Energy-Efficiency Through Micro-Managing Communication and Optimizing Sleep

A. Caracaş, C. Lombriser, Y.A. Pignolet,
T. Kramp, T. Eirich, R. Adelsberger
IBM Zurich Research Laboratory
(xan,cll,yvo,thk,eir,rad)@zurich.ibm.com

U. Hunkeler
École Polytechnique Fédérale de Lausanne
urs.hunkeler@epfl.ch

***Abstract*—Energy-efficiency is key to meet lifetime requirements of Wireless Sensor Networks (WSN) applications. Today's run-time platforms and development environments leave it to the application developer to manage power consumption. For best results, the characteristics of the individual hardware platforms must be well understood and minutely directed. An Operating System (OS) with suitable programming abstractions can micro-manage power consumption of resources. We demonstrate with the Mote Runner platform how the inherent overhead of managed application code is compensated for by a platform-independent communication API together with sleep optimizations. The proposed abstractions and optimizations can be applied to other modern sensor network platforms. To quantify the effectiveness of our approach, we measured the energy efficiency of a real-world WSN application using a custom TDMA communication protocol fully implemented on both Mote Runner and TinyOS. Mote Runner's power management and sleep phase optimizations outperforms TinyOS in our test application for duty cycles below 10% on the Iris hardware.**

I. Introduction

Energy-efficiency is crucial for all types of embedded systems that rely on batteries or energy harvesting as their power source, with wireless sensor networks for long-time monitoring being the prime example. Typical applications are glacier monitoring [1], precision farming [2], [3], water management [4], heritage monitoring [5], or soil vibration monitoring at volcanic sites [6]. Such scenarios describe remote and rough sensing environments. Physical access to the motes and their batteries is often not an option and the sensor nodes have to run on their initial set of batteries or on harvested energy (often not spread evenly in the network and in many cases not satisfying the demand). For a long lifetime, energy must be minutely managed during all phases of operation, including the energy spent while sleeping.

The energy expenditure is mainly influenced by the decisions programmers make when developing the applications running on the sensor nodes. At the same time certain energy is spent by the OS without a direct influence from the developers.

TinyOS [7] is the predominant native operating system for sensor networks. The platform is based on a modular design where components can be layered and are wired at compile time. Applications are written in nesC, a C dialect with full control of hardware peripherals. The TOSSIM [8] simulation environment and extensions support application development. Different communication protocols are available for TinyOS and the default collection tree protocol [9] uses CSMA. We have implemented a custom TDMA protocol to meet our application requirements.

Mote Runner is a sensor network platform where virtual machine (VM) and OS are merged. This combination makes the application independent from the underlying hardware platform. The platform offers native support for delegates as a callback mechanism [10] for efficient reactive programming. Developers can use well-known high-level programming languages such as Java or C# in combination with standard compilers. A tool-chain converts the compiler's output into optimized byte codes. Mote Runner integrates network simulation, source-level debugging, and accurate time and power traces in a coherent development environment. The advantages of VM-based approaches, the architecture and design decisions for the Mote Runner platform compared against other embedded VMs are discussed in [11].

The cost of VM-based approaches is typically lower performance due to the overhead of byte code execution. In this paper we focus on how the effect on power consumption can be minimized by enabling micro-management for the run-time platform using a time-based API for radio operations. Further optimizations include the deep sleep phase. The API abstraction as well as the sleep optimizations are generic and can be applied to other OSs and sensor network platforms.

As a show case, we analyze and measure the power consumption of a complex, real-world application running on Iris motes [12]. The application monitors seismic vibrations in an industrial setting using a custom sensor board and transmits the data collected via a custom TDMA network protocol to a base station. To avoid bias, two independent teams developed the same functionality including the complete communication protocol, one written in Java for the Mote Runner run-time platform [11] and one written in nesC for TinyOS [7]. We analyze the overhead introduced by the Mote Runner VM and show that its automatic power management can outperform the solution based on TinyOS for low duty cycles.

The remainder of this paper is organized as follows: In Section II we present our power-measurement findings for core operations on the Iris mote hardware. These measurements serve as a prerequisite for optimizing application-level energy expenditure for both communication and the sleep phase which we describe in Section III. In Section IV, we measure the real-

OPERATION	COST (μAs)
MCU	
MCU POWER SAVE (deep sleep) [1] for $1s$	9.5
MCU EXTENDED STANDBY for $1s$	280
MCU IDLE w/ ALL peripherals DISABLED for $1s$	2390
MCU IDLE w/ TIMER1[2] ENABLED for $1s$	2460
MCU IDLE w/ TIMER2 ENABLED for $1s$	2510
MCU IDLE w/ ALL peripherals ENABLED for $1s$	3390
MCU ACTIVE w/ ALL peripherals DISABLED for $1s$	5760
MCU ACTIVE w/ TIMER1[2] ENABLED for $1s$	5880
MCU ACTIVE w/ TIMER2 ENABLED for $1s$	5920
MCU ACTIVE w/ ALL peripherals ENABLED for $1s$	6720
Radio	
Radio SLEEP for $1s$	0.02
Radio TRX_OFF (idle) for $1s$	1360
Radio PLL_ON (active) for $1s$	7490
Radio RX_ON (receive) for $1s$	15510
Radio BUSY_TX (transmitting) w/ max power for $1s$	16520
Iris	
Receiving[3] a 13 byte message (608 μs)	12
Receiving[3] a 127 byte message (4256 μs)	81
Transmitting[4] a 13 byte message (608 μs)	13
Transmitting[4] a 127 byte message (4256 μs)	87
Intermittent wake-up ($\sim 4.3ms$)	6.4

Table I
AVERAGE MEASUREMENTS COMPARING THE COST OF DIFFERENT OPERATIONS ON THE IRIS MOTE PLATFORM.

world energy consumption of the monitoring application on both Mote Runner and TinyOS. Related work is discussed in Section V before we conclude the paper in Section VI.

II. Energy Characteristics

To handle power management optimally on behalf of applications, an operating system requires detailed knowledge about the power consumption of primitive operations and individual system components in their various states. In this section we describe our findings for the Iris mote hardware. Similar measurements have been done for other mote platforms [13], [14]. In addition to previous measurements, our measurements show the cost of an intermittent wake-up from deep sleep which is discussed in Section III-B. Also note that the idle (TRX_OFF) radio state is ten times less expensive than using the receiving (RX_ON) state and should be used as default state for the radio as discussed in Section III-A. While in this paper we use the Iris mote as an example, the presented concepts are applicable to other modern sensor platforms.

First we focus on the current drawn for various state combinations of the radio and micro-controller (MCU) of Iris motes [12]. The Iris is equipped with an ATmega1281 MCU [15] running at 8 MHz, with 128 KB Flash memory, 8 KB of RAM, and the RF230 radio-chip [16] for communication. Our measurements with a reference voltage of 3 V are listed in Table I. The transition between radio states are not instantaneous and in our case typical values are specified in [16].

Note that the current drawn for the whole system in deep sleep mode (i. e. when the MCU is in POWER_SAVE mode and the radio is in SLEEP mode) is four orders of magnitude smaller than the current drawn when operating the radio (i. e., either transmitting or receiving a message). Because of such an immense difference in energy consumption, a commonly used strategy for saving energy is to keep the radio system in SLEEP mode for as long as possible, wake up only just before a radio operation is required, and then sleep again as soon as possible. To understand the power consumption of radio operations better, Table I further shows the average consumed charge when receiving and sending (without acknowledgment) a message of minimum and maximum length (given in bytes), respectively, measured on the Iris platform.

III. Optimizing Power Consumption

The decisions developers make influence the power consumption of the resulting application. Such decisions involve radio communication which is particularly expensive as we have seen in the previous section.

Micro-managing power manually is a hard task and involves measuring and calculating all hardware-specific transition times. Depending on the remaining time until an operation, it might payoff to completely power-off the radio for a period as short as 100 μs. The OS can automatically manage such micro transitions if the API allows to express the exact intentions of the applications using precise points in time.

Our discussion uses the Iris platform as an example, yet the presented concepts are generic and can be applied to other modern sensor platforms such as MicaZ, TelosB, etc.

A. Radio Micro-Management

In this section, we introduce a platform-independent abstraction for the radio API which enables the OS to minutely manage the power state of the radio chip on behalf of the application. Developers implementing communication protocols can focus on protocol functionality while energy management decisions and respective transitions are automatically handled by the radio abstraction in the OS, even for very short time intervals. The main goal of the radio abstraction is to simplify development effort and improve energy efficiency.

As a main design criteria our abstraction generally includes time values to schedule critical, externally observable actions such as radio transmissions. This empowers the run-time to manage resources in the most energy-efficient way on the given hardware. Developers no longer need to be concerned with manual power management. The key difference is that developers simply state the exact time when some action should be performed, without knowing or guessing hardware-specific transitions. It is the run-time that ensures devices such as the radio are operational and in the desired state when the application requires them. By default, the micro-controller,

[1] To wake-up the MCU, TIMER2 is clocked from the external 32 kHz crystal. No other peripherals are active in this mode.

[2] When the MCU is ACTIVE or IDLE, TIMER1 is clocked from a 8 MHz oscillator and is typically used for time operations with μs granularity.

[3] Radio is typically in RX_ON or BUSY_RX mode and the MCU is in IDLE mode to be able to react to interrupts from the radio and timestamp received messages with the local system time.

[4] Radio is typically in BUSY_TX mode and the MCU is in IDLE mode to be able to react to interrupts from the radio and possibly timestamp sent messages with the local system time.

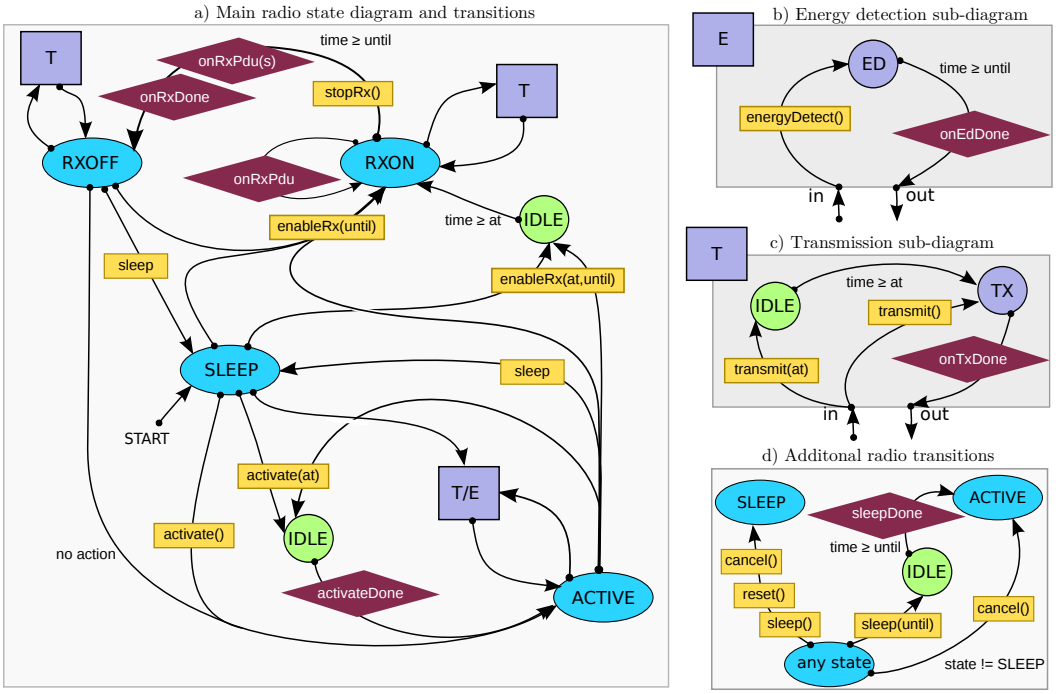


Figure 1. (a) Main radio state diagram used for micro-managing the radio power consumption based on the application requests using time information. (b)-(c) Sub-diagrams T and E are embedded in the main diagram (a) as squares. (d) Additional radio state transitions. Generic radio states managed by the OS are shown using ellipses (stable states) and circles (volatile states). API function calls are depicted using rectangles while callbacks to application code are shown with rhombuses. API methods take precise points in time as parameters while callbacks report the exact completion time and status.

radio, and all sensor devices are kept in the most energy-efficient power mode with the decision of when to power on and off certain devices being made on a microsecond basis depending on the API calls from the user application. Transitions between radio states are not instantaneous, such delays are platform-specific and can be configured based on hardware measurements. For example on the Iris platform it takes $\sim$880 μs for the radio to transit from the SLEEP to the TRX_OFF state. This configuration is part of the OS implementation and not the application code.

Our radio abstraction consists of the following elements: a) internal operating states which may map to different platform-specific radio-chip states, b) method calls which trigger some action which changes the internal state, c) callbacks which notify application code about completed events, and d) internal state changes due to passing of time. The radio abstraction ensures that actions are performed, callbacks are invoked and the internal state is changed in time to meet the deadlines requested by the application code. Normally, an application starts a task (e.g. transmit) by calling an API method. The task execution takes some time and upon completion an application-specific callback will be invoked by the OS.

Figure 1 shows all possible radio operation states and respective transitions. The states are abstract but can be mapped to different modern radio-chips such as the RF23x or CC24xx. In this discussion we use the RF230 radio chip as an example.

We distinguish between stable and volatile operating states. Individual methods of the radio API may be called only if the radio is in a stable state. Calling methods while the radio is in a volatile state, where this operation is not permitted, will result in an exception being thrown.

The stable states are shown using ellipses: SLEEP (sleeping), RXON (background reception of messages), RXOFF (reception period ended), and ACTIVE (ready to quickly transmit or receive). The volatile states are shown using circles: TX (transmit), ED (energy detection), and IDLE (preparing the execution of an action). The T and E sub-diagrams are depicted using squares and show the transitions performed when the respective transmit and energy detection actions are requested. The sub-diagrams blocks are expanded in Figure 1(b)-(c) and are collapsed and embedded in the main radio state diagram in Figure 1(a) to simplify the graphical representation.

In addition, from any state, all operations may be canceled and the radio may be instructed to go to sleep. A sleep request may take a point in time as a parameter which represents the time until which the radio should be kept asleep. In this particular case, the OS ensures that at the specified time the radio is in the ACTIVE state and the callback `sleepDone` is invoked, cf. Figure 1d).

To save energy, the initial operation state for the radio is SLEEP (directly mapped to the same state from Table I). From this state, the application may request one of the following actions: `activate` the radio, `transmit` a message, enable the receiver (`enableRx`), or perform an energy measurement on the radio channel (`energyDetect`).

Applications request certain actions using API function calls which are depicted using rectangles in Figure 1. API calls take a point in time as a parameter which specifies the precise

time at which the action should be executed. Without a time parameter the start of the action is triggered immediately. In the case of a transmission, this parameter represents the exact time when the transmission of the radio frame must start. Further energy considerations allow to select the power level for each transmission individually. To simplify application development a transmission mode can be specified such as a transmission with a single clear channel analysis (CCA), or a back-off mechanism such as carrier sense multiple access (CSMA) with corresponding parameters. The OS will take care of starting any necessary wake-up procedures sufficiently ahead of time so that the desired target state is entered and the requested action is started at the specified point in time. Such an abstraction enables the same application code to execute on heterogeneous platforms. Moreover, a simulation environment can use the same timings for radio transitions to match the behavior of the correspnding hardware platform.

The generic, volatile IDLE state is entered whenever a scheduled action with a specified execution time is ahead. This state may map to any of the actual RF230 radio-chip power states (SLEEP, TRX_OFF, or PLL_ON from Table I) depending on the time remaining until the time of execution, the current state of the radio hardware, the desired hardware state, and the hardware-specific transition times. For energy considerations, on our example platform, the ACTIVE state corresponds to the TRX_OFF state in Table I.

Applications are notified of completed actions using callbacks which are depicted in Figure 1 using rhombuses. Before the callback is invoked (i.e. control is passed to application code) the radio transits to the next state, according to the state diagram. In other words, when a callback is invoked the radio operation state is indicated by the arrow head in Figure 1. For example, the radio is already in the ACTIVE state when the `activateDone` callback is invoked. During the callback the application code may request a radio action which will trigger the respective internal transitions. If nothing is requested the OS leaves the radio in the same operation state, because it cannot predict what the application will do next. After the callback is invoked the radio operation state might change depending on the actions specified by the application code.

For applications to appropriately recover from failures, they must be informed if certain requests failed and what the exact reason for the failure was. Thus, callbacks are always executed and report the completion status of the requested action. The completion status may have several values depending on the requested action. For example, in case of a transmission several errors could happen: no channel access, the time specified for the transmission could not be met, no acknowledgment was received, etc. An exception to this rule are the `cancel` and `sleep` calls without a time parameter: they reset the state of the radio and remove any pending callbacks.

To simplify application scheduling and state maintenance, callbacks report the exact timestamp for the completion of the task or the occurrence of the respective event.

To save energy, energy spent while receiving messages should be minimized. Thus, the radio API allows to specify a precise start time when to reach the RXON state and an end time when to exit the RXON state using the `enableRx(at,until)` method.

The `TX` and `ED` states represent volatile radio operation states, where the radio chip is transmitting respectively measuring energy on the wireless channel. When the respective callback is invoked the radio returns to the originating state. For example if transmitting a message from the RXON state, after the transmission is finished and before the callback is invoked the radio is set back to the RXON state. If the application decides to transmit a message from the RXOFF state the radio is put back to the RXOFF state after the transmission is complete.

To complete the API further functions are required such as setting the radio channel used for communication, querying and setting radio address information used for filtering. The full description of these functions is omitted here due to space limitations[5]. As a general rule these calls are synchronous and an application may invoke them only in a stable state.

The radio abstraction described in this section is fully implemented in the Mote Runner platform. Yet, a similar implementation is possible for other sensor network operating systems such as TinyOS [7] and Contiki [17]. We will show that this API enables a VM to perform a TDMA as efficient as a custom OS in terms of energy expenditure.

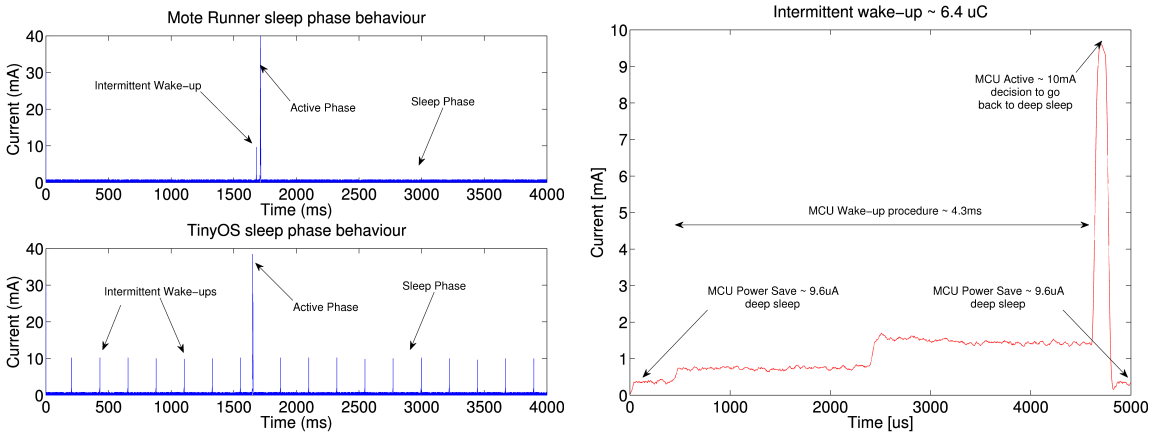
B. Sleep Energy

Much of the literature seems to neglect that a mote cannot remain in deep sleep mode for extended time periods. Sleep periods are governed by the maximum sleep time span the hardware timer allows. When this timer overflows, the software counter which keeps track of the local system time must be updated. Therefore, the mote must wake up intermittently to update its local time. Typically these intermittent operations include computing the next future deadline when to wake up again, picking a comparator value for the timer module, and finally updating the logical system time. For the ATmega1281 this has to happen at least every eight seconds. Figure 2(a) shows such an intermittent behavior for an Iris mote using Mote Runner and TinyOS respectively. The tall spikes represent a period of useful activity, i.e. sending or receiving a radio message. The smaller spikes represent intermittent wake-ups during the sleep phase.

Our measurements show that the energy consumed in a single intermittent operation is $6.4\mu C$, half the cost of transmitting or receiving a small radio message of 13 bytes (cf. Table I). The energy consumed by an intermittent wake-up is mainly due to the micro-controller wake-up sequence[6] which ensures a stable operation voltage and clock. While this sequence is in progress, the current drawn is gradually increased. On the Iris platform, the wake-up sequence takes on average $4.3\,ms$. During an intermittent wake-up the MCU

[5]The full API is available in the Mote Runner documentation [11]

[6]A wake-up sequence starts when an interrupt condition is met and completes when the MCU is active and calling into the interrupt routine.



(a) (b)

Figure 2. Sleep behavior and communication activity (a) measured on an Iris mote for Mote Runner (upper signal) and TinyOS (lower signal) respectively. A typical intermittent wake-up (b) consumes half of the energy of transmitting a 13 byte radio message.

is in ACTIVE mode for less than 100 μs to update the software system time. Subsequently the MCU returns to deep sleep.

Figure 2(b) shows a detailed view of a typical intermittent wake-up for the Iris mote, where the y-axis represents the current drawn in mA and the x-axis represents the time in μs. The sharp peak represents the period during which the micro-controller is active, computing the required values for the next wake-up. The slow rising curve with two levels represents the micro-controller wake-up sequence. The consumed charge is the integral of the curve.

The charge consumed while sleeping $q_{sleep}(t)$ for a time interval t given in seconds is a function of the charge consumed during intermittent wake-ups $q_W(t)$ and the charge consumed in the deep sleep periods $q_S(t)$:

$$\begin{aligned} q_W(t) &= N(t) \cdot Q_W \\ q_S(t) &= [t - N(t) \cdot T_W] \cdot I_S \\ N(t) &= \left\lceil \frac{t}{T_{MAX}} \right\rceil \\ q_{sleep}(t) &= q_S(t) + q_W(t) \end{aligned} \quad (1)$$

Q_W denotes the charge consumed by a single intermittent wake-up ($Q_W = 6.4\,\mu C$), T_W denotes the duration of the intermittent wake-up ($T_W = 4.3\,ms$), and I_S denotes the current drawn during sleep ($9.5\,\mu A$). The values for these constants have been measured on the Iris hardware platform. The number of intermittent wake-ups for a given time period, $N(t)$, depends on the maximum time span for which an implementation may sleep without interruptions, T_{MAX}. This constant is limited by the hardware features which determine the longest sleep interval without imposing a wake-up. These features are: the frequency of the clock used during sleep, the largest timer/counter value, and the largest pre-scaler[7] for the respective timer. For example, for the ATmega1281, an 8-bit platform with a clock frequency of 32768 Hz (in deep sleep mode) and with the largest pre-scaler value of 1024, the longest sleep interval is approximately $\sim 8s$.

The total power consumption decreases if motes wake up less frequently. We denote by η the ratio between energy consumption during wake-ups and the total energy spent in sleep periods. It follows that the theoretical limit of the wake-up to sleep energy ratio η is given by:

$$\eta = \frac{q_W(t)}{q_{sleep}(t)} = \frac{Q_W}{Q_W + I_S \cdot (T_{MAX} - T_W)}. \quad (2)$$

Using our measurements, this energy ratio is at least $\eta \geq 7.7\%$, i.e. at least 7.7% of the total sleeping energy is spent waking up and handling the clock. In order to save energy while sleeping, the number of intermittent wake-ups is to be minimized. An optimized sleep strategy reduces the energy consumption and is independent of the application or communication protocol used.

The Mote Runner implementation computes a wake-up schedule with dynamic values for the clock pre-scaler and respective comparator value. A precomputed schedule reduces the work of an intermediate wake-up to a single table look-up and immediately going back to sleep. To this end, a knapsack optimization problem minimizing the number of items has to be solved. Items represent wake-ups and they have a finite number of sizes given by the combination of pre-scaler and comparator values. Computing such a schedule introduces a variable delay which must be accounted for when synchronizing with the local system time. To measure this delay we use a hardware timer with a μs granularity.

When switching the MCU between deep sleep and active mode the two respective clocks (given by the external low and internal high frequency) must be synchronized to maintain an accurate local system time. Switching between these clock sources includes clearing the state of the pre-scaler, dealing with possible overflows, and waiting for the two clocks to

[7] A pre-scaler divides the oscillator frequency to allow the timer/counter to reach longer time spans which controls the maximum sleep time.

synchronize. During the synchronization process the OS ensures the MCU is kept in a power-saving state (either POWER SAVE or EXTENDED STANDBY from Table I).

The sleep schedule computations and system time synchronization are time critical operations. The hardware peripherals used for these operations must be shielded from application code. Implementing such a sleep strategy is possible, but challenging, on embedded platforms which allow applications to control hardware resources. In this case applications striving for energy efficiency must to be omniscient to avoid, for example, disabling a hardware timer which another component requires for sleep operations. Such errors in the sleep implementation may adversely affect time synchronization. We implemented the optimized sleep strategy for the Mote Runner OS/VM because it provides encapsulation by design.

Our discussion used the Iris mote as an example. However, the described sleep optimizations can be implemented on other modern hardware platforms. A necessary requirement is a dynamic and deterministic pre-scaler to control the external low-power clock.

IV. Application Evaluation

To show the effectiveness of the presented abstractions we implemented them on the Mote Runner platform which combines VM and OS. In our case, the main goal of the optimizations is to counterbalance the overhead of a VM when compared to native implementations. We compare the total energy consumption of a Mote Runner implementation and respectively a TinyOS implementation of the same application functionality.

For comparison we selected a monitoring application under development at IBM. The task of the application is to observe soil vibrations of an area and to determine their maximum velocity and frequency. Motes report the aggregated values of 100 bytes every 10 seconds to a base station and to issue alerts in case certain thresholds are surpassed. The system has been designed and tested to run for several months to years.

For energy consumption and sensing quality considerations, a node consists of an Iris mote to run the network stack and a custom sensor board featuring vibration sensors and a low-power FPGA for signal processing. The sensor board is running continuously to process the signal data, and periodically passes the computed vibration parameters to the Iris mote. As the sensor board's power consumption is the same for both systems, our comparison focuses only on the Iris mote running the network stack.

The networking stack is a hierarchical TDMA-based protocol with a central controller imprinting a routing tree and schedule onto the network. Within a $10\,s$ TDMA superframe, each network node has slots assigned to synchronize with its routing tree parent, to synchronize its children, for collecting the children's messages, and to forward them to the parent, including its own data. The rest of the time is spent asleep.

The networking stack has been implemented by two independent teams for TinyOS and for Mote Runner to compare its performance. We measured the black-box energy consumption of these two implementations with the same code structure and approximately the same number of code lines. To analyze the total power consumption we first measure the consumed energy in the sleep phase for the two implementations.

A. Sleep Phase (Application-Independent)

An important difference between TinyOS and Mote Runner is the number of wake-ups during the sleep phase, cf. Figure 2(a). TinyOS wakes up in regular $T_{MAX} = 225\,ms$ intervals until the specified deadline. This behavior stems from a sleep strategy with a fixed pre-scaler and counter values. On the other hand, Mote Runner wakes up only once, sleeping for up to $T_{MAX} = 8\,s$ based on our dynamic sleep strategy described in Section III-B. By inserting these numbers in Equation (2) we obtain the ratio of wake-up to sleep energy for TinyOS, $\eta_{TOS} = 75\%$, and Mote Runner, $\eta_{MR} = 7.7\%$ respectively. In the case of TinyOS almost ten times more energy is spent in the intermittent wake-ups compared to the total energy spent in sleep periods.

B. Active Phase (Application-Specific)

The active phase represents the core functionality of a sensor network application. This part typically involves sending, receiving, or forwarding messages, sampling and applying filtering functions on the sensor data.

Computations on top of a VM are by definition more expensive than native code. This price is paid for the benefits of virtual homogeneity, managed code, ease of debugging, and maintenance capabilities such as dynamically updating applications. Yet, for applications whose behavior is not clearly dominated by computations, an efficient VM implementation and smart sleep strategy can result in an overall solution whose overhead is well-controlled and which, from an overall application perspective, can compete with native solutions.

Figure 3 shows the power consumption measured for the active phase within a superframe for the TinyOS and Mote Runner implementations running on the real Iris hardware as well as for the Mote Runner simulation. The active phase includes 5 slots of $50\,ms$ duration during which the mote receives 3 synchronization messages from its parent, sends 3 synchronization messages to its children, then sleeps one slot. In the fourth slot, it collects one message from a child node and forwards it during the fifth slot to the parents, appending its own message. Afterward, the mote sleeps for $10\,s$ until the start of the first slot in the next superframe.

The overhead of VM computations is visible in the wider spikes in the first receive slot in Figure 3: Mote Runner requires more time than TinyOS for processing messages. This overhead is compensated by the radio API which micromanages power as described in Section III-A.

Figure 3(a) shows the power trace for the monitoring application running in the Mote Runner simulation. Before a slot starts, the application decides whether to sleep, enable radio reception, or transmit messages during the respective slot. Peaks in the receive and send slots correspond to radio activity. The Mote Runner simulation runs the same VM/OS

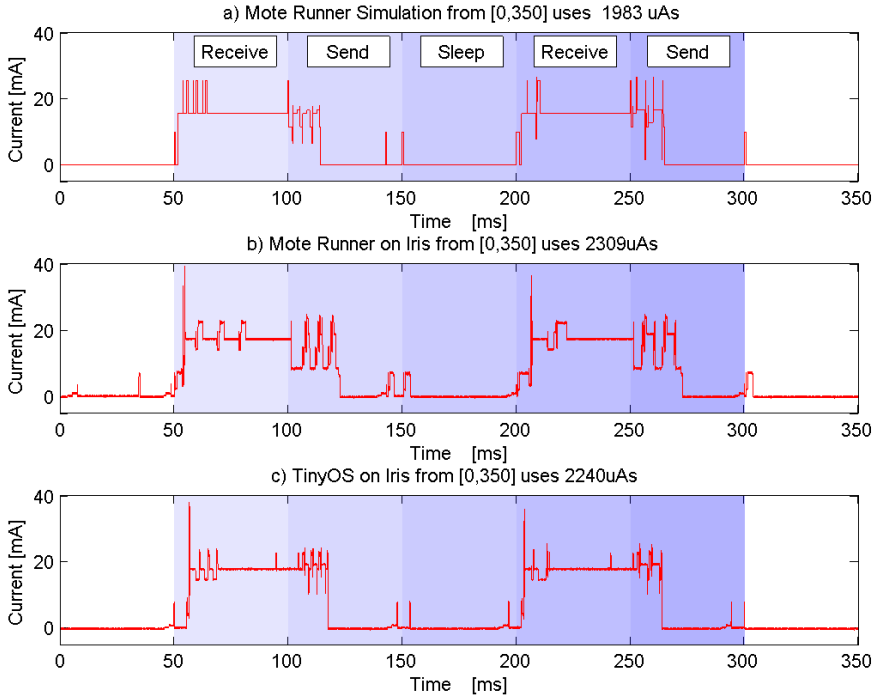


Figure 3. Application active-phase power trace comparison. The Mote Runner implementation uses a VM and in the active phase the application consumes only 4% more energy than the respective TinyOS implementation on the Iris platform. The slower processing of managed code is compensated by using a radio API which micro-manages radio power states and transitions, thus reducing overall energy consumption. Micro transitions are visible in the Mote Runner power trace as there are more levels corresponding to more radio and MCU states than the respective TinyOS power trace.

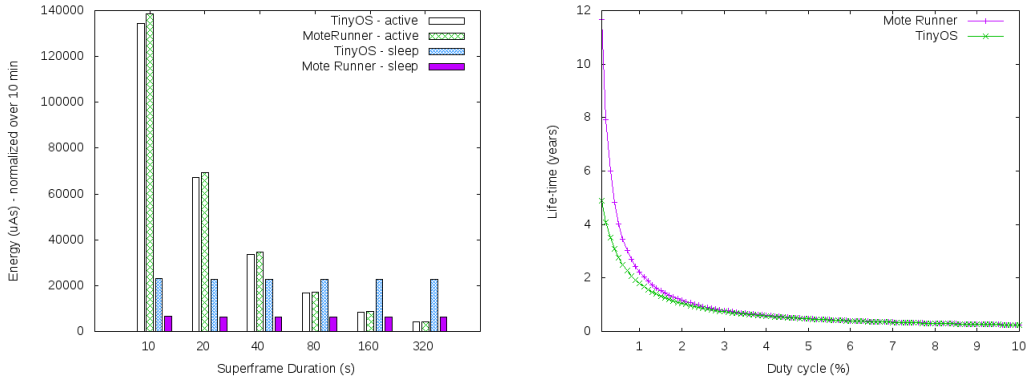
and run-time libraries as the hardware and executes the exact same application byte codes. This allows the simulation to emulate the application execution down to low-level hardware calls like memory copy routines, writing to flash, or radio chip control. It is therefore a close match to the hardware as illustrated in Figure 3(b) and gives the tendency and estimation for the overall power consumption. Simulated execution times are shorter than on the real Iris hardware mainly because of the different cost of native routines.

The power consumption of the Iris mote with the TinyOS implementation is shown in Figure 3(c). This is an example where power management of the radio module was coded by hand in TinyOS while it was automatically done in Mote Runner according to the API described in Section III-A. Before sending a message TinyOS is usually in receive mode.

Mote Runner optimizes this behavior and automatically handles radio-chip state transitions to more efficient power states before the actual transmission. In the Mote Runner power traces these optimizations are visible as there are more power levels than in TinyOS, in particular before radio activity spikes in the send slots. The different power levels show that Mote Runner effectively uses multiple radio and MCU power states to reduce power consumption. For example, in the send phase ([100,150] ms), most of the time before sending messages is spent in the power level TRX_OFF, followed by a short period in PLL_ON.

The wider time span between messages in the Mote Runner power trace ([100-150] ms and [200-250] ms) stems from different APIs to timestamp sent messages when compared with TinyOS. This is an example where in Mote Runner, the programmer has to manually include the send time to the message before sending, thus requiring a sufficient message preparation time, which was set here to 5 ms. On the receiver side, the receive callback provides the exact time for the start of the reception. In contrast, TinyOS has a dedicated message type for synchronization, which reserves 4 bytes in a message, into which it writes a timestamp right upon sending the message to the transceiver. The receiver will timestamp it immediately upon reception. Our measurements show that the two mechanisms are equivalent in terms of synchronization accuracy. Using only one message transmission as a synchronization point, we measured a synchronization error with a standard deviation of 49.4 μs between the sensor nodes at the beginning of each 10 s superframe. This is comparable to the error achieved by TinyOS [18] and close to the 20 ppm drift of the Iris mote's clock source. Mote Runner can thus provide sufficient time accuracy to support more advanced protocols such as FTSP [19] which can further improve synchronization.

In summary, Mote Runner needs more time for processing due to the VM overhead but it remains competitive by micro-managing power using the presented communication abstractions and sleep optimizations.



(a) (b)

Figure 4. (a) Breakdown of total power consumption into sleep and active energy for the Mote Runner and respectively TinyOS implementations for different superframe durations normalized over a period of 10 minutes — the active energy values are obtained from Figure 3. (b) Life-time estimate for a mote powered by batteries (2000 mAh) and running the network stack implementation on Mote Runner respectively TinyOS for different duty cycles.

C. Total Energy Consumption and Life Time

The total energy consumption for an application over a superframe, i.e. the time span t during which the mote sleeps, transmits, and performs functional activity, is the sum of energy consumption over the active (q_{active}) and sleep (q_{sleep}) phase. In our concrete case, q_{active} corresponds to the measurements in Figure 3. We compute q_{sleep} using Equation 1 and the evaluation results from Section IV-A for Mote Runner and TinyOS respectively. Figure 4(a) shows our results for the total energy consumption for different superframe durations. For long superframes the application-independent sleep energy dominates the application-specific active energy.

To put power consumption into perspective, we compute the application life-time which depends on the charge consumed during one superframe. The life-time $L(t)$ can be expressed as the fraction between the battery capacity C and the sum of consumed charges over a superframe duration t as $L(t) = C/(q_{sleep}(t) + q_{active})$.

We assume the sensor nodes are powered by two off-the-shelf AA-sized lithium batteries with a capacity of 2000 mAh or 7.2 GC charge. Figure 4(b) plots the results for the expected life-time of the Mote Runner and TinyOS implementations for the network stack under scrutiny. In this case, the break-even point where the Mote Runner sleep strategy pays off is for sleep periods of at least 2.5 s, corresponding to duty cycles lower than 10%. Thus, lower duty cycles, further improve the life-time of the Mote Runner implementation in comparison to the TinyOS implementation.

V. Related Work

An overview of current virtual machines for WSN is given by [20]. Most solutions are specialized on a particular application or abstraction. A prime example for such VM specialization is SwissQM [21] which provides a database query abstraction for an entire network. One of the first attempts of application-specific virtual machines is Maté [22] running on top of TinyOS. The Maté VM introduces tailored byte codes to compress the representation of a sensor application with the main goal of enabling dynamic uploads. Generic VM approaches are based on the J2ME standard and require significant hardware resources, like the Squawk VM [23] operating SunSPOTs using a 32 bit ARM. The Mote Runner multi-language VM focuses on energy-efficiency and targets more limited 8 and 16 bit platforms because larger hardware platforms inherently consume more energy [24]. Energy consumption is not the main focus of more recent proposals for Java VMs for motes such as Darjeeling[25] and TakaTuka[26].

Because power consumption is a major concern, simulation environments exist for estimating energy expenditure of applications before deployment. In this space, PowerSim is an extension which estimates power consumption based on logical models of the hardware in the PowerTOSSIM [27] simulation environment for TinyOS. In contrast to Mote Runner where the same application binary is executed by the same VM/OS code both in the simulation and on the hardware, the PowerTOSSIM simulation creates a special build of the application emulating a MicaZ mote. For the respective TinyOS binaries, the Avrora [28] system offers cycle-accurate simulations from which power estimations can be inferred.

Several systems have been developed to monitor and analyze power traces of real WSN deployments. These approaches are based either on instrumented operating system driver calls for tracing power consumption of logical activities [13] or hardware monitors[29]. Such instrumentation is concerned with determining the variances found in the real hardware and environment conditions and are fully complementary to our work. The advantage of such tools is in assessing real power traces for deployed applications without the overhead of complicated and expensive equipment like oscilloscopes and line analyzers.

VI. Summary and Conclusion

In this paper, we compared the power consumption of a real-world monitoring application implemented for two different WSN operating systems, Mote Runner and TinyOS, on the Iris mote. With Mote Runner, we show that cross-

layer VM/OS optimizations and power management reduce the expected energy consumption overhead of a VM-based system compared to a native implementation. One such optimization concerns the sleep strategy. During the active phase of the analyzed application, the Mote Runner VM overhead is clearly exposed. This penalty is below 4% when compared to the TinyOS implementation. However, the total energy consumed by the Mote Runner implementation is smaller than the respective TinyOS implementation for sleep intervals beyond 2.5 s, corresponding to duty cycles lower than 10%.

Based on our practical measurements, we conclude that the operating system can and should optimize and micro-manage power decisions on behalf of the application developer. To effectively use such optimizations an OS should provide an API so applications can communicate their intended actions and respective timings to the OS. The proposed abstractions and optimizations are applicable to other embedded operating systems and wireless sensor network platforms.

A VM always introduces some overhead. However, the combined advantages of a VM-based operating system, a coherent development and simulation environment can be traded for the additional power consumption. Such an abstraction enables more productive high-level programming and debugging and facilitates the maintenance of large scale WSN applications.

REFERENCES

[1] J. Beutel, S. Gruber, A. Hasler, R. Lim, A. Meier, C. Plessl, I. Talzi, L. Thiele, C. Tschudin, M. Woehrle, and M. Yuecel, "Permadaq: A scientific instrument for precision sensing and data recovery in environmental extremes," in *IPSN '09: Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*. IEEE, 2009, pp. 265–276.

[2] L. Bencini, F. Chiti, G. Collodi, D. D. Palma, R. Fantacci, A. Manes, and G. Manes, "Agricultural monitoring based on wireless sensor network technology: Real long life deployments for physiology and pathogens control," in *SENSORCOMM '09: Proceedings of the 2009 Third International Conference on Sensor Technologies and Applications*. IEEE, 2009, pp. 372–377.

[3] J. McCulloch, P. McCarthy, S. M. Guru, W. Peng, D. Hugo, and A. Terhorst, "Wireless sensor network deployment for water use efficiency in irrigation," in *REALWSN '08: Proceedings of the workshop on Real-world wireless sensor networks*. ACM, 2008, pp. 46–50.

[4] B. O'Flynn, R. Martinez-Catala, S. Harte, C. O'Mathuna, J. Cleary, C. Slater, F. Regan, D. Diamond, and H. Murphy, "Smartcoast: A wireless sensor network for water quality monitoring," in *LCN '07: Proceedings of the 32nd IEEE Conference on Local Computer Networks*. IEEE, 2007, pp. 815–816.

[5] M. Ceriotti, L. Mottola, G. Picco, A. Murphy, S. Guna, M. Corra, M. Pozzi, D. Zonta, and P. Zanon, "Monitoring heritage buildings with wireless sensor networks: The torre aquila deployment," in *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE, 2009, pp. 277–288.

[6] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, "Fidelity and yield in a volcano monitoring sensor network," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX, 2006, pp. 381–396.

[7] TinyOS, "TinyOS Community Forum," http://www.tinyos.net.

[8] P. Levis, L. N., M. Welsh, and D. Culler, "Tossim: accurate and scalable simulation of entire tinyos applications," in *Proceedings of the 1st international conference on Embedded networked sensor systems (SenSys)*. ACM, 2003, pp. 126–137.

[9] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis, "Collection tree protocol," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '09. ACM, 2009, pp. 1–14.

[10] A. Caracas, T. Kramp, M. Baentsch, M. Oestreicher, T. Eirich, and I. Romanov, "Mote Runner: A multi-language virtual machine for small embedded devices," in *SENSORCOMM '09: Proceedings of the 2009 Third International Conference on Sensor Technologies and Applications*. IEEE, 2009, pp. 117–125.

[11] IBM, "Mote Runner," http://www.zurich.ibm.com/moterunner.

[12] MEMSIC, "IRIS wireless measurement system," http://www.memsic.com/products/wireless-sensor-networks/wireless-modules.html.

[13] R. Fonseca, P. Dutta, P. Levis, and I. Stoica, "Quanto: Tracking energy in networked embedded systems," in *Proceedings of the Eighth USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2008, pp. 323–338.

[14] J. Polastre, R. Szewczyk, and D. Culler, "Telos: Enabling ultra-low power wireless research," in *Proceedings of the 4th International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE, 2005, pp. 364–369.

[15] Atmel, "Atmega640/1280/1281/2560/2561 preliminary," http://www.atmel.com/dyn/resources/prod_documents/doc2549.pdf, 2009.

[16] ——, "Low power 2.4 GHz radio transceiver for ZigBee™and IEEE 802.15.4™applications - AT86RF230," http://www.atmel.com/dyn/resources/prod_documents/doc5131.pdf, 2009.

[17] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors," in *Proc. of the 29th Conf. on Local Computer Networks*, 2004, pp. 455–462.

[18] S. Ganeriwal, R. Kumar, and M. B. Srivastava, "Timing-sync protocol for sensor networks," in *Proceedings of the 1st international conference on Embedded networked sensor systems*, ser. SenSys '03. ACM, 2003, pp. 138–149.

[19] M. Maróti, B. Kusy, G. Simon, and A. Lédeczi, "The flooding time synchronization protocol," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, ser. SenSys '04. ACM, 2004, pp. 39–49.

[20] N. Costa, A. Pereira, and C. Serodio, "Virtual machines applied to wsn's: The state-of-the-art and classification," in *ICSNC '07: Proceedings of the Second International Conference on Systems and Networks Communications*. IEEE, 2007, pp. 50–.

[21] R. Müller, G. Alonso, and D. Kossmann, "A virtual machine for sensor networks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. ACM, 2007, pp. 145–158.

[22] P. Levis and D. Culler, "Maté: A tiny virtual machine for sensor networks," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2002, pp. 85–95.

[23] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White, "Java™ on the bare metal of wireless sensor devices: the Squawk Java virtual machine," in *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*. ACM, 2006, pp. 78–88.

[24] A. Reinhardt and R. Steinmetz, "Exploiting platform heterogeneity in wireless sensor networks for cooperative data processing," in *Proceedings of the 8th GI/ITG KuVS Fachgespräch "Drahtlose Sensornetze"*, 2009.

[25] N. Brouwers, K. Langendoen, and P. Corke, "Darjeeling, a feature-rich vm for the resource poor," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '09. ACM, 2009, pp. 169–182.

[26] F. Aslam, L. Fennell, C. Schindelhauer, P. Thiemann, G. Ernst, E. Haussmann, S. Rührup, and Z. Uzmi, "Optimized java binary and virtual machine for tiny motes," in *Distributed Computing in Sensor Systems*, ser. LNCS, R. Rajaraman, T. Moscibroda, A. Dunkels, and A. Scaglione, Eds. Springer Berlin / Heidelberg, 2010, vol. 6131, pp. 15–30.

[27] V. Shnayder, M. Hempstead, B. Chen, G. Allen, and M. Welsh, "Simulating the power consumption of large-scale sensor network applications," in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*. ACM, 2004, pp. 188–200.

[28] B. L. Titzer, D. K. Lee, and J. Palsberg, "Avrora: scalable sensor network simulation with precise timing," in *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*. IEEE, 2005.

[29] J. Beutel, R. Lim, A. Meier, L. Thiele, C. Walser, M. Woehrle, and M. Yuecel, "The flocklab testbed architecture," in *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2009, pp. 415–416.