensor Networks; Apps;**

I. Introduction

With the emerging IoT, everyday objects are expected to communicate with each other ubiquitously in the future. These devices form a system, i.e. they offer functionalities which none of the individual devices could provide on its own. For example, the heaters in an apartment can communicate with multiple temperature sensors to adjust the ambient temperature. In order to create even more useful applications, other kinds of devices can be included. For instance, heaters could also communicate with the smartphones of the persons living in the apartment in order to detect them and automatically raise the temperature whenever at least one person is at home, or lower it to save energy otherwise.

However, currently the only possibility to realize such a smart system is with a *packaged* solution, where a single manufacturer needs to provide all essential parts, i.e. thermostats, sensors and smartphone App. As a result, new applications that are developed after the system has been shipped and in which additional devices manufactured by third parties need to be included cannot be realized. For example, a smart LED light whose color is adjusted according to the temperature measured by the room sensors cannot be included seamlessly in the scenario above.

As a result, most applications that are tailored to specific setup of devices, or where their details are unknown at the design time are not feasible with packaged solutions. Monolithic architectures of the firmwares in todays IoT-devices and hard-coded functionalities prevent such flexible designs. The functionality of smart devices cannot be easily extended after the purchase, as in the case of smartphones with Apps that can be downloaded and installed from the internet. The only way users are able to modify the functionality of small embedded devices is applying sporadic updates offered by the manufacturers that are not customizable and are generic to all users. Or, less probably, cumbersome cross-compilation and physical flashing procedures which require the users to disassemble the device.

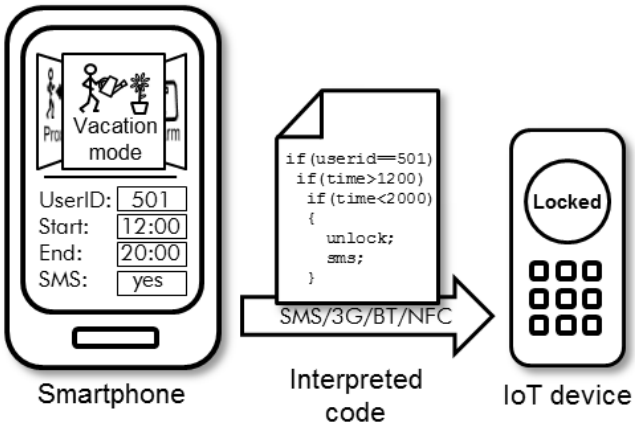

Fig. 1: Programming of a smart device by a smartphone

To overcome this lack of flexibility, we propose to add instantaneous reprogramming capabilities to smart devices by equipping them with a code interpreter. Using a smartphone App, users can create custom ubiquitous applications on their own in an intuitive fashion, and send them to the smart device, as show in Fig. 1. We call this novel design paradigm of IoT applications *hybrid Apps*. With *hybrid Apps*, custom applications that are specific to their contexts, setups and individual preferences can be realized. This works as follows.

- On the smartphone, an App is installed which contains a custom code generator for a specific functionality.
- This App guides the user to setup and customize its smart environment. Based on the results and the devices involved, custom code is generated for all smart devices which is denoted as a *microApp*.
- The generated code is then sent wirelessly to the IoT devices.
- All IoT devices that are part of the setup contain an optimized code interpreter which is capable of installing and executing the generated *microApp*.
- During operation, the smartphone App and the *microApp* exchange information and, if needed, further pieces of generated code.

Accordingly, a *hybrid App* is separated into two parts. The

one part is running as a mobile phone application. The other part is run on one or many smart devices in the form of interpreted code. An appropriate software architecture on the embedded system provides that the code can make use of the full functionality of the device, as long as the standardized interfaces offered by it (e.g., temperature sensing, light temperature adjustment, etc.) are known to the phone. As a result, the proposed technique takes advantage of the unique features of each device class. On the smartphone, powerful computational capabilities, a sophisticated Graphical User Interface (GUI) and access to the internet can be used. On the IoT devices, sensor data can be collected and processed, or the physical world can be influenced by actuators. Such applications collaborate seamlessly with each other without the need of having been designed together as packed solutions. As a result, new use cases can be realized easily in a convenient and intuitive fashion. We believe that this technique can be the key for ubiquitous computing and the Internet of Things becoming widespread in our everyday lives.

The primary objective of this paper is to introduce the concept of *hybrid Apps* as a vehicle for realizing applications in the IoT and to identify the associated opportunities and challenges. We evaluate the feasibility of the concept and analyze the requirements on the design of such IoT devices, along with the achievable benefits in detail. In addition, we propose a novel, preliminary system architecture that can realize the proposed concept. Since the execution of *microApps* imposes the need of code interpreters on computationally constrained devices to serve as lightweight Virtual Machine (VM)s, we present some preliminary results on the feasibility of such a system. With this paper, the authors attempt to motivate further discussions on this topic which will result in more concrete efforts to address the challenges involved.

The rest of this paper is organized as follows: Related work is presented in Section II. In Section III, the concept of *hybrid Apps* is described in detail. In Section IV, new opportunities that are opened up by our proposed approach are presented. Technical challenges towards the realization of this concept are discussed in Section V. Further, we analyze new possibilities that can be achieved with our proposed technique using a smart door lock as a running example in Section VI. In Section VII, we first present a demonstrative implementation of an embedded system which is capable of executing *microApps*. Next, preliminary results on the computational overhead of code interpretation and energy consumption based on real-world measurements on this implementation are presented. Finally, concluding remarks based on our results are presented in Section VIII.

II. Related Work

A seamless collaboration scheme for smart, connected devices as proposed requires a software layer between the hardware and the application space known as middleware. One important task of the middleware is to enable programming support aspects such as safe code execution and reliable code distribution among nodes [1]. In addition, the middleware must provide data and hardware abstraction. In this aspect, several approaches adapted to the resource constrained environment have been proposed such as database-based querying interfaces [2], mobile agents [3] or VMs.

Even with the high execution overhead involved, VMs have proven to be effective to achieve a scalable development of Wireless Sensor Networks (WSN) [1]. Accordingly, application-specific VMs [4] as well as generic VMs with reduced instruction sets have been developed successfully for constrained embedded systems. The most well-known generic VM is Java. Due to its many features like thread support, safe execution and a compact code format, many authors have developed modified versions of Java to reduce its hardware requirements. These modifications limit the set of supported instructions in order to run it on resource-constrained embedded devices. Accordingly, the open-source project Squawk [5] applies an off-line adaptation procedure at compile-time to reduce the memory footprint of the standard Java bytecode. For example, on an ARM7 architecture, Squawk requires 270 KB of flash and 80 KB of RAM. Squawk was used in the SPOT [6] device, an early attempt of *Sun*, to introduce their language on wireless devices. Much later, in 2013, Oracle released *Java ME Embedded* for ARM Cortex devices, which is based on the *Connected Limited Device Configuration* (CLDC) instruction set. CLDC specifies a minimum of 160 KB of non-volatile memory for the VM and CLDC libraries, and 32 KB of RAM. An even smaller implementation of Java is Darjeeling [7], which offers lightweight thread support, a garbage collector, and exception handling on devices having 2-10 KB of RAM.

However, in all cases, an off-line process is required to obtain the intermediate bytecode used by the Java VM. While there exist several open-source Java compilers which could be included within a Smartphone App to generate code, eliminating intermediate steps carries the benefit of permitting more constrained devices also to generate code quickly and independently, for example by using code templates. As a consequence, IoT devices could send code to Smartphones to generate a GUI dynamically, for example, or reconfigure each other, as shown in Section VI. Accordingly, in order to simplify the code generation process, we propose the use of code interpreters along with a middleware platform. Such interpreters exist and have been implemented for resource-constrained embedded devices with several programming languages such as C, BASIC [8], Scheme [9], Python [10] and Lua [11]. The lightweight C interpreter PicoC [12] is an open-source project, easy to be ported and suitable for embedded applications. Because of its simplicity and small implementation, we chose it for our case-study. As described in Section VII-A, it fits into less than 40 KB of flash, only needs a few KB of RAM, and is therefore well suited to be used on memory-constrained devices.

III. Hybrid Applications

In this section, we describe the concept of *hybrid Apps* in detail. As can be seen in Figure 1, the necessary elements to build such a system are a smartphone, a smart device and a common communication protocol. On the smartphone, an App is responsible of providing the GUI to enable the user to define new functionalities for the smart device. On the smart device, a middleware and basic routines related to the hardware characteristics are available. Some basic functionalities are fixed and directly accessible using the respective communication stack, e.g. Bluetooth Low Energy (BLE) GATT services, as any other traditional remote device. In addition, a special *code interpreting* GATT service is used as interface to send and hypervise custom code. This scheme can easily be adapted to

other communication protocols, e.g. instead of GATT services, access to basic functionalities and the hypervisor would be realized in the case of the IP protocol using different TCP/UDP ports. In the following, the main building blocks of a *hybrid App*, viz. code generation, code transmission and code interpretation are described in detail.

Code generation: The first step towards executing a hybrid application is the installation of a conventional App to the smartphone. This smartphone application makes use of a predefined infrastructure to interact with its surrounding smart devices, thereby expanding its functionality towards a *hybrid App*. This interaction works as follows:

1) The smartphone discovers its surrounding devices using a standardized wireless protocol (e.g., BLE). As a part of this discovery process, it determines which capabilities are available on the surrounding nodes. Each capability is advertised by the IoT appliance using a Universally Unique Identifier (UUID) which corresponds to a predefined standardized Application Program Interface (API), which it makes available to interpreted code. Using this API, interpreted code which is executed on the embedded system can access hardware in an abstract fashion. As an example, a smart door lock could advertise a service for locking or unlocking a key cylinder and another one for cryptographic functionalities.

2) Once available devices have been identified, a configuration and setup wizard is shown to the user. A GUI guides the user through the configuration and customization of the desired functionality in a convenient and intuitive fashion. A list of predefined functionality templates can be downloaded from a repository.

3) After the completion of the configuration, code for the IoT appliances is generated in a standardized language and sent wirelessly to them. As a result, the code can be executed on any device without the need for the smartphone to be aware of the device details as long as the offered APIs are known.
In addition to the code transfer service, the smart device also provides an interface for controlling a basic hypervisor. By using it, users can remove installed Apps from their devices or adjust access privileges and configuration options.

Code transmission: Once the custom functionality is synthesized as code, it is sent to the smart devices using a wireless communication protocol supported by the smartphone like Bluetooth (BT), BLE, WiFi or Near Field Communication (NFC). So, the code can be delivered to the smart device in a standardized fashion, given that a common interface for the installation procedure is available on the device. On top of this protocol, a *hypervising* protocol must exist to allow both devices to negotiate a set of rules regarding the following aspects:

- How long may the *microApp* remain installed on the device?
- What triggers its execution? (Persistent execution, event-triggered execution, periodic execution, etc.)
- Which device is allowed to send code and which services may be used by that code? How many resources may be used by the device?

Once the code is transmitted, the code can be executed. This step is described next.

Code interpretation: As already mentioned, functions related to communication and hardware access can be delegated to the middleware. For example, it is responsible of interrupting the interpreter's execution to attend the arrival of new communication packets or read a new sample of the ADC. By using a multitask-capable OS as middleware like FreeRTOS [13], several instances of the interpreter can be executed dynamically and operated as concurrent tasks, as shown in Fig. 2. In addition to benefits like concurrent and supervised execution of *microApps*, this also enables multiple users to allocate their custom functionalities on a single smart device. In which case, shared resources accessed through the API should be safely administered by the OS in order to avoid race conditions. Nevertheless, safe operation of *microApp*s can be achieved even without a sophisticated OS and can be supervised by a dedicated interrupt service routine (ISR) executed periodically, for example. In case of the detection of long execution times, or if any other operation rule is broken, the state machine that governs the interpreter can be reset.

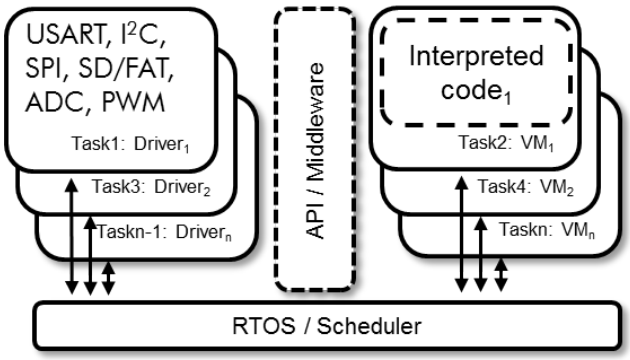

Fig. 2: Virtual platform inside of an IoT device

With the scheme described above, many new interaction possibilities between user and smart devices show up. Some of them are described in the next section.

IV. Benefits of Hybrid Apps

In this section, we provide a detailed analysis of possible benefits and describe the main advantages of our scheme. While these advantages are presented in a generic fashion in this section, we provide concrete benefits of a realistic use-case in Section VI.

Flexibility: The most important benefit of *hybrid Apps* is the capability of changing the functionality of smart devices quickly and easily. This can be seen as a method to optimize memory use, since an unlimited amount of functionalities can be implemented on the device that has the capacity for just a few. In addition, flexibility allows the adaptation of IoT devices to new usage scenarios, deployment conditions, communicating parties or operation schemes on-the-fly. For instance, a lightweight compression algorithm can be installed on all smart devices of a network as *microApps* in order to reduce the communication bandwidth.

Specificity: Data generating devices like sensors can be instructed to be more specific considering the final purpose of data. For example, a temperature sensor that originally broadcasts raw values can be programmed alternatively to record daily maximal, minimal and average values if the data is for statistical purposes, or can be programmed to communicate

with the room's heater controller only when a significant difference in the temperature is detected, thereby saving link bandwidth and energy.

Decentralization: Combined functionalities can be realized without a central decision maker like a smartphone or the Cloud. As mentioned in the introduction, several appliances of different manufacturers can be programmed on-the-fly to communicate and execute a collaborative function. For example, the office's air extractor can be linked with the window opener and the temperature sensor to control the quality of air. Another case of decentralization is that smart devices can be instructed to alternate from smartphone-supervised mode to operate autonomously, e.g. if the host needs to reduce its workload or is absent for some time.

Disposable code: *Hybrid Apps* can be utilized to execute *disposable* code. Particularly, a *microApp* can be installed on a target device to execute a set of tests, generate results and send them back to the phone. Afterwards, the *microApp* is eliminated. For example, to determine the communication QoS, a QoS *microApp* is sent to a target device before a specific testing byte sequence. Once there, the QoS App gathers packet statistics to determine the packet loss ratio. Similarly, a calibration procedure can be run to write offset and gain parameters into the non-volatile memory, and then be eliminated.

Platform independence: As mentioned before, the benefit of using interpreted code is that the same *hybrid App* can be executed on different smart devices similarly to conventional Apps that can be executed on any compatible smartphone model. This enables software development for an increasing number of heterogeneous smart devices without the inconvenience of creating multiple versions of the same program.

V. Technical Challenges

As already mentioned, the concept of *hybrid Apps* imposes a couple of technical challenges which need to be resolved. In this section, we identify the most important challenges towards the realization of *hybrid Apps*. These are the following.

Hardware constraints: One of the most significant challenges is the development of a general-purpose VM that fits into a memory-constrained Microcontroller (MCU). Further, because the device operates from battery power, the VM's execution must also be energy-efficient. Moreover, each of the aspects described below, e.g., security or interoperability, increases the complexity of the VM as well. This imposes the need for MCUs with more resources, which are more complex and power-hungry.

Security: Security on IoT is increasingly becoming a concern, since internet-connected embedded devices can be hacked to be used for malicious network activities such as remote attacks, spamming or the theft of private information. Accordingly, enabling arbitrary code execution on millions of such devices will turn them into desired targets of hackers. For this reason, one big challenge is balancing security measures with memory utilization and computational overhead while keeping the execution platform flexible. Apart from external risks, certain internal functionalities of the device need to be protected from any access by the interpreted code. Buggy or malicious programs might crash the device or even cause physical harm if they have unrestricted access to the hardware.

Interoperability: In addition to hardware, communication mechanisms must be abstracted as well from virtual processes in order to support them with simple message exchange interfaces, thus enabling collaborative behavior. Moreover, efficient methods must be developed to ease the compatibility verification and consequent collaboration of any two heterogeneous devices. This compatibility must be maintained without internet connection nor extensive databases. Accordingly, lightweight identification concepts which are state-of-the-art in communication protocols (e.g. attributes and services in Bluetooth) can be extended or included for this purpose. Finally, important aspects of *microApps* must be defined, such as installation procedures, their lifespans, hardware access privileges and all other aspects which are important for a seamless interaction.

Multiuser support: Many users should be allowed to send *microApps* and use a single smart device concurrently, if this has enough free computational resources in a certain moment. However, this imposes further requirements. First, sessions and execution contexts for different users must be separated from each other to ensure security and data integrity. Second, users should be identified, authenticated and authorized in order to get access. Third, simultaneous sessions require a careful coordination to allow them accessing shared resources without the risk of deadlocks.

Resource management: Control mechanisms of the CPU utilization are required to prevent computationally-intensive *microApps* from unfairly utilizing the device's processing and energy resources. Accordingly, the smart device's CPU utilization needs to be supervised constantly. In addition, rules, quotas or black lists might be necessary to avoid abuse.

VI. Use case: Smart Door lock

To emphasize the novel possibilities for IoT devices that can be realized with *hybrid Apps*, we present the benefits of a smart door lock as a concrete example. Smart door locks which can be locked- or unlocked by a smartphone using cryptographic keys have recently been commercialized [14]. If a future door lock would make use of the concept of *hybrid Apps*, a large variety of additional functionalities could be realized. Below, we describe a selected choice of these advanced functionalities. While a few of them could also be realized with conventional techniques, the richness of different applications cannot be achieved without our proposed concept due to the following reasons. First, the manufacturer cannot foresee and implement all possible use cases at the design time, especially if other devices from third-party manufacturers are to be involved. Second, all of these functionalities would not fit into the memory of the lock at once. This imposes the need of *hybrid Apps*. In particular, examples for novel features that can be realized with our proposed concept are as follows.

Vacation service: Because the user is on vacation, he/she programs the door lock to allow a colleague to enter the office only for two times to water the plants, as shown in Fig. 1. After the second time the colleague enters, the permission is withdrawn automatically and the *hybrid App* is destroyed.

Proximity unlocking: The user want to avoid locking and

unlocking the door manually every time he/she left the office. Accordingly, the user sends a *hybrid App* to the door lock to unlock it automatically when detects his/her smartphone, and is locked similarly once it is absent.

Pincode unlocking: The door lock sends a *hybrid App* to the smartphone to show a keypad GUI that can be used to enter a pin code in combination with some certificate on the phone to grant access, thus increasing the security. For example, a PIN code could be required only on phones which are unknown to the lock, e.g. on phones with one-time keys.

Notification/Mailbox: The user will attend a meeting for the next hour. Therefore, he/she sends a *hybrid App* to the lock to return a message if someone attempts to enter to the office in his/her absence. Similarly, visitors can leave messages like on a mailbox by receiving a *hybrid App* with an appropriate GUI from the lock.

Interoperation with smart buildings: If the lock is used in an environment with smart sensors forming a wireless network, the lock can communicate with smoke- and movement detectors. In case of a fire, doors are unlocked automatically and the fire brigades can enter all rooms without the need to break any doors.

Further functionalities can be invented, especially if the smart door lock has access to the internet via a wireless gateway. In summary, the possibilities and functionalities of the smart door lock can be greatly extended with a significant added value for the owner.

VII. Evaluation

As presented, *hybrid Apps* can add significant benefits to IoT-devices and hence their realization is desirable. However, can they be implemented with todays available technologies? Many embedded processors that are used as wireless sensors or actuators run on batteries and are equipped with limited amounts of flash and RAM. In this section, we evaluate whether *hybrid Apps* can be realized on these constrained systems using real-world measurements. For this purpose, we implemented a preliminary architecture for executing *microApps*, which is described below.

A. Preliminary setup

As a proof of concept, we realized a reprogrammable smart device by applying a code interpreter as a virtual platform. We used PicoC [12], an open-source lightweight C-code interpreter, as mentioned in Section II. We ported PicoC bare-on-the-metal, i.e. without underlying RTOS, to a STM32F401RE Nucleo board with an ARM Cortex-M MCU. The embedded platform has 512 KB of Flash memory and 96 KB of SRAM. The measurements were made with the internal clock configured to run at 8 MHz and 84 MHz. The code was sent using PicoC's interactive mode through the MCU's USART. This system was used to perform benchmarks that evaluate the feasibility of our proposed concept, as described in the next section.

B. Memory Demands

With the setup described above, we have shown that *microApps* can be realized on embedded systems. However, important questions need to be answered to quantify the overheads and requirements in terms of memory and processing. Towards this, we have measured the memory- and flash demands of PicoC. These results are shown in Table I.

TABLE I: Benchmarking results of Programs 1-5

	PicoC			Compiled		
	Code [Byte]	RAM [Byte]	Exec [ms]	Binary [Byte]	RAM [Byte]	Exec [us]
Init	.dat:3K .txt:36K tot:39K	.bss:2K+ memVM St:364	wkup 8/84MHz .15/.19	.dat:8 .txt:2K tot:2K	.bss:0 St:64	wkup 8/84MHz .15/.19
Prog	Code chrLen [Byte]	RAM mVM+St [Byte]	Exec 8/84MHz [ms]	Binary indiv/total [Byte]	RAM Stack [Byte]	Exec 8/84MHz [us]
1	94	6.3K+1.0K	3.2 /.30	40 /2.6K	72	.75 /.07
2	106	6.4K+1.2K	3.5 /.33	48 /2.8K	80	26 / 3.0
3	1.1K	9.8K+1.7K	7.2 /.68	212 /3.0K	112	3.9 /.37
4	116	7.1K+1.6K	115 / 11	64 /2.8K	88	46 / 4.4
5	196	7.8K+1.8K	1.1K/104	168 /3.0K	496	1.3K/120

Benchmark programs: In the following, we attempt to measure this overhead by benchmarking the compiled and interpreted versions of five different programs. As an example of a time triggered *microApp*, Program 1 reads the state of a button, then toggles a digital output and puts the MCU into sleep mode for 1 second. Program 2 wakes up with a button press, then evaluates a converted value of the ADC to realize a simple if..then..else construct as a demonstration of an event triggered *microApp*. In addition, Program 2 activates a digital output depending on the ADC's value and then enters into sleep mode. Program 3 executes a matrix-based state-machine that controls a digital output and uses a button's state as transition parameter. Finally, in order to evaluate more computationally intensive algorithms, Program 4 calculates the 40th number of the Fibonacci series, whereas Program 5 calculates all prime numbers below 100.

Memory requirements during startup: Table I shows that the PicoC runtime along with hardware libraries and their respective APIs needs 39 KB of flash. The initial reserved RAM, observed under the .bss section, requires 2 KB plus the memory size assigned to picoc's *virtual* stack and heap (memVM). The minimum size for the memVM is 5.8 KB, to get the API and C libraries loaded. Further, the MCU's stack takes up to 364 Bytes during the system's start-up. In contrast, the compiled system required a total of 2 KB of flash, it does not require any initial reserved RAM and the stack grew up to 64 Bytes during the initialization process, only. Stack was directly observed using a debugging tool.

Memory requirements during execution: During the benchmarking process, interpreted programs required up to 9.8 KB for the memVM to run (including API and C libraries), whereas the MCU's stack never exceeded 1.8 KB. On the other hand, compiled programs required between 72 and 496 Bytes for the stack, only.

Code size: In order to compare interpreted and compiled code size, we contemplated three remote reprogramming scenarios: code interpretation, partial binary replacement and total binary replacement. For interpreted code size (denoted as *chrLen* in Table I), function and variable names were set 3 characters long, and all dispensable characters like spaces and carry-returns were eliminated. For the second scenario, partial binary replacement, only the individual size of the tested function without libraries was considered (denoted as *indiv*), whereas for the third scenario, total binary replacement, the size of the whole program with libraries was considered (denoted as *total*). Accordingly, sending interpreted code requires slightly more bytes than sending the binary of an individual function, but significantly less than the whole firmware.

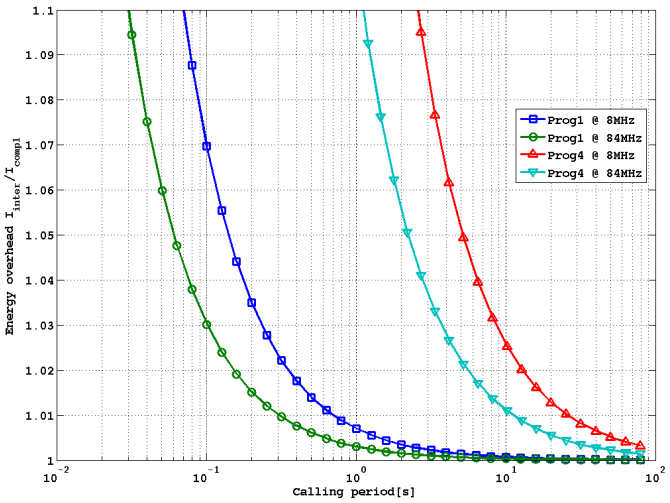

Fig. 3: Relative energy overhead of interpreted Program 1 and Program 4 at 8 MHz and 84 MHz.

C. Computational overhead

We have shown that currently available interpreters have memory footprints which are small enough to fit on most embedded devices. However, another important aspect is the computational overhead of such interpreters. To quantify this, we measured the execution time as compiled and interpreted code of the 5 benchmarking programs described previously. Table I shows the results of this comparison for CPU clocks 8 MHz and 84 MHz. The time measurement procedure consisted of setting and clearing a digital output around the measured program, and capturing the delay with an oscilloscope. As a conclusion, the overhead in terms of execution time is significant. In the case of *Program 1*, the interpreted code takes up to 4300 times the execution time of its binary counterpart, whereas for *Program 4* the overhead is 2500. However, this is not conclusive regarding our concept's feasibility, due to the analysis presented next.

D. Energy consumption

The incurred execution overhead unavoidably leads to inefficient energy use, an unwanted situation in battery-powered devices. However, since duty-cycle of many IoT applications is very small, overall consumption is dominated in these cases by the MCU's sleep-mode current, damping the inefficiency problem with it. In order to evaluate this effect, we measured the sleep-mode current of our platform (1.8 mA), as well as its active current under two CPU frequencies, 8 MHz (5.8 mA) and 84 MHz (20 mA). In addition, we measured the wake up times at both speeds (shown in Table I under *wkup*), which have not taken part of the benchmarking results (since execution times were measured after the waking up process), but were used for energy estimation purposes. During this period, which is associated with clock setting and peripheral configuration procedures, currents between 6.0 mA and 6.5 mA were measured. Finally, using a MATLAB script, the MCU's mean current when running compiled ($\bar{I}_{compl}$) and interpreted ($\bar{I}_{inter}$) versions of Program 1 and Program 4 were estimated for both frequencies and different calling intervals t_{call}. As a result, Fig. 3 was generated, where the energy overhead expressed as $\bar{I}_{inter}/\bar{I}_{compl}$ is shown.

In terms of consumption, the energy overhead per execution becomes irrelevant beyond a certain program calling interval, which is applicable to many IoT applications. As Fig. 3 shows, the energy overhead is less than 1% for an application like Program 1 that is executed periodically with an interval longer than 0.3 seconds (round markers), or Program 4 after 10 seconds (inverted triangles). For applications that are triggered by rare events, like Program 2 or those running on the smart door lock, energy overhead of code interpretation is even more negligible.

VIII. Concluding Remarks

In this paper we presented the concept of *hybrid Apps*, which consist of a conventional smartphone App and a *microApp* which is run on an embedded system. Even though significant overheads in terms of execution time and memory footprints exist both for the code interpreter itself and for the interpreted programs, it is feasible to realize such architectures on small embedded devices. However, use cases such as the smart door lock require even smaller MCUs, such as the widely used CC2540 BLE SOC [15] with only 8 KB of flash. Further research should be performed to reduce these overheads in order to realize *hybrid Apps* also on MCUs as constrained as the CC2540. In this paper, we also presented further challenges associated with a full realization of *hybrid Apps*. We would like to encourage further research on the problems described, since *hybrid Apps* could provide a major step towards the vision of the Internet of Things becoming widespread.

Acknowledgment

This work was partially supported by HE^{2}mT - High-Level Development Methods for Energy-Saving, Mobile Telemonitoring Systems, a project funded by the Federal Ministry of Education and Research of Germany (BMBF).

References

[1] S. Hadim and N. Mohamed, "Middleware for wireless sensor networks: A survey," in *Comsware 2006*, 2006.

[2] R. Mueller and J. Rellermeyer, "A dynamic and flexible sensor network platform," in *SIGMOD '07*. New York, NY, USA: ACM, 2007.

[3] T. Paul and K. Stanley, "Data collection from wireless sensor networks using a hybrid mobile agent-based approach," in *LCN 2014*, 2014.

[4] P. Levis and D. Culler, "Mate: A tiny virtual machine for sensor networks," *SIGARCH Comput. Archit. News*, vol. 30, no. 5, Oct. 2002.

[5] D. Simon and C. Cifuentes, "Java on the bare metal of wireless sensor devices: The squawk java virtual machine," in *VEE*, 2006.

[6] The sun spot platform home page. [Online]. Available: www.sunspotworld.com

[7] N. Brouwers, K. Langendoen, and P. Corke, "Darjeeling, a feature-rich vm for the resource poor," in *SenSys*, 2009.

[8] Parallax basic stamp. [Online]. Available: https://www.parallax.com/catalog/microcontrollers/basic-stamp

[9] M. Huber. Armpit project. [Online]. Available: armpit.sourceforge.net

[10] Python-on-a-chip project. [Online]. Available: code.google.com/p/python-on-a-chip

[11] Elua project. [Online]. Available: www.eluaproject.net

[12] Z. Saleeba. Picoc project homepage. [Online]. Available: code.google.com/p/picoc

[13] Freertos homepage. [Online]. Available: www.freertos.org

[14] Kevo smart lock. [Online]. Available: www.kwikset.com/kevo

[15] Texas Instruments, "CC2540 datasheet - 2.4-GHz Bluetooth low energy System-on-Chip," 2013.