COIN: Opening the Internet of Things to People's Mobile Devices

Maria Laura Stefanizzi*, Luca Mottola+, Luca Mainetti*, Luigi Patrono*
*University of Salento (Italy), +Politecnico di Milano (Italy) and SICS Swedish ICT

***Abstract*—People's interaction with Internet of Things (IoT) devices such as proximity beacons, body-worn sensors, and controllable light bulbs is often mediated through personal mobile devices. Current approaches often make applications operate in separate silos, as the functionality of IoT devices is fixed by vendors and typically accessed only through low-level proprietary APIs. This limits the flexibility in designing applications and requires intense wireless interactions, which may impact energy consumption. COIN is a system architecture that breaks this separation by allowing developers to flexibly run a slice of a mobile app's logic onto IoT devices. Mobile apps can dynamically deploy arbitrary *tasks* implemented as loosely-coupled components. The underlying run-time support takes care of the coordination across tasks and of their real-time scheduling. Our prototype indicates that COIN both enables increased flexibility and improves energy efficiency at the IoT device, compared to traditional architectures.**

I. INTRODUCTION

The rise of smartphones and tablets makes them the established means for people to interact with Internet of Things (IoT) devices such as proximity beacons, body-worn sensors, and controllable light bulbs. A multitude of applications employs personal mobile devices to allow people to control, interact, and query sensors and actuators in the environment or on the human body. Technologies such as Bluetooth Low Energy (BLE), now commonly found both on personal mobile devices and IoT ones, are key facilitators for this trend.

Motivation. Architectures applied in these applications, however, treat the IoT devices as immutable black-boxes and operate in separate silos. Mobile apps are sometimes re-routed to intermediate gateways. Whenever direct access to the IoT device is allowed, the latter typically offers application-agnostic APIs that mainly enable extracting raw data and/or controlling basic actuator functionality. Changes to the on-board software are limited to firmware updates released by manufacturers to patch bugs or security flaws.

Such a state of affairs entails that: *i)* mobile apps are developed based on *vendor-specific APIs*, preventing portability; *ii)* even the simplest functionality requires *intense wireless interactions*, affecting energy consumption; and *iii)* app functionality are limited to the *time of wireless connection*, that is, disconnected operations are fundamentally hampered.

Unlike current practice, many foresee the interaction between humans and IoT devices to happen over an open, vendor-independent programmable substrate that enables a multitude of apps to co-exist, similar to smartphones and tablets. For example, an IoT device with body temperature and heart rate sensors may serve both fitness apps to track an individual's well-being and smart-health apps that communicate vital parameters to doctors. Individual building blocks necessary to realize these functionalities are often already available, yet a system architecture that blends them together in a working realization is arguably still lacking.

COIN. We bridge this gap by designing a system architecture that allows IoT devices to: *i)* run an arbitrary slice of a mobile app's logic in an on-demand fashion, and *ii)* host application data according to programmer-provided criteria, independent of the connection to user devices.

The problem is unique in many respects. Unlike traditional sensor networking, for example, applications are supplied by third parties. Their characteristics, such as processing and memory requirements, are difficult to anticipate. Multiple applications may need to operate concurrently, that is, not simply run side-by-side, but be able to exchange data with other a priori unknown apps. Processing is also expected to be largely event-driven; for example, being dictated by connections of mobile devices, rather than occurring periodically.

COIN rests upon two pillars: a custom programming model and a dedicated run-time support. The former is based on a notion of lightweight *task* as a programmer-defined relocatable slice of mobile app logic. In a fitness scenario, for example, programmers may define a task that computes burned calories based on the sensors available on fitness trackers. The mobile phone may opportunistically deploy such a task on the fitness tracker to limit data exchanges to a single quantity rather raw data. Multiple tasks on an IoT device can interact in a loosely-coupled manner, based on an actor-like [1] model.

COIN's run-time support accommodates existing building blocks to efficiently implement the required semantics. A message broker mediates interactions across tasks, while their executions are scheduled using an Earliest Deadline First (EDF) policy. Dynamic deployment of tasks is supported using a Virtual Machine (VM) developers plug in. Although COIN is independent of the underlying hardware platform, our prototype targets Cortex M microcontrollers (MCUs) and BLE radios, representative of target applications where energy budgets are as small as a COIN-cell battery.

We describe the design and implementation of a pervasive game using COIN, an application otherwise unfeasible with comparable features using vendor-specific architectures. We also report on the performance of COIN in energy consumption and execution times. The results indicate that the energy savings enabled by reducing wireless interactions through device-

local processing overcome the cost of code interpretation, validating our design choices. The price to pay are larger execution times, yet the values we obtain are not expected impact the application responsiveness.

COIN's main contributions are therefore: *i)* to increase the flexibility in the design of IoT applications by giving developers the ability to relocate slices of a mobile app's logic onto IoT devices, and *ii)* to improve the energy efficiency at the IoT devices in applications where real-time requirements are soft or absent. The remainder of the paper describes how we concretely achieve these contributions.

II. State of the Art

We report on application scenarios, requirements to overcome current limitations, and existing functionality.

A. Applications

Employing personal mobile devices as people's interface to the IoT yields applications with distinct characteristics, exemplified next.

Body sensor apps. Body-worn devices with physiological or inertial sensors are often used to monitor physical parameters, for example, temperature and heart rate. The raw sensor values are streamed to a smartphone, which acts as the sole processing unit. Similar architectures tend to be inefficient. Frequent wireless interactions between smartphones and sensors are costly in energy consumption. Algorithms exist to relocate part of the processing closer to the sensors. Examples are found in activity recognition and electrocardiogram analysis (ECG) [2]. The amount of data to transmit thus reduces, and so energy consumption improves. Flexible system support on the IoT device is required to employ these algorithms in a vendor-independent and reconfigurable manner.

Immersive computing. Interactions between people's devices and IoT ones need not be continuous, but simply occur opportunistically whenever the two are in range. Representative examples are pervasive games [12], where embedded devices are hidden in the environment to bridge the game's virtual world with the physical reality. These devices are often used as environment-immersed data stores to handle information relevant to the game plot. Access to digital information is dictated by physical location, enhancing the experience. Pervasive games are currently installed using dedicated hardware, deployed solely for running the game and later removed. Reusing already installed hardware is generally not possible, as it misses the necessary programming facilities.

Monitoring and tracking. The continuous interactions between mobile and IoT devices may break also because the latter are mobile. In supply chain applications [7], sensors are attached to packages to log information such as temperature and vibrations during transportation. When a package enters a warehouse, locally stored information are uploaded to the mobile phones of the warehouse personnel for inspection. In this scenario as well, the ability to store and process sensor data on the IoT device independent of the connection to a person's device is fundamental. Right now, such degree of decoupling can only be realized with one-off application-specific implementations.

B. Requirements

System architectures at the IoT device that overcome these limitations should fulfill several requirements:

- **Device-local processing:** running application-specific functionality on the IoT device decouples its operation from the mobile device and allows one to reduce wireless interactions, saving energy.
- **Data persistency:** the IoT device must be able to retain application data according to programmer-provided criteria independent of the connection to a mobile device, enabling disconnected operations.
- **Dynamic deployment:** the logic at the IoT device may not be known beforehand, but be provided on-the-fly by the mobile device; the run-time support at the IoT device must accommodate this need.
- **Real-time scheduling:** independently-developed applications may need to coexist on the same IoT device; the system must ensure that their real-time requirements are fulfilled whenever possible.
- **High-level programming and portability:** application functionality for the IoT device must be developed using high-level languages and not require increased efforts to adapt to different hardware.

In contrast, current applications are normally developed with a "sense-and-send" design. IoT devices are employed as shipped by manufacturers, that is, with pre-loaded firmwares that only enable low-level interactions. As a result, the entire application logic executes at the mobile device and is encoded in a vendor-specific manner. Besides not enabling any disconnected operation, these designs decrease portability, consequently increase development efforts, and are detrimental to energy consumption.

C. Building Blocks

Approaches exist that address specific issues in the scenarios we target; for example, in the field of operating systems (OSes) for sensor nodes, VM technology for resource-constrained devices, and interoperability frameworks.

Sensor network OSes such as Contiki and LiteOS offer dynamic linking capabilities, which allow different applications to be added or replaced at run-time. However, the OS per se does not provide a programming model that allows dynamically-deployed applications to discover each other and exchange data. Differently, components must be developed based on how they bind to already running components, which requires intimate knowledge of the latter. Most importantly, OSes in this area offer low-level programming interfaces based on languages such as C, which contrasts with the modern development tools available for mobile apps.

VMs for resource-constrained devices retain the ability of dynamic code deployment while offering hardware independence and higher-level programming languages, such as Java

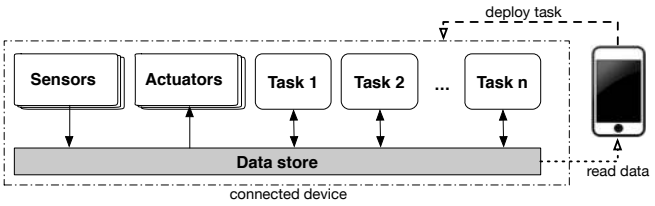

Fig. 1. Tasks interacting in a loosely coupled manner.

```
1 def foo(x):
2     # do something...
3
4 def boo(x):
5     # do something else...
6
7 startTask()
8 y = foo(inputData)+boo(inputData)
9
10 output(y)
```

Fig. 2. Example task code.

or Python. These aspects motivate us to base COIN on VM technology. The cost of code interpretation is the price we pay to facilitate the development process. Because of the rise of energy-efficient 32-bit MCUs, such as ARM's Cortex M series, we demonstrate this represents an effective design point.

The design of COIN is *orthogonal* to the specific VM one plugs into the architecture. Maté [10] offers a custom language that can be tailored to specific application domains. TakaTuka [3] and Darjeeling [5] provide Java VMs for 16-bit MCUs, whereas Squawk [14] targets higher-end devices. DAViM [8] focuses on isolating multiple applications from each other. In general, the design of embedded VMs targets efficient code interpretation, without providing dedicated abstractions for coordinating concurrent third-party applications.

The heterogeneity of IoT devices motivates efforts in interoperability frameworks and reference architectures. Examples are AllJoyn and IoTivity. These ease development by defining vendor-agnostic APIs for applications to inter-operate, based on IoT devices with much greater resources compared to ours and without providing the ability to relocate parts of the application logic. Efforts such as IoT-A, SENSEI, and OpenIoT provide open architectures to facilitate application development via semantically-interoperable interfaces. These are complementary to COIN, which may help realize more flexible or efficient implementations exported through the same interfaces as in these architectures.

III. COIN ARCHITECTURE

COIN revolves around a dedicated programming model and a run-time support to implement the required semantics. We only provide a few highlights here, and refer the reader to a companion technical report [11] for details.

A. Programming Model

We design COIN's programming model based on the characteristics of mobile apps where relocating a slice of the logic onto the IoT device may achieve benefits such as better designs or improved performance, as we exemplified in Section II. On the other hand, we do not target applications where IoT devices need only be equipped with application-agnostic functionality; for example, in case they are used as "beacons" in the environment for indoor localization.

The key feature in COIN's programming model is the shift of interactions to data rather devices. At the core of this is a notion of *task* as a relocatable slice of app logic.

Decoupling. To facilitate interactions among third-party functionality, tasks are fully decoupled. They cannot share global data and only interact asynchronously in an actor-like fashion [1]. Data exchanges occur through a single abstract data store, as in Figure 1. This spares the need of dynamically reconfiguring the bindings among tasks as these come and go, and enables data-driven discovery of available functionality.

Tasks are completely defined by the data types they consume or produce. Information on these are included in a *manifest* deployed with the task. The data types are specified as named data structures. The manifest of the task deployed by the fitness app, for example, may indicate input types that include acceleration and blood pressure as double precision numerical values, and outputs such as burned calories as integer values. Existing sensor data models [4] can be applied to uniform naming and format.

Execution. Figure 2 shows a code snippet for a simple task using the Python syntax, yet the programming model is independent of the specific language.

Task execution is reactive, and triggered only by the availability of any of the input data types. For example, we discourage the use of long-running threads whose fair scheduling may become difficult. Input data is made available using a dedicated API, which also provides operations to output the results and to indicate the start of processing, required to simplify Python's modularity model. An example of the former is the **`output()`** function in Figure 2. This API is the only interface developers employ to write tasks; other than this, developers can encode arbitrary application logic.

The input data of a task may come from the sensors aboard the device, or be the result of a different task. In the former case, sensors are automatically probed according to the required input rates specified in the manifest. For example, a health-monitoring app may employ the burned calorie information of the fitness app to augment the long-term time analysis. Such a data-driven programming facilitates developing vendor-independent interaction paradigms.

Execution of tasks is also decoupled from the connection to a person's mobile device. Tasks may, for example, reside on the IoT device also whenever the mobile device that originally deployed them moves away. Unlike the traditional actor model [1] where data is lost if no actor immediately consumes it, COIN applies persistency to data as well. Data resides on the IoT device according to programmer-defined criteria, such as a given time interval, catering for the needs of immersive mobile computing applications [12].

B. Run-time Support

COIN's run-time support includes three components: *i)* a data broker to mediate task interactions, *ii)* a scheduler to regulate task execution, and *iii)* a VM layer.

Broker and scheduler. The *broker* matches data producers and consumers based on data types. Whenever a match is identified, the consumer task is handed over to the scheduler. In our prototype, the broker maps items in the data store to BLE *characteristics* to give mobile devices standard-compliant access to data.

If multiple tasks consume the same data type, the match happens simultaneously. The *scheduler* thus implements an Earliest Deadline First (EDF) policy. Information on the absolute deadline of a task and its expected execution time are part of the manifest. Static analysis tools and emulators can be used to estimate the latter. The scheduler also ensures that every task runs to completion; concurrent events, such as connection requests from other mobile devices, are postponed until the task finishes.

We choose EDF because of its real-time optimality: if a schedule able to meet all task deadlines exist, EDF finds one. The processing overhead of EDF is no issue in our setting, also because we do not expect a large number of tasks to be triggered simultaneously. As tasks should be short-lived, running them to completion does not pose problems, as in architectures with similar design rationales.

Virtual machine. We port PyMite, a reduced Python interpreter, as VM layer. COIN is independent of the language to write tasks, but we choose Python for several reasons. Compared to languages such as Java, its implementation on embedded devices is less limited; for example, PyMite retains the support to multiple programming paradigms, including object-oriented and functional. Moreover, Python directly compiles to bytecode, which reduces network traffic when deploying tasks. The most complex application we tested so far yields slightly more than 1 KB of bytecode.

We map COIN tasks to PyMite threads, which requires adapting the latter along multiple dimensions. We replace the built-in round-robin scheduler with EDF. In the original PyMite, the state of a thread is lost when the execution exits; we thus extend the VM to maintain the thread state across executions of the same task. Finally, we choose to save the precious RAM segments and store the Python bytecode on flash memory, which demands the VM to execute off the latter.

Prototype. Our prototype targets 32-bit Cortex M MCUs and BLE radios. Albeit COIN's programming model is independent of the underlying network technology, BLE is arguably a natural choice whenever integration with people's mobile devices is necessary and interactions may be triggered by proximity, as determined by radio connectivity [13]. We offer a primary example in Section IV. The prototype is mainly intended to provide a basis to assess the design rationale. It has a few limitations, which would require further implementation work and yet would *not* alter COIN's conceptual design.

PyMite does not offer per se resource arbitration, required to ensure that tasks by different parties safely share resources. This feature may be seen as desirable in any IoT VM. There exist literature on the subject [9] that can be applied to address this issue. Moreover, interactions across personal mobile devices and IoT ones are currently encoded by directly accessing the APIs of the BLE stack, that is, by reading and writing BLE *characteristics*. A dedicated APIs would be, however, needed on both sides to express such interactions at a higher-level of abstraction.

Modern networking stacks, such as BLE, are already equipped with built-in security features. As a result, the main security threat for COIN is likely going to be the authenticity of the Python bytecode. Techniques such as code signing [6] exist to address this issue, and are shown to be applicable to devices even more constrained than the ones we target. For example, the digital signatures employed in the Deluge protocol [6] incur very limited processing overhead. Porting these solutions to COIN should therefore be feasible with limited effort. Most importantly, code signing would be a one-time cost at the moment of deploying a task. The performance figures we discuss in Section V—obtained *after* a task is successfully deployed—would therefore retain their validity.

IV. USING COIN

We report on the use of COIN in the design and implementation of a pervasive game [12] whose logic spans smartphone and IoT devices in the environment.

The game. Only the bravest can become pirates! To prove their qualities, the aspiring pirates must travel to a mysterious island and overcome several challenges.

Players are divided into teams. The team who obtains the highest score becomes the pirate crew. To accumulate points, a team must collect items scattered across the island. Items are of different types: compasses, rare seeds, parrot eggs, bottles of rum, and sabers. The value of the first three kinds of items is 10 points. The latter two give 200 points, and can be obtained by paying a merchant with some of the collected items.

A team may decide to transform rare seeds and parrot eggs into plants and parrots, respectively. By doing so, the transformed item yields a score 10 times higher of the original one. However, eggs and seeds grow only if they live at the right temperature for a sufficient time. To this end, teams must deposit the collected eggs or seeds in suitable places, taking care not to be seen by opponents, who could kidnap the items after their transformation.

From virtual to physical. COIN allows us to develop the game in a way that no other existing platform would enable. Real-time user interactions happen through the touch interface of a standard smartphone. We deploy COIN on ST Nucleo-F091RC prototyping boards equipped with a BlueNRG BLE radio and an ST X-Nucleo-IKS01A1 sensor shield, as shown in Figure 3, and install them in our department building.

We map virtual items in the game to dedicated data structures dynamically stored through COIN aboard the Nucleo boards. As a result, accessibility of items corresponds to

Fig. 3. User interface on a person's smartphone and environment-immersed sensor device in our pervasive game.

```
1 from coin import *
2
3 # ... initialization ...
4 startTask()
5
6 #get temperature value
7 temp = getIntInput(0)
8
9 # ...seed germination ...
10 if (count == TIME_TO_GERMINATE):
11
12  # process and output germination level
13  output(germination)
```

Fig. 4. Task code to simulate seed germination over time and depending on physical temperature.

proximity to the Nucleo board storing the corresponding data structure—as dictated by BLE communication range—and mobility on the island maps to physical mobility in our building. Implementing the collection or release of items in the game is thus as simple as reading or writing from/to COIN's data store from a player's smartphone. With this design, straightforwardly enabled by COIN, virtual and physical dimensions spontaneously blend together.

However, there is more that COIN enables. Temperature conditions that determine how eggs and seeds grow are now simple to link to temperature in the physical environment. A player's smartphone can dynamically deploy, together with the item itself, a simple COIN task that periodically probes the temperature sensor on the Nucleo board and accordingly modifies the values of the data structures representing eggs and seeds. This may happen *independent* of the connection to a player's smartphone, giving players the illusion that the game unfolds across the virtual and the physical world. Figure 4 shows an excerpt of the task implementation that simulates the germination of seeds, which becomes as small as 320 Bytes at the time of deploying the task from the smartphone.

Finally, unlike existing pervasive games [12], COIN naturally allows other apps to re-use the deployed sensing infrastructure. Say, for example, a new app is developed to control air conditioning in our offices based on temperature and individual preferences. A user's smartphone may deploy a new COIN task that computes short- and long-term trends of relevant quantities, useful as inputs for implementing the feedback loop. Provided the sampling periods are compatible, the new task may just re-use part of the sensed data that the game already requires.

V. PERFORMANCE

Our prototype requires about 154 KB of program memory and about 10 KB of data memory. About 90% of this is due to PyMite and BLE drivers. However, PyMite is only meant to provide a working Python interpreter and its memory demands could be significantly reduced by tailoring it to COIN. Even at prototype stage, COIN fits most existing 32-bit embedded platforms. For example, the Cortex M0 core often used in SoC designs with a BLE radio provides 256 KB (16 KB) of program (data) memory.

Replacing wireless transmissions with device-local processing typically improves energy consumption. With energy efficient protocols such as BLE, 32-bit MCUs, and the overhead of code interpretation, such a claim needs to be newly demonstrated. Therefore, we measure COIN's energy performance in a set of representative applications against a traditional "sense-and-send" design implemented in C to validate our design choices. Similarly, we compare the execution times of COIN against functionally-equivalent implementations in C. In both cases, the C implementations are deployed as an *immutable* binary on the target platform.

A. Benchmarks and Metrics

We consider three applications based on the scenarios and requirements previously discussed. Each application corresponds to a COIN task.

We first consider Run Length Encoding (RLE) compression. RLE is often advocated for applications where sensors report stable values. Next, we consider an activity-detection algorithm to distinguish between *standing* or *walking* activities. The activity detection (AD) occurs on 5 Hz accelerometer data by computing average and standard deviation of the signal amplitude. Finally, we consider an algorithm to extract ECG information [2]. The signal is passed through multiple tap filters and then compared against a threshold to detect peaks indicating physiological issues.

We consider four prototyping platforms: a Freescale FRDM-KL46Z, a Nordic nRF51-DK, a NXP LPCXpresso1549, and a NXP LPCXpresso4337. These offer the full range of Cortex M MCUs, as well as varying amounts of program (256 KB to 1024 KB) and data (16 KB to 136 KB) memory. We attach Bluetooth extension boards where necessary. In the absence of an earth-rate sensor, we use the accelerometer; this does not impact the execution of the ECG algorithm. We disable all unnecessary peripherals.

We measure *energy consumption* and *execution times* through a Tektronix 1072B oscilloscope. The values we present are averages over at least five repetitions. The standard deviation across different runs, not shown in the charts, is always within 5% of the average. Detailed information on the experimental setup are found in the companion report [11].

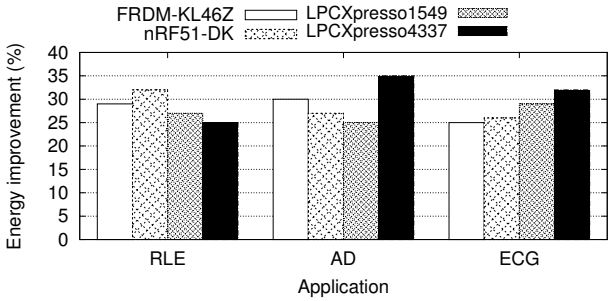

Fig. 5. Energy performance of COIN compared to a sense-and-send design.

TABLE I
EXECUTION TIMES OF TASKS WITH PCXPRESSO4337.

Task	Plain C [us]	COIN [ms]	Ratio
RLE	108,37	8,63	79,93
AD-local	163,97	4,98	30,55
AD-transmission	68,31	5,22	76,76
ECG-local	127,82	3,78	29,76
ECG-transmission	89,21	4,55	51,12

B. Results

Energy consumption. We feed data to RLE so to achieve a 50% compression ratio, in fact pessimistic for RLE compression of sensor data [15]; AD reports data to a smartphone every 30 sec, whereas ECG samples the sensors at 30 Hz. Results by varying these parameters are, nonetheless, available [11].

Figure 5 reports the results. The trade-off between saving transmissions by deploying COIN tasks and the additional MCU overhead due to code interpretation is in favor of COIN. In our experiments, the improvement in energy consumption is at least 25%, with a best case of 35%. This is despite the efficient energy performance of BLE radios.

The LPCXpresso4337 board shows the best performance in Figure 5 when running the AD task. The FPU of the Cortex M4 core speeds up the execution of the floating point operations in AD. In contrast, the Cortex M0+ core on the nRF51-DK provides the best performance with sequential byte-level operations, as in RLE.

Note that the energy cost for deploying the task is a *one-time* cost. The AD task, for example, requires about 50 packets. These may be transmitted using BLE's streaming mode, which reduces efforts for packet trains. Thus, the energy overhead quickly amortizes as a task continues to run.

Execution times. Larger execution times represent the cost for increased flexibility and better energy efficiency in COIN. For the AD and ECG tasks, we separate the case of regular local processing at every iteration from the case of data transmission that requires extra computations to prepare data for transmission.

Table I shows the results for PCXpresso4337 board. The ratios are similar for the other platforms. The values are still in the same order of magnitude of packet transmissions, and should not be detrimental to the app responsiveness, including user interactions. The slowdown is vastly dominated by code interpretation, and yet the values in Table I are in line with existing literature [5], [3]. Note that PyMite is not expressly designed for the platforms we target, neither we explicitly optimize it besides the adaptations in the previous section.

As each task takes longer to run, the slowdown may impact the overall schedulability, limiting the number of tasks concurrently executing. However, we can run up to six instances of the AD task on a Cortex M0 MCU, for example, before scheduling becomes unfeasible.

VI. CONCLUSION

We presented COIN, a software architecture that provides the glue necessary to create an open vendor-independent programming substrate of IoT devices accessible from people's mobile devices. COIN offers a high-level programming model based on a lightweight notion of task as a relocatable unit of mobile app logic. Its run-time support takes care of the tasks' dynamic deployment, real-time scheduling, and cross-task coordination. We demonstrated how COIN overcomes the limitations of traditional designs; for example, by enabling a degree of flexibility in the design of immersive computing applications that no other existing platform may similarly provide. We also showed that the device-local processing COIN enables can improve a device's energy consumption up to a 35% factor in our tests, at the expense of larger execution times due to code interpretation.

Acknowledgments. The authors would like to thank Riccardo Paccagnella for the implementation work on the pervasive game described in Section IV, and Naveed Bhatti for supporting the authors during the experimental evaluation.

REFERENCES

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems*. PhD thesis, MIT Artificial Intelligence Laboratory, 1985.
[2] D. Albu, J. Lukkien, and R. Verhoeven. On-node processing of ECG signals. In *IEEE CCNC*, Jan 2010.
[3] F. Aslam et al. Optimized Java binary and virtual machine for tiny motes. In *IEEE DCOSS*, 2010.
[4] M. Botts and A. Robin. OpenGIS sensor model language (SensorML). *OpenGIS Implementation Specification OGC*, 7, 2007.
[5] N. Brouwers et al. Darjeeling, a feature-rich VM for the resource poor. In *ACM SENSYS*, 2009.
[6] P. K. Dutta, J. W. Hui, D. C. Chu, and D. E. Culler. Securing the deluge network programming system. In *IEEE/ACM IPSN*, 2006.
[7] S. Hoppough. Shelf life. Forbes Magazine, 2006.
[8] W. Horré, S. Michiels, W. Joosen, and P. Verbaeten. DAVIM: Adaptable middleware for sensor networks. *IEEE Distributed Systems Online*, 9(1), 2008.
[9] A. Lachenmann et al. Meeting lifetime goals with energy levels. In *ACM SENSYS*, 2007.
[10] P. Levis and D. Culler. MatÉ: A tiny virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(5), 2002.
[11] M. L. Stefanizzi, L. Mottola, L. Mainetti, L. Patrono. Coin: Opening the internet of things to people's mobile devices. Technical report 2016.17, Politecnico di Milano, http://goo.gl/oMHRRT.
[12] C. Magerkurth, A. D. Cheok, R. L. Mandryk, and T. Nilsen. Pervasive games: Bringing computer entertainment back to the real world. *Comput. Entertain.*, 3(3), 2005.
[13] L. Mottola et al. Enabling scope-based interactions in sensor network macroprogramming. In *IEEE MASS*, 2007.
[14] D. Simon et al. Java™on the bare metal of wireless sensor devices: The Squawk Java virtual machine. In *ACM VEE*, 2006.
[15] N. Tsiftes et al. Efficient sensor network reprogramming through compression of executable modules. In *IEEE SECON*, 2008.