Enabling Ultra-Low Power Operation in High-End Wireless Sensor Networks Nodes

Carlo Brandolese, William Fornaciari, Luigi Rucco, Federico Terraneo
Politecnico di Milano - P.zza Leonardo da Vinci, 32 - 20133 - Milano (Italy)
{brandole,fornacia,rucco,terraneo}@elet.polimi.it

ABSTRACT

This paper presents a prototype hardware/software architecture for minimizing energy consumption on high-end microcontrollers, while simplifying the development of applications providing a general-purpose-like programming environment. The key features enabling this twofold goal are operating system support to processes, optimized sensing and hibernation of the system state. To balance performance offered by high-end microcontrollers with the need for ultra-low power operation—especially required by WSNs—it is essential to minimize the duty cycle by keeping the microcontroller in a low-power state as long as possible, without affecting the performance required by the application. To cope with the increasing leakage current observed in today's microcontrollers a software-based hibernation mechanism has been implemented to transparently save the memory of processes in a non-volatile memory, allowing to completely switch off the microcontroller. To further avoid costly process loading overheads during the sleeping periods, we devised a smart sensing mechanism capable of gathering data from peripherals without restoring the state of processes. Preliminary results show that the lifetime of the proposed node architecture is of the same order of magnitude of that of ultra low-power nodes and significantly better than that of high-end ones.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management; C.2.4 [**Computer-Communications Networks**]: Distributed Systems—*Network operating systems*; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms

Design, Performance

Keywords

Power Optimization, Wireless Sensor Networks

CODES+ISSS'12, October 7-12, 2012, Tampere, Finland.

1. INTRODUCTION

The goal of this work is to bring some of the features of general-purpose systems, such as multitasking, memory protection, dynamic loading, standard programming languages and libraries, into wireless sensor networks. Such features are, in fact, expected to significantly increase the capabilities of wireless sensor networks, allowing for more complex tasks on-node, reducing radio transmission, running multiple isolated tasks on the same node and simplifying code development by leveraging on standard libraries and APIs. Moreover, this approach enables code mobility, useful to dynamically reconfigure nodes both to add applications while the network is operating and to distribute the power consumption over nodes [1]. From this point of view, ultra-low power microcontrollers used for WSN nodes are seen as a stopgap to their evolution. This stimulated experimenting the use of high-end microcontrollers while at the same time optimizing their duty cycle to minimize the power consumption. The paper is organized as follows. Section 2 provides an overview of wireless sensor nodes and operating systems, in the light of which the positioning of the proposed approach is motivated in Section 3; Section 4 presents the hardware and software architectures that have been developed; Section 5 describes the energy model adopted to compare against existing nodes; Section 6 presents and discusses the quantitative results obtained. Finally, Section 7 summarizes the achievements of the present work.

2. RELATED WORK

Many sensor nodes are currently available off-the-shelf of two main classes: small, cheap and ultra-low power sensor nodes, on one hand, and costly, powerful high-end nodes, on the other. Within each family, the general structure of the hardware, as well as the average power consumption profile, do not vary significantly. Moreover, depending on the node characteristics, different operating systems are available: from small OS suitable for ultra-low power motes, to full-fledged embedded versions of general purpose operating systems like Linux. Each presents some advantages and disadvantages, depending on the goal being pursued: ultra-low power performances rather then flexibility and easiness of programming and development.

2.1 WSNs nodes

One of the best known nodes of the ultra-low power family is the Berkeley TelosB [2]. This node is provided with an MSP430 microcontroller, the internal memory of which is composed of 10KB of RAM and 48KB of Flash. Older

nodes, also well known, belongs to the MICA[3] family, in particular Mica, Mica2Dot and Mica2: all of them integrate an ATmega128 microcontroller, with 4KB of internal RAM and 128KB of Flash. Power consumption figures for these nodes are reported in Table 3, Section 6. Other nodes with similar characteristics are Firefly [4], with camera-enabled features (so more power consuming), WiSMote, devised by LCIS (Grenoble), and others.

Slightly more powerful architectures are the BTnode, developed at ETH Zurich, which does integrate larger memories (64KB of RAM, 128KB of Flash, 4KB of EEPROM and 180KB of external SRAM), but consuming 100–200mW in full active state. Another node in this class is Egs [5], based on a Cortex-M3 Atmel SAM3U2C μC, whose maximum operating frequency scales up to 96MHz and whose internal memory is composed by 36KB of RAM and 128KB of Flash, plus an external 2Gb NAND Flash. The microcontroller consumes 48mA in active state, while in wait mode (a sort of deep sleep) its consumption decreases to 15μA.

Among the high-end and powerful architectures, two of the most widespread platforms are the IMote nodes by Intel and the SunSpot, by Oracle Sun. The IMote2.0 [6],integrates a PXA271 XScale processor, whose operating frequency can range from 13MHz to 416MHz and is equipped with 32MB of SDRAM and 32MB of Flash, with additional 256KB of SRAM. The current absorption in active state and radio off are considerably high, and are reported as well in Table 3, Section 6. Previous versions of this node, i.e. IMote and IMote1.0, are both based on ARM architectures, with 64KB of RAM and 512KB of Flash. SunSpot[7] is another example of powerful and energy-consuming device: these nodes use an ARM920T core operated at up to 180MHz, and are provided with 512KB of RAM and 4MB of Flash memory. The current consumption reaches 98mA with processor in active state and radio on and 80mA with radio off. In sleep mode, the current consumption is 24mA, while in deep-sleep it decreases to 33μA.

2.2 WSNs Operating Systems

Because of the different amount of memory and computing power available in the two classes of nodes, also the operating systems and the services offered at system level considerably vary. Many operating systems have been proposed for low-end nodes, all of them having in common the need for small footprint (10KB or less) and low processing overheads. These operating systems can be divided in three main categories with respect to their architecture [8]: *monolithic* (TinyOS, MagnetOS, etc.), *modular* (Contiki, Mantis OS, SOS, Bertha, CORMOS, Kos, etc.) and *virtual machines* (Maté, ContikiVM, MagnetOS JVM, VM*, SwissQM, etc.). Some of the above-cited systems fall into the paradigm of pure event-driven (e.g. TinyOS and SOS), other into pure thread-based (e.g. MantisOS) and others are hybrid and incorporate both the aspects (e.g. Contiki and kOS). Besides this very fundamental distinction, the various operating systems differs for the way they support scheduling, memory management, dynamic reprogramming, power management and network protocol stack(s). Due to the resource-constrained hardware, hardly any small operating system of this family really provide complete functional coverage for a general-purpose like programming to the end-user, such as standard libraries, standard system call, standard executable format and reprogramming mechanisms, standard programming languages and so on. On this last point, it is worth noting that many operating systems have been developed in lightweight variants of C language, e.g. TinyOS in NesC or LiteOS in LiteC++. Other, like Contiki and Mantis, have been written in C, but the programming style needs to be adapted to specific features of the systems Some virtual machines try to emulate a subset of the Java instruction set, e.g. SwissQM[9].

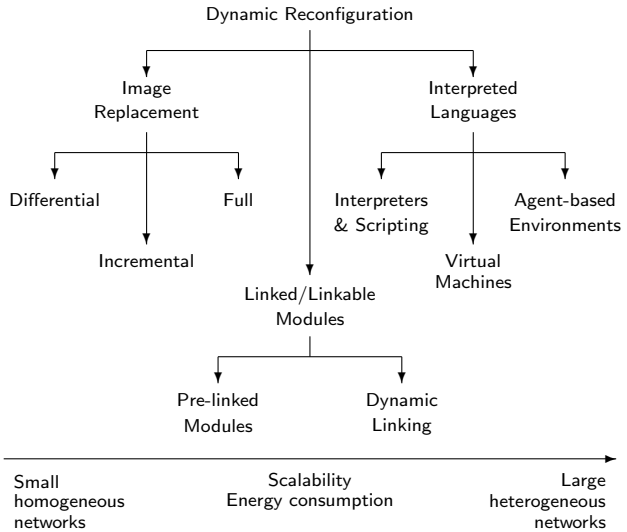

Figure 1: Reconfiguration techniques overview

The reprogramming mechanism is one of the features with greater impact on the user-level development process. Virtual machines well fit with the need for dynamic reconfiguration of the networks, but usually require very high run-time costs. Similar advantages and problems arise in other interpreted environments, like run-time script interpreters (e.g. SensorWare [10], COMIS [11], SpatialProgramming [12] or DeepPython [13]) or distributed agent-based approaches (e.g. Agilla [14]). Besides raw image replacement techniques (native in TinyOS, MantisOS, NutOS and others), some distributed mechanisms have also been devised to make network-wide reconfiguration more efficient. The most notable examples are XNP [15], MOAP [16] and Deluge[17, 18]. In addition to theses monolithic approaches, some *binary differential* solutions have been proposed (Koshy and Pandey [19], Reijers & Langendoen [20], Jeong & Culler [21]), which are very energy efficient, but not scalable.

Many operating systems support pre-linked modules, for example Contiki, SOS, Bertha, and other like Impala for Zebranet [22] or ScatterWeb [23]. Lightweight variants of dynamic linkers have also been proposed in FlexCup [24] and Contiki [25]. A synopsis of the reprogramming approaches is presented in Figure 1.

Power management service are offered by various small WSN operating systems like TinyOS [26] and SenOS [27], while dedicated APIs are provided in NanoRK and PEEROS.

In small WSN operating systems, there is no isolation among running programs because of the lack of real processes, mainly due to the absence of a MPU or a MMU on the microcontroller. Very few proposals have been presented for managing a sort of virtual memory in MMU-less platforms at software-level. SNACK-pop [28] implements a

form of compiler-assisted paging. The t-kernel [29] operating system provides MMU virtual memory management and memory protection for `text` and `data` segments. A proposal for paging only the `data` segment, by enforcing virtual memory through application-level library and VM-aware assembler can be found in [30], while software-level memory expansion techniques can be found in MEMMU [31] and Enix [32]. To the best of our knowledge, none except isolated cases of commercial operating systems for embedded devices, like Micrium, have modules to support processes, while standard C libraries are available, often in a reduced version, in very few operating systems such as Enix. New proposals like MansOS [33] claim to offer some principles of Unix operating systems like sockets, threads and device managers, with plain C programming.

More powerful nodes, like Imote2, can run embedded Linux releases like μCLinux, while the SunSpot mounts the Squawk Virtual Machine [34], a full-fledged micro-edition Java virtual machine. Clearly, such a virtual machine has very high energy costs, reducing the estimated lifetime of a node in active state to 23 hours only [7]. These systems, designed for high-end and energy-consuming platforms, have a footprint in the order of megabytes, so they are unportable to the small *ultra-low* power nodes. On the contrary, however, they provide to the end-user a development environment similar to that of general-purpose platforms, at the expense of a lifetime significantly shorter.

Some other operating systems and middleware layers for WSNs have been proposed, supporting real-time applications and other special use cases, but they fall beyond the objectives of the present work.

3. PROPOSED APPROACH

As seen, a myriad of operating systems have been proposed for small nodes, each with particular features which aim at providing to the developer services to cope with the high complexity of a WSN. None of them, however, seems to reach a reasonable level of completeness and generality to avoid a programmer to be acquainted with a lot of specific issues of the OS and of the hardware underneath. High-end nodes and operating systems, on the contrary, succeed in the purpose of furnishing to the end-user ease of programming and general-purpose like environments, but show unacceptable power consumption and thus do not fit for application which require long lifetime and cheap hardware devices. The solution appears to place in the balance between the low-power features offered by small nodes and operating systems, with the abstraction level offered by high-end ones, but this clearly leads to a trade-off.

A possible solution to manage this trade-off is to take advantage from the newest and next-generation microcontrollers, which offer a good amount of memory and computing resources and have been designed to provide different operating states with different profiles of power consumption. Unfortunately, increasing leakage current due to the reduction to nanometer scale of newest high-end microcontrollers seems to be an harsh barrier to the development of hardware/software WSN architectures based on these devices. On the other hand, the multiple operating points offered by these microcontrollers constitutes a foremost opportunity to cope with the problem of leakage by optimizing the duty cycle, exploiting the possibility of switching from high-performance to low-power states and taking advantage

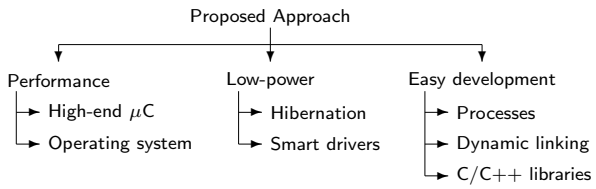

Figure 2: Goals of the proposed approach

from the high scalability in operating frequency. The lowest power consumption states, however, often require the internal memory to be switched off, maintaining in active state only a small portion of backup memory (usually less than 10KB). Thus the possibility of adopting this new class of microcontrollers, making the most of their duty cycle optimization, passes through the need of *hibernating* the system state on an external memory. High frequency of swapping operations does not fit well with the usage of Flash or EEPROM memories, which have a limited number of write cycles and need expensive wear-leveling at software-level. This limitation can be overcome thanks to emerging magneto-resistive, ferro-electric and phase-changing RAM, which have unlimited number of write cycles, retain their state even when switched-off and have comparable energy consumption figures. Optimization of swap-in/swap-out operations must then be performed by the operating systems to reduce hibernation overheads to an acceptable level.

In conclusion, a possible effective way for enforcing the adoption of these new microcontrollers and memory devices in WSNs seems to be a joint development of both the hardware and the operating systems architecture, eventually extending an existing small operating systems with a proper power-management layer, which should enforce, in particular, an efficient management of the system hibernation as well as a smart usage of peripherals drivers, to maximize energy saving during idle and purely sampling periods. Moreover, the superior resources made available by new generation high-end μC can lead to the implementation of operating system features typical of general purpose systems, with which many programmers are familiar, eventually promoting a wider adoption of WSNs outside highly-specialized fields. Among these general features, some of the most important to make the development phase easier are the availability of standard Libraries, support for full-fledged processes as well as Dynamic Linking and Reprogramming (Figure 2).

4. HW/SW ARCHITECTURE

This section describes the architecture of the PoliNode and Miosix operating system. First, the hardware architecture will be discussed, whose distinguishing feature is the presence of a high-end microcontroller and a non-volatile memory for performing process hibernation. Then, the software architecture is discussed, detailing the implementation of processes, hibernation and smart drivers. It should be noted that while we focused our choices on a specific hardware implementation and operating system, the concept of combining processes, hibernation and smart drivers constitute a general methodology useful for a wider range of energy critical and reconfigurable systems. Such concepts, not only can be implemented also on different microcontrollers and

Transceiver

Operating state	Current drawn
Deep sleep	900 nA
Receive	13.1 mA
Transmit (0dBm)	11.3 mA

Microcontroller

Operating state	Current drawn
Running at 60MHz	12 mA
Sleeping at 60MHz	5 mA
Running at 16MHz	4 mA
Sleeping at 16MHz	2.1 mA
Sleep, unclocked	300 μA
Deep sleep, unpowered and unclocked	4.8 μA

Magnetoresistive RAM

Operating state	Current drawn
Quiescent	7 μA
Reading at 15MHz	3.75 mA
Writing at 15MHz	13.4 mA

Table 1: Power consumption summary

with different operating systems, but can also be proficiently used outside of the area of wireless sensor networks.

4.1 Hardware architecture

An high-level sketch of the PoliNode architecture is shown in Figure 3. The sensor node is divided into five functional units, each serving a well defined purpose.

The *Communication unit* integrates a power optimized wireless transceiver operating at 2.4GHz—a frequency commonly used by many WSN nodes, including the TelosB and the IMote— and a PCB antenna, which, again, is a common choice to minimize production costs. A summary of the transceiver power consumption figures is reported in top part of Table 1.

The *Processing Unit* is composed of an ARM Cortex-M3 based microcontroller providing hardware memory protection and integrating 128KB of RAM and 1MB of Flash memory. The core can operate at up to 120MHz, but in the current implementation it is used at 60MHz. The core also features a low-power operating mode in which it can be clocked by an internal 16MHz oscillator. This shortens the startup time from the Deep Sleep mode, since it is not necessary to wait for the external oscillator and internal PLL to settle. Concerning sleep modes, it is possible to have the whole processor powered but unclocked (except for the RTC), guaranteeing RAM data retention, or shut down the internal voltage regulator, losing the content of the internal RAM, except for a small 4KB portion and maintaining the RTC active. As the power figures in the middle part of Table 1 show, the power consumptions is significantly different in these two modes. This microcontroller has been selected to provide enough computational power and memory to both be able to perform complex on-node tasks, accommodate applications making use of C/C++ standard libraries, as well as running multiple processes concurrently.

A key component of the node is the magneto-resistive RAM (MRAM) used to save the data image of the running processes while the node is in deep sleep mode. The use of an external non-volatile memory coupled with hibernation support allows to write applications that are programmed as if they were always resident in memory, relying on hibernation activated by the operating system during deep sleep

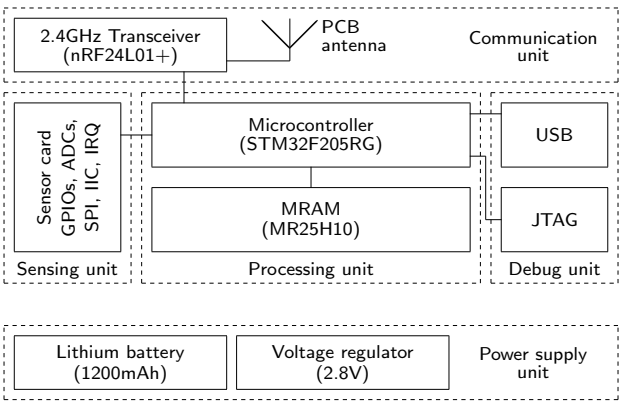

Figure 3: Architecture of the PoliNode

periods. The lack of such a feature would have prevented the microcontroller from entering its lowest power state of operation, and is therefore a key component of the node. A 1 Mbit MRAM memory has been selected instead of a Flash memory both due to its lower power consumption and for having a virtually unlimited number of write cycles. The MRAM is connected to the microcontroller through an SPI bus operating at 15MHz. A DMA channel of the microcontroller is used to ensure that the required speed is actually achieved when copying the data image of processes. A summary of the power consumption information for the MRAM is reported in the bottom part of Table 1. Note that when the MRAM is not used, i.e. during deep sleep periods, the microcontroller can switch it off, bringing its power consumption from 7μA of the quiescent state to zero.

The *Sensors card* collects specific sensing devices and is strongly application dependent. Therefore the node, instead of integrating specific sensors, provides a connector for a custom sensor daughter-board. The connector provides a number of analog channels connected to the ADC, SPI and I^2C interfaces for digital sensors, as well as some GPIOs. One of the GPIOs is configured to generate an interrupt, used to wake the processor from deep sleep when specific sensors generate an event. Furthermore, additional circuitry allows switching sensors on and off by means of a subset of the GPIOs available in the daughter-board connector. Efficient policies can be designed both at application and operating system level to exploit this feature to further reduce the overall power consumption.

The *Power Supply Unit* hosts a 1200mAh lithium battery and integrates a 2.8V voltage regulator ensuring that the voltage used to power the processor, the transceiver, the sensors and the MRAM remains constant throughout the battery's lifetime. Finally, a *Debug Unit* provides an USB connector to load the operating system to the board, and a JTAG header is for in-circuit debugging during the development phase. It must be noted that once the sensor is operating, new applications can be downloaded over-the-air, as explained in more detail when discussing processes.

4.2 Software architecture

The software architecture adopted for the node is based on the Miosix operating system (previously developed by one of the authors). A simplified view of its structure is shown in Figure 4. It is composed of a board support package that collects low-level, architecture dependent code, a kernel

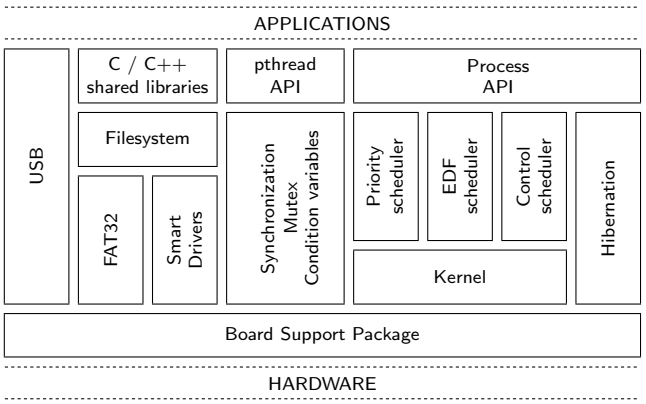

Figure 4: Architecture of Miosix

providing scheduling and synchronization primitives, as well as an support for the C/C++ standard libraries. Multithreading is also available through the standard POSIX API. Extensions to this base system are described in the following.

4.2.1 Processes

Processes are a key feature of the proposed architecture, and their introduction allows the implementation of *run-time application loading* onto the sensor nodes without affecting the operation of other running processes. In addition, processes—differently from threads—provide *memory isolation* and simplify the implementation of the hibernation functionality. A process is an independent instance of a program running on a node, with its own *data image* composed of a stack, optionally a heap, and some global variables. Multiple processes can run in the system, and preemptive multitasking is used to schedule them.

Program loading starts by allocating a suitably sized RAM area to store the process data image. Then, global variables are initialized and—if needed—relocated, since no MMU is available and addresses of data are not known until run-time. Relocation of program code, on the other hand, is not necessary, as the microcontroller and toolchain allow for efficient Position-Independent Code (PIC). The resulting process image is therefore composed of two non-contiguous memory areas, one in Flash memory, containing the program code, and one in RAM, hosting process data. This arrangement simplifies memory isolation, obtained by means of the Memory Protection Unit (MPU) configured to set permissions for read, write and execute operations on selected memory regions, according to the process memory layout. The operating system, at every context switch, configures two areas of the MPU with the actual memory positions of the program code and the data of the process that is about to execute. Any attempt to access locations outside these memory areas or to violate access permissions results in an hardware interrupt that the operating system uses to terminate the process, preventing run-time errors in one program from affecting other software running on the same node.

Since processes cannot access other memory areas, including the operating system code, a general-purpose *system calls* mechanism has also been implemented.

It is worth comparing the presented approach with similar proposals used in thread-based operating system supporting dynamic linking. As a reference we consider Contiki

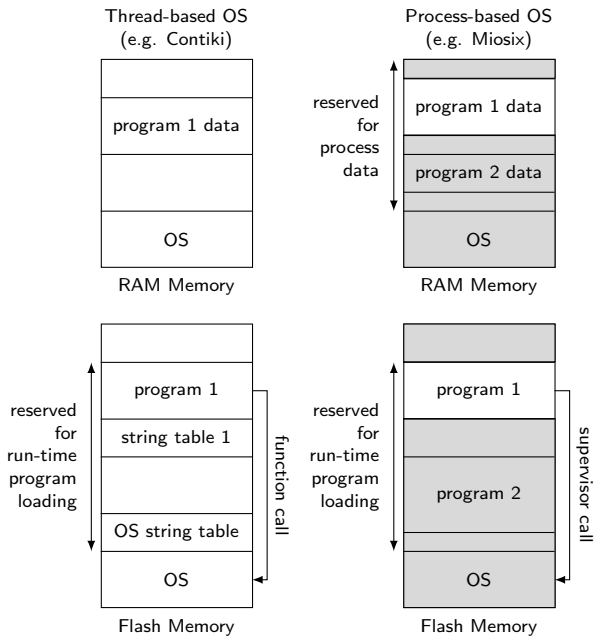

Figure 5: Memory layouts and system calls

[35]. Figure 5 shows the memory layout and system call approaches in the two cases. Miosix provides memory protection (grayed-out areas) and system calls, while Contiki, lacking any form of protection, can use regular function calls.

Both these approaches allow to execute programs loaded at run-time, thus enabling an efficient reprogramming mechanism for remote nodes. However, while in Contiki and other similar embedded operating systems there is no clear separation between system and user addressing space, a process-based execution model enforces memory isolation, thus providing a secure environment for testing applications and preventing malicious software from untrusted sources to affect the functionality of the system.

4.2.2 Program file format

The program format chosen for dynamic loading is the standard ELF format. As noted in previous works [35], the ELF format has a non negligible size overhead (meta-data), resulting in an increase of the energy required to transfer a program over the network. The cited authors reduced the ELF size by replacing 32bit addresses in the meta-data with 16bit ones, but this approach is only suitable for 16bit microcontrollers. While the meta-data found in *sections* is vital for compilers, loaders do not need it, and the ELF specification do not mandate their presence in executable files. We have thus completely removed section information. In addition, the *symbol table* and *string table* have been removed as well, since calls into the operating system are not made through function calls—which would require relocation—but via system calls. The resulting ELF structure is still compliant to the standard, but only consists of the minimal information required by the loader.

4.2.3 Hibernation

As already discussed, the lowest power state of high-end microcontrollers, differently from ultra-low power ones, can

be achieved by powering off the processor core, therefore losing the content of the internal RAM memory. This would not allow saving the data of running processes, therefore hibernation was implemented. When all processes are either waiting for data from the smart drivers or sleeping, and the closest wakeup time is greater than a certain threshold, the operating system performs an hibernation operation. This consists in: i) saving the RAM data image of each running process in the external non-volatile memory, ii) configuring the RTC to wakeup the processor at the required time, and, iii) entering deep sleep mode. On the other hand, the operating system data image is not saved, and a new boot is performed when the RTC interrupt occurs. This is made possible by the very short boot time of Miosix (144μs) combined with a minimal set of information (mainly the process descriptors) stored in the microcontroller's backup RAM. After booting, thus, the operating system restores the state of processes from the backup RAM, swaps-in their data image from the external memory and restarts them.

A number of optimizations can be adopted to minimize the overhead of swap operations. A first improvement consists in limiting the data to be swapped to the portion of the RAM image actually used (the operating system knows about the actual heap and stack usage), while an even more aggressive approach would postpone swapping-in processes with longer sleep times to the moment they are strictly required to be executed.

4.2.4 Smart drivers

While the aforemetioned optimizations reduce the overhead of swapping processes in and out, the need for an application to perform periodic measurements results in a particularly inefficient situation where a process wakes up only to perform a very short operation, such as doing a single measurement from a sensor for later processing. In such cases the time and energy to swap-in and swap-out the process data image is much higher than that strictly required to perform actual work.

To solve this issue, *Smart Drivers* have been conceived and introduced. A smart driver is an enhanced peripheral driver within the operating system, that offloads the burden of periodically performing measurements from the applications. Accessing a smart driver from a process, in fact, causes the process itself to be blocked and to be awakened only after a predefined *measurement batch* is completed, i.e. when a queue allocated for storing samples is full. Individual measurements are then performed by the kernel, without the need of swapping-in and awakening any process. For example, the smart driver for performing ADC sampling has the following interface:

```
int adc_sample( int channel, int *queue, int size,
                int period, int min, int max );
```

where the function arguments have the following meaning: `channel` specifies which ADC channel to sample, `queue` points to a queue where samples are stored, `size` is the queue size, and `period` is the time between two samples. Finally, `min` and `max` are parameters used to wakeup the process earlier, should a sample fall out of the specified range. This allows awakening the process as soon as an anomaly is detected.

The operation of the smart driver depends on the type of measurement requested, which can either be periodic or event-driven. In case of periodic sensing, the smart driver works by configuring the internal RTC to wakeup the processor at regular intervals to perform measurements, while in case of event-driven operation sensors themselves are responsible to wake the processor up by means of an interrupt. In both cases, the processor and the internal RAM are powered-off in the waiting periods, while only the internal 4KB backup RAM is active and retains the queues for the smart drivers. In such a way the overall power consumption profile in the sleeping period is lowered to a level typical of ultra-low power systems.

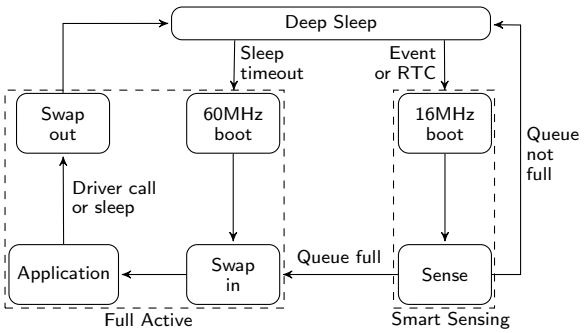

Figure 6: Operating model of the system

When an RTC or an event occurs, the processor is awakened and clocked by internal low-power 16MHz oscillator. At this frequency, the operating system boots in 540μs[1], then the actual *sensing* phase begins. In this phase the sensor data is retrieved and the result pushed in the queue associated to the process requiring the measurement. If the queue is not full the processor goes back to deep sleep, otherwise it immediately loads the calling process performing a swap-in operation and enters full active state. After an entire measure batch is completed, the queues in the backup RAM are elaborated by the processes, minimizing the full active duty cycle and reducing the overhead of frequent processing and related swapping operations. Figure 6 shows the different phases of this mechanism.

4.3 Operating model

The overall mode of operation of the proposed solution is summarized in Figure 6, where are three main states are highlighted, *Full Active*, *Deep Sleep* and *Smart Sensing*, each characterized by a specific power consumption profile (see Section 5 for details). The transitions among the various states follow the logic described in previous sections.

5. ENERGY MODEL

In this section we derive a first energy and timing characterization of the different operating states with the goal of building a simplified overall non-functional model of the entire system. Such a model will be used in the next section as a basis to obtain the presented results. Timing figures used in the models are derived from direct measurement on the hardware using an oscilloscope. Energy consumption data, on the other hand, are all average values derived from the data-sheets. The reason is that the current hardware

[1]The sum of the time required by the microcontroller voltage regulator startup and OS boot poses a lower bound of approximately 1ms on the node response latency, limiting its adoption for tight real-time applications.

implementation of the node does not provide probes and dedicated circuitry for current sensing.

5.1 Full Active mode

As discussed, the Full Active mode is always preceded by a startup phase, which in turns is composed of three steps. A pictorial view of these phases is provided in Figure 7, where the contributions of the microcontroller, the external magneto-resistive non-volatile RAM and the radio transceiver are shown separately.

As far as the microcontroller is concerned, the first step requires waiting the internal voltage regulator to stabilize. At this point the microcontroller is clocked at 16MHz and waits for the internal PLL to lock at a frequency of 60MHz, which is the frequency we have chosen for normal operation. Once the PLL is stable, the operating system starts booting, which requires 144μs. The system is then ready to start the execution of one or more processes. This requires (apart from the very first time) swapping-in the process data section from the external memory into RAM before starting the execution of the processes. The number of processes n_p that need to be loaded is, in general, a time varying value as different processes may have different sleep and sensing cycles, and those still sleeping or waiting on smart drivers do not have to be loaded. The time needed to load a process is bound by the size of its data image (8KB in the current implementation) and the data rate of the SPI bus (15MHz, in our setting) connecting the microcontroller with the MRAM. Once the process image has been loaded, the operating system schedules processes and remains in this state for a time τ_{app} that, depending on the specific application, is left as a parameter of the models. The energy consumption for this phase is expressed as average power and is 34nJ/μs.

When the application terminates its time slot, the updated data images in RAM are swapped-out. Reading and writing magneto-resistive memories requires the same time, thus the duration of this phase is exactly the same as the swap-in. It must be noted, though, that if one or more processes are not scheduled during the application time period, its data will have undergone no changes and thus needs not to be swapped out. Summarizing these contributions leads to the equations (units are μs and μJ):

$$\tau_{\mu C,fa} = 944 + 8740 \cdot n_p + \tau_{app} \quad (1)$$

$$E_{\mu C,fa} = 15 + 188 \cdot n_p + 0.034 \cdot \tau_{app} \quad (2)$$

In addition to the microcontroller, the Full Active phase requires access to the MRAM, with an additional energy consumption contribution, shown in the middle diagram of Figure 7. The specific components used in the current implementation requires a power-on phase during 400μs but having a negligible energy contribution of 100nJ. Reading and writing require the same time but consume different energies. The swap-in phase absorbs 46μJ, while the swap-out phase consumes 164μJ. While the application is running, the memory is kept in its quiescent state, with a power consumption of 20pJ/μs. Compared to energies involved in the other phases this last contribution can be neglected. The resulting models are the following:

$$\tau_{mram,fa} = 400 + 8740 \cdot n_p + \tau_{app} \quad (3)$$

$$E_{mram,fa} = 210 \cdot n_p \quad (4)$$

Finally, while in Full Active mode, the radio transceiver is powered on to perform two main operations: synchronization and data exchange. The synchronization phase requires receiving a specific packet at predefined time intervals. To compensate for drifts of the interval duration two slack periods, one before and one after the actual receive phase, are needed and the interval duration itself is adjusted at runtime using an integral regulator. As far as the data exchange phase is concerned, we suppose that its duration is 25% of τ_{app}, avalue compatible with a wide spectrum of ambient intelligence applications. Th resulting models are:

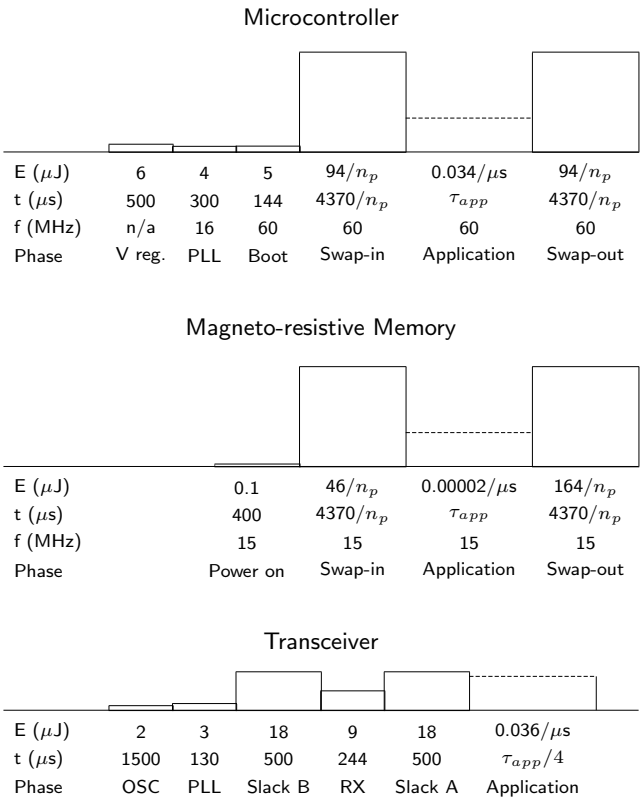

Figure 7: Full Active mode characterization

$$\tau_{tsc,fa} = 2870 + \tau_{app}/4 \quad (5)$$

$$E_{tsc,fa} = 50 + 0.036 \cdot \tau_{app}/4 = 50 + 0.009 \cdot \tau_{app} \quad (6)$$

The complete Full Active mode model is thus:

$$\tau_{fa} = 944 + 8740 \cdot n_p + \tau_{app} \quad (7)$$

$$E_{fa} = 65 + 398 \cdot n_p + 0.043 \cdot \tau_{app} \quad (8)$$

where $\tau_{fa} = \max(\tau_{\mu C,fa}, \tau_{mram,fa})$, while the total energy is simply the sum of the three contributions.

5.2 Smart Sensing mode

Smart Sensing is performed totally by the operating system and does not require executing the application code of the processes. Smart sensing begins with a startup phase similar to that of the Full Active mode, except that the operating frequency is set to 16MHz instead of 60MHz. This choice is mainly motivated by the fact that smart drivers only need to perform simple operations—basically register/memory transfers—and thus do not require the core to be running at an higher frequency. At this frequency, though the operating system boot takes 540μs and consumes 12μJ

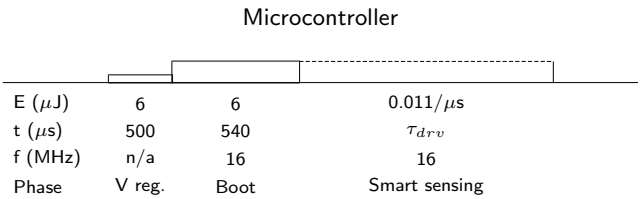

Figure 8: Smart Sensing mode characterization

of energy. After the operating system boot is complete, actual sensing operations are performed and require a time that, again, is application dependent and is thus left as a parameter in the model, namely τ_{drv}. The average power consumption of the microcontroller in this state is 11nJ/μs. The model is expressed by the following equations:

$$\tau_{ss} = 1040 + \tau_{drv} \quad (9)$$

$$E_{ss} = 12 + 0.011 \cdot \tau_{drv} \quad (10)$$

Sensing time and current consumption strongly vary depending on the type of sensor as well as on the specific device. Accounting also for this contribution, hardly fit with the need of comparing the proposed approach with previous architectures, since each of them integrates different sensors with different power consumption figures. For this reason we omitted sensors' contributions to E_{ss}, being possible at any time to account for it.

5.3 Deep Sleep mode

In Deep Sleep mode, the node is almost completely powered off. In particular, the MRAM is totally switched off, while the microcontroller is forced into its low-power state, with a current consumption of 4.8μA. This accounts for the internal RTC used to wake the microcontroller up to perform smart sensing, and the small 4KB of backup RAM used to temporarily store the data acquired by smart sensors as well as network topology information. Thus:

$$E_{\mu C,ds} = 14 \times 10^{-6} \cdot \tau_{off} \quad (11)$$

$$E_{mram,ds} = 0 \quad (12)$$

$$E_{tsc,ds} = 2.5 \times 10^{-6} \cdot \tau_{off} \quad (13)$$

5.4 Complete model

To derive the complete model, in addition to the contributions considered in the three previous sections, it is necessary also to account for the energy absorbed by the ancillary circuitry, mainly the voltage regulator. A rough estimate of this contribution yields a value below 12μW, i.e. 12pJ/μs. It is worth noting that this contribution is only significant during the Deep Sleep phase, while is negligible in the Full Active and Smart Sensing phases.

Summarizing the parameters of the complete model are the number of processes n_p, the application time τ_{app}, the sensing time τ_{ss}, the time for which the system is kept in Deep Sleep mode τ_{off} and the number of times n_{ss} the system enters the smart sensing mode while in one period of Deep Sleep. Figure 9 shows the relation among these times. In particular, recalling Equation (1), it turns out that:

$$\tau_{on} = 944 + 8740 \cdot n_p + \tau_{app} \quad (14)$$

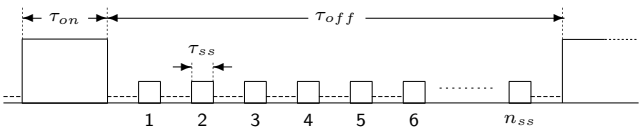

Figure 9: Timing parameters for characterization

while the time the system is in Deep Sleep mode is given by:

$$\tau_{ds} = \tau_{off} - n_{ss}\tau_{ss} \quad (15)$$

The overall energy consumption for one period is thus:

$$E_{on} = 65 + 398 \cdot n_p + 0.043 \cdot \tau_{app} + n_{ss}(12 + 0.011\tau_{ss}) \quad (16)$$

$$E_{off} = 29 \times 10^{-6} \cdot \tau_{ds} \quad (17)$$

$$E_{tot} = E_{on} + E_{off} \quad (18)$$

and the effective duty cycle, including smart sensing, is:

$$DC = (\tau_{on} + n_{ss}\tau_{ss}) \,/\, (\tau_{on} + \tau_{off}) \quad (19)$$

6. EXPERIMENTAL RESULTS

The proposed approach has been assessed against both high-end and ultra-low power architectures in order to determine the power consumption trend, and consequently the expected lifetime, by varying the duty cycle, the sampling rate and the number of running processes.

To perform time measurements for the phases described in Section 5 we used a Tektronix TDS2014B oscilloscope with 100 MSa/s, relying also on data-sheet values for a few time measures, such as the PLL lock-time. For current consumption figures we used data provided by device manufacturers, since the board did not permit an accurate direct detection of the current, especially in lowest power states. For timing and energy figures for the architectures we compare against we relied on components data-sheets and reference literature on specific nodes.

A first result concerns the executable programs footprint. Far from being a secondary aspect, the size of the binaries has a significant impact both at the application level—a high number of available processes improves the overall flexibility—and on the energy efficiency of the entire network. Spawning and moving programs across the nodes, in fact, requires transmitting binary images over the wireless interface, and thus impacts the total power consumption.

The reduction of the size of the ELF images, obtained as discussed in Section 4.2.2, thus, is a first step towards energy optimization. Table 2 shows a comparison between the size we obtained for ELF images and that of images produced by the Contiki dynamic linker, as reported in literature [35]. This required minor adaptations of the code examples to fit the Miosix application interface. It should be noted that the resulting code size slightly differs for the two systems due to differences of the instruction sets of the two processors and APIs of the two operating systems. The table reports the code size (in bytes), the standard ELF size and the reduced ELF size, referred to as SELF for Miosix and CELF for Contiki. The introduction of processes carries along an unavoidable overhead due to context switching, again resulting in a potential loss of energy efficiency. To quantitatively evaluate such an overhead we have directly measured the context

	Miosix			Contiki		
Name	Code	ELF	SELF	Code	ELF	CELF
Blinker	98	1154	340	130	1056	361
Tracker	334	1456	548	344	1668	758

Table 2: ELF size reduction (bytes)

Parameter	**TelosB**	**Mica2**	**Imote2.0**
Processor/μC	MSP430	ATmega128L	PXA271
Frequency	8 MHz	8 MHz	13–416 MHz
DeepSleep/Standby	5.1 μA	19.0 μA	387 μA
Active	1.8 mA	8 mA	53 mA
Active+Radio(RX)	21.8 mA	15.1 mA	66 mA
Active+Radio(TX)	19.5 mA	25.4 mA	66 mA

Table 3: Characteristics of the benchmark nodes

switch and scheduling time which turned out to be approximately of 5μs. Assuming time slots of 5ms, the average overhead is between 0.1% and 0.2%, resulting in a negligible energy increase.

The results discussed in the following are based on the energy model presented in Section 5, considering a typical ambient intelligence application collecting environmental measures with a specific *sensing period*. Samples are then aggregated, processed and transmitted with a longer period, referred to in the following as *processing period*. Referring to Figure 9, the sampling period equals τ_{off}/n_{ss}, while the processing period is simply $\tau_{on} + \tau_{off} \approx \tau_{off}$.

To evaluate the energy efficiency of our approach we compared against an ultra-low power node (TelosB), a low-power node (Mica2) and an high-end node (Imote2.0). Figures characterizing the electrical and functional parameters of these benchmark nodes are reported in Table 3, where the current values for the Imote2.0 refer to an operating frequency of 104MHz, used in our evaluation. It should be noted that none of the nodes is characterized considering the power consumption of sensors. This is the most reasonable approach, since sensors can significantly vary from application to application and even different devices for the same physical measure show a very wide spectrum of electrical characteristics. By neglecting such contributions, the lifetimes reported in the following are necessarily overestimates of the actual operating times, but, on the other hand, have the advantage that can be directly compared, since are guaranteed to be as homogeneous as possible.

Though the energy consumed by the sensor themselves is omitted from all models, the time τ_{ss} required for sensing must be considered. During this time, in fact, the microcontroller, though in low-power mode, is active and thus contributes to the overall energy. While in the model the time τ_{ss} has been left as a parameter, to perform quantitative comparison we have set it to 3ms, since analog sensors exist that can reach a steady state condition within this time. It should also be noted that increasing this time to 10ms, i.e. more than three time as much, increases the overall energy consumption of less the 27%.

The time the system remains fully active performing actual processing has been scaled according to the operating frequency of the four architectures as well as the data transmission rate of the related transceivers. We roughly estimated the total amount of data exchanged through the radio channel during one active period in 8Kbit, accounting also

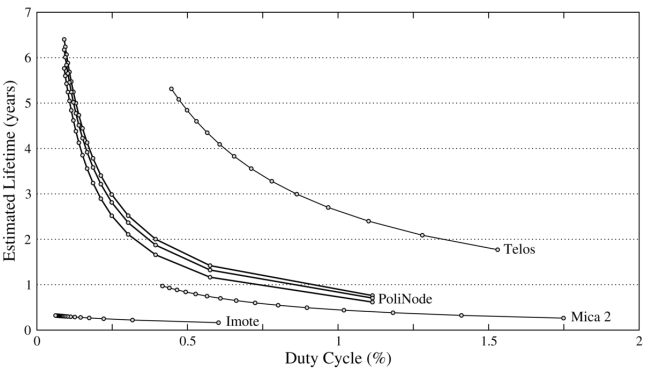

Figure 10: Lifetime versus duty cycle

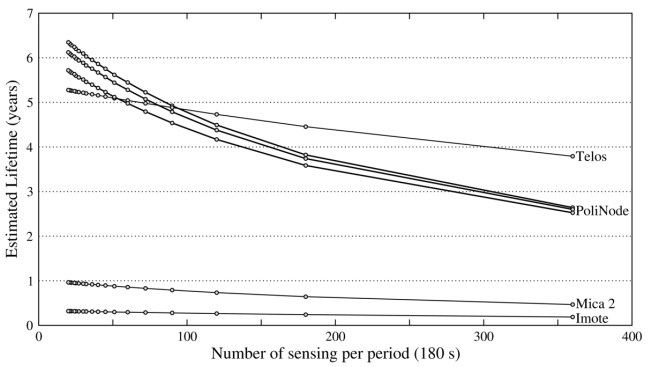

Figure 11: Lifetime versus number of samples

for forwarding operations as well as possible packet retransmission due to collisions, delays or obstacles in the broadcast radio channel.

Figure 10 shows the resulting lifetime for a test case with fixed sensing period and varying processing period. The effect of changing the processing period and consequently the number of samplings in one such period can be summarized by the overall duty cycle. It must be noted that the PoliNode performs sensing operations by means of smart drivers, while the other nodes resort to the typical approach that requires bringing them to the normal active mode[2] The figure reports three sets of data for the PoliNode corresponding to applications with 1, 2 and 4 processes, the lower curve referring to the case of 4 processes. The difference between the three data sets, as can be deduced from the model, is essentially related to energy contributions for swapping in and out the data images of the processes. Since other nodes do not support hibernation, no swapping is necessary and the energy does not vary with the number of running tasks.

A different view of the energy performance of the proposed solution is to set the processing period to a fixed value and changing the sampling period. The chart in Figure 11 refers to a 180s processing time and shows the lifetimes estimated varying the number of samples per processing period from 20 to 360, that is considering sampling period from 500ms to 9s. Again, the three data sets for the PoliNode refer to 1, 2 or 4 processes. As the estimates show, for a number of samples between 50 and 90 the proposed approach reaches

[2]For this reason, fixing the parameters of the application results in different duty cycles for the different platforms.

the same energy efficiency of an ultra-low power node and outperforms it for an even lower number of samples. Increasing the number of samples, on the other hand, the ultra-low power solution shows its superior performance.

Far from trying to reach such a performance level, the proposed solution demonstrated reasonably close to ultra-low power nodes and much more efficient than low-power and high-end ones. It is worth noting that the developed hardware/software architecture offers functional and high-level features such as reprogramming, code mobility and ease of development that are typical of high-end nodes.

7. CONCLUSIONS

This paper presented an hardware/software architecture for enabling ultra-low power performances on scalable and reasonably powerful WSN nodes. A WSN node and operating system have been implemented according to the proposed approach to evaluate its feasibility and the associated energy and functional performance. It should be noted though, that the same approach is also suited for developing general embedded system platforms. Experimental results demonstrated that the proposed architecture shows energy consumptions and lifetime values in the same order of magnitude of those of ultra-low power architectures, but providing, at the same time, enough memory and computational resources to enable a general-purpose like programming and development style, typical of high-end powerful and power-consuming nodes.

8. REFERENCES

[1] C. Brandolese and L. Rucco. Optimization of functional allocation to maximize the lifetime of wireless sensor networks. In *Wireless Communication and Sensor Networks (WCSN), 2010 Sixth International Conference on*, pages 1 –6, dec. 2010.

[2] J. Polastre et al. Telos: enabling ultra-low power wireless research. In *IPSN 2005, ACM/IEEE*, pages 364 – 369, april 2005.

[3] J.L. Hill and D.E. Culler. Mica: a wireless platform for deeply embedded networks. In *Micro, IEEE*, volume 22, pages 12 – 24, nov/dec 2002.

[4] A. Rowe et al. Firefly mosaic: A vision-enabled wireless sensor networking system. In *IEEE RTSS 2007*, pages 459 –468, dec. 2007.

[5] JeongGil Ko et al. Egs: A cortex m3-based mote platform. In *IEEE SECON 2010*, pages 1 –3, june 2010.

[6] Crossbow. Imote2 hardware reference manual, 2007.

[7] Oracle-Sun. Sun small programmable object technology (sun spot) owner' s manual release 3.0, October 2007.

[8] Adi Mallikarjuna V. Reddy et al. Wireless sensor network operating systems: a survey. In *Int. J. Sen. Netw.*, volume 5, pages 236–255, August 2009.

[9] René Müller et al. Swissqm: Next generation data processing in sensor networks. In *Proc. of CIDR 2007*, pages 1–9, 2007.

[10] A. Boulis and M.B. Srivastava. A framework for efficient and programmable sensor networks. In *Proc. of IEEE OPENARCH 2002*, pages 117 – 128, 2002.

[11] D. Janakiram et al. Comis: Component oriented middleware for sensor networks. In *Proc. of LANMAN '05*. IEEE Computer Society, 2005.

[12] Liviu Iftode et al. Programming computers embedded in the physical world. In *International Standard ISO*, volume 8327, 1987.

[13] J. Lilius and I. Paltor. Deeply embedded python, a virtual machine for embedded system. Technical report, Turku Centre for Computer Science.

[14] Chien-Liang Fok et al. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *Proc. of ICDCS '05*, pages 653–662. IEEE Computer Society, 2005.

[15] J. Jeong et al. Network reprogramming. In *Berkeley University, at http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/NetworkReprogramming.pdf*.

[16] T. Stathopoulos et al. A remote code update mechanism for wireless sensor networks. Technical report, 2003.

[17] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. of SenSys '04*, pages 81–94. ACM, 2004.

[18] Prabal K. Dutta et al. Securing the deluge network programming system. In *Proc. of IPSN '06*, pages 326–333. ACM, 2006.

[19] J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pages 354 – 365, jan.-2 feb. 2005.

[20] Niels Reijers and Koen Langendoen. Efficient code distribution in wireless sensor networks. In *Proc. of WSNA '03*, pages 60–67. ACM, 2003.

[21] Jaein Jeong and D. Culler. Incremental network programming for wireless sensors. In *IEEE SECON 2004*, pages 25 – 33, oct. 2004.

[22] Ting Liu et al. Implementing software on resource-constrained mobile sensors: experiences with impala and zebranet. In *Proc. of MobiSys '04*, pages 256–269. ACM, 2004.

[23] Jochen Schiller et al. Scatterweb - low power sensor nodes and energy aware routing. In *Proc. of HICSS '05*, page 286.3. IEEE Computer Society, 2005.

[24] Pedro José Marrón et al. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *Proc. of EWSN 2006*, pages 212–227, 2006.

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

[26] Philip Levis et al. The emergence of networking abstractions and techniques in tinyos. In *Proc. of NSDI'04 - Volume 1*, pages 1–1. USENIX Association, 2004.

[27] A. Sinha and A. Chandrakasan. Dynamic power management in wireless sensor networks. In *Design Test of Computers, IEEE*, volume 18, pages 62 –74, mar/apr 2001.

[28] Chanik Park et al. Compiler-assisted demand paging for embedded systems with flash memory. In *Proc. ACM EMSOFT '04*, pages 114–124. ACM, 2004.

[29] Lin Gu and John A. Stankovic. t-kernel: providing reliable os support to wireless sensor networks. In *Proc. of SenSys '06*, pages 1–14. ACM, 2006.

[30] Siddharth Choudhuri and Tony Givargis. Software virtual memory management for mmu-less embedded systems. Technical report, 2005.

[31] Lan S. Bai et al. Memmu: Memory expansion for mmu-less embedded systems. In *ACM Trans. Embed. Comput. Syst.*, volume 8, pages 23:1–23:33. ACM, April 2009.

[32] Yu-Ting Chen et al. Enix: a lightweight dynamic operating system for tightly constrained wireless sensor platforms. In *Proc. of ACM SenSys '10*, pages 183–196. ACM, 2010.

[33] Girts Strazdins et al. Mansos: easy to use, portable and resource efficient operating system for networked embedded devices. In *Proc. of SenSys '10*, pages 427–428. ACM, 2010.

[34] Doug Simon and Cristina Cifuentes. The squawk virtual machine: Java on the bare metal. In *Conference on Object Oriented Programming Systems Languages and Applications*, page 150, 2005.

[35] Adam Dunkels et al. Run-time dynamic linking for reprogramming wireless sensor networks. In *ACM SenSys '06*, pages 15–28. ACM Press, 2006.