forward the data from other nodes and, therefore, decrease energy-intensive radio communication. Second, it could stop storing other nodes' data and avoid flash memory accesses. Finally, it could reduce energy consumption by decreasing the rate it queries its own position from the GPS receiver.

In these applications it is possible to identify parts which are more energy-intensive than others and which are not actually needed to provide some basic functionality. Therefore, in this paper we present *Levels*, a novel abstraction for energy-aware programming of sensor networks that allows the developer to explicitly single out optional functionality.

With our approach developers can specify several energy levels in an application which differ in their energy consumption and the functionality they offer. The code within each such level is associated with the energy it consumes. At runtime *Levels* monitors the remaining battery capacity and the energy consumed in each level. It then selects an energy level that allows the application to achieve its target lifetime, if necessary with restricted functionality. This way the lifetime of individual nodes can be significantly extended and, for example, network connectivity can be preserved. Compared to manual implementations of such functionality this programming abstraction and its corresponding runtime system save much development effort. For example, the application developer no longer has to write code to estimate the energy remaining in the battery, the energy consumed by parts of the application, or the time full functionality can be provided.

Our approach is based on measuring the energy consumption of individual energy levels using an energy profiler with accurate simulation models [14, 23]. At runtime each node tries to maximize the utility of the energy levels while achieving its lifetime goal. As we show in the evaluation, the abstraction of energy levels is useful in real-world applications and *Levels* is able to help meeting lifetime goals while providing good application quality.

Our solution has several benefits. First, the developer does not have to deal with the low-level issues of energy consumption, which simplifies the development of energy-aware applications. Second, our solution helps to ensure that a given lifetime is reached and that good application quality is offered. Third, the overhead for the developer is just small and we provide a powerful programming abstraction that allows for modular application development. Finally, the runtime overhead of our system is negligible.

The rest of this paper is organized as follows. Section 2 gives an overview of related work. Section 3 presents design considerations and our energy level abstraction. In Section 4 we then describe our method to measure energy consumption of code blocks and in Section 5 how *Levels* adjusts energy levels at runtime. Section 6 shows evaluation results and Section 7 discusses special application requirements. Finally, Section 8 gives an outlook on future work and concludes this paper.

2 Related Work

In this section we give a brief overview of work related to *Levels*. Particularly, we describe systems that take into account energy considerations for adaptation, extend network lifetime by deactivating redundant nodes, map energy consumption to code blocks, and model battery behavior.

In the realm of mobile computing, Odyssey [7] monitors available energy and adapts the fidelity of applications to meet a user-defined lifetime goal. For example, a video player switches to a differently compressed source file or reduces its window size if energy becomes scarce. Odyssey does not provide a programming abstraction like our energy levels and does not leverage simulation data. Therefore, it cannot take advantage of predicting energy consumption after adaptation. Furthermore, it has been designed for less resource-constrained devices and relies on highly accurate measurement equipment, which we cannot assume on inexpensive sensor nodes.

Similarly, ECOSystem [27] tries to achieve a target lifetime by limiting the discharge rate of the battery. It introduces the Currentcy Model to deal with the demands of competing tasks in a multitasking system. Rather than identifying optional functionality in applications, it modifies the scheduler to execute only those tasks that have not spent their energy budget for the current round yet. Unlike our approach it does not exploit information from simulation and, therefore, has to do detailed energy accounting at runtime.

In the field of sensor networks we have been working on TinyCubus [17], a framework that adapts applications to the properties of the environment and to application requirements. TinyCubus takes into account other parameters like reliability that we do not consider with *Levels*. In contrast to *Levels*, TinyCubus does not provide any mechanisms to reach given lifetime goals yet. However, we plan to integrate *Levels* into TinyCubus to provide this functionality.

TinyDB [16], a query processor for sensor networks, allows to adapt the interval between the measurements of a query in order to meet user-defined lifetime goals. Similar to our rationale, its authors argue that in environmental monitoring scientists are more concerned about meeting a lifetime goal than about the sampling rate. Since TinyDB's programming interface is based on high-level SQL-like queries, changing the sampling rate is the only way to influence network lifetime.

There is already a large body of work dealing with the coverage problem in wireless sensor networks. This work switches redundant nodes into sleep mode to maximize the time that a given area is monitored by the network [2, 11]. Closely related are topology control mechanisms like ASCENT [3] that switch off redundant nodes but strive to preserve network connectivity. Similarly, duty-cycling approaches [9] periodically turn off nodes to extend network lifetime. Unlike *Levels* all of these approaches are targeted to dense networks where redundant nodes can be temporarily deactivated. In addition, they do not have any given lifetime goals but try to maximize the time for which they provide coverage or connectivity, respectively.

Many network protocols already include mechanisms to reduce energy consumption. For example, at the link layer several protocols try to reduce the energy spent for idle listening. Therefore, they often switch the radio chip into its sleep state [19, 25]. These optimizations are orthogonal to our approach. Energy levels could still be used on other lay-

ers, as long as, of course, the same protocol stack is used for both energy profiling and the application itself.

Sensor network simulators like Avrora [14, 23] and PowerTOSSIM [22] enable the prediction of the energy consumption of a sensor node. The values obtained from these tools are often used for evaluation purposes and to give the developer hints about energy consumption, although usually not at runtime. Avrora allows to break down energy consumption to individual functions. However, this part of Avrora can only associate the energy consumption of the CPU with some code rather than including the other hardware components on a node. In addition, unlike our approach, it does not take into account the energy consumed by functions that are called by the code under measurement or, in the case of TinyOS [10], by asynchronously executed tasks.

There are several more advanced battery models than ours described in the literature [21]. They take into account effects resulting from temperature changes and time-varying loads, for example. However, because the voltage sensor on typical sensor nodes cannot provide the precision of lab equipment, we have to deal with inaccuracies anyway. Furthermore, the computational overhead of many accurate battery models is too large for resource-constrained sensor nodes, and it takes even for more powerful computers hours to simulate a load profile.

3 System Design

In this section we present relevant properties of sensor networks that influenced our design decisions, state our design goals, and give an overview of our system and of the energy levels abstraction.

3.1 Sensor Network Properties

Several properties of wireless sensor networks aid our approach of measuring energy consumption and switching between energy levels at runtime.

First of all, there is usually just a single application running on each sensor node. Therefore, the expected lifetime of a node depends only on one application that can be controlled more easily than a multitasking system.

Second, sensor networks typically exhibit some periodic behavior. For example, sensor readings are sampled periodically at user-defined time intervals. If the sensor network application reacts to external events, these events often repeat for sufficiently large periods. Thus it is possible to estimate future energy usage based on past consumption.

Third, because sensor nodes have only limited output capabilities and are often deployed in inaccessible locations, simulation has become an integral part of the development process [23]. In addition, simulators are often equipped with detailed energy models [14, 22]. Therefore, we can use simulation to get information about the energy consumption of a piece of software.

Fourth, most sensor nodes available today are equipped with voltage sensors. Since the voltage provided by a battery depends on its remaining capacity, we can use the voltage data to estimate the residual energy.

Finally, as the nodes are strictly energy-constrained, in the domain of sensor networks software developers are more concerned about energy consumption and node lifetime than developers in other areas. Therefore, we expect that most developers are willing to invest some effort for specifying energy levels and measuring energy consumption.

3.2 Design Goals

Our central design goal is to *provide a programming abstraction and runtime support that helps to meet the user's lifetime goals by deactivating parts of an application if necessary*. To achieve this main objective we have identified the following subgoals:

- The programming abstraction should allow for the definition of optional functionality and it should be general enough to support a wide range of applications.
- *Levels* should be easy to use and the development overhead should be limited.
- For a given lifetime goal *Levels* should provide good application quality, i.e., nodes should not live much longer than required.
- The system should have a low runtime overhead to avoid that the overhead absorbs its benefits.
- The runtime system should be able to deal with inaccurate energy estimations which are inevitable with inexpensive sensor nodes.

3.3 System Overview

Based on these design goals we have created the programming abstraction of "energy levels" that allows to specify optional code blocks. At runtime the system then decides which energy levels are active, i.e., which code blocks should be executed. This abstraction is described in more detail in Section 3.4.

Basically, our system is similar to well-known model predictive control (MPC) schemes [1]: First, we build a model that helps to predict energy consumption by profiling the energy consumed by optional code (see Section 4). This model allows to compute the energy used by each level at runtime. Second, it is complemented by a battery model that maps voltage readings to the remaining energy usable by the sensor node (see Section 5.1). Third, using the information from energy profiling *Levels* keeps track of how much energy is consumed by each energy level at runtime (see Section 5.2). This part of *Levels* considers both energy that is consumed just once when executing a code block (e.g., to store some data in flash memory) and changes in the rate of continuously consumed energy (e.g., by enabling a sensor). Finally, together with the battery model this data allows to compute the expected node lifetime in each energy level. This information is then used to optimize the energy level for the remaining lifetime considering the given energy constraints (see Section 5.3). Like other MPC algorithms, our system considers just the result for the current time interval and later recomputes the remaining level assignments to better reflect the new situation.

3.4 Energy Levels

An energy level includes all statements that can be deactivated together to reduce energy consumption. Therefore,

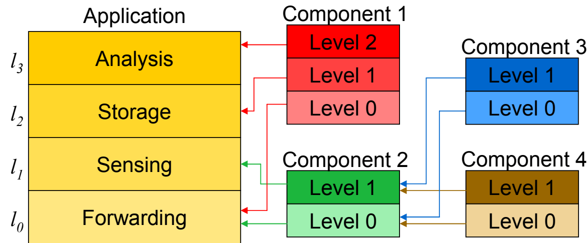

Figure 1. Combining energy levels

code in an energy level is optional for providing some basic functionality of an application. If the level is deactivated, however, the functionality of the application may be degraded. To put a code block into an energy level the developer has to place it into a conditional statement that checks if the level is currently active. The lowest energy level l_0 is always active and is declared implicitly; it includes all code that has not been added to any other level.

Several energy levels form a stack where levels can be deactivated starting from the top one. If level l_i is active, all levels below it, i.e., $l_0, \ldots, l_{i-1}$ are active, too. Therefore, the code in l_i can rely on the functionality of lower levels. Levels above l_i, however, might be deactivated.

Each energy level l_i is associated with a utility value u_i: The application developer can define this utility in a way that reflects the improvement in functionality.

Having a stack of energy levels does not mean that the functionality of an application has to increase monotonically with higher levels. By using appropriate conditions it is possible to, for example, transmit sensor readings in a low energy level to the base station whereas they are just stored locally (without being forwarded) in a higher level.

Levels assumes that higher levels lead to an increase in energy consumption. We expect this to be true for almost any application. Otherwise, energy levels should be merged because they are ill-defined. Such a situation could be easily detected during the development phase.

If an energy level is being activated or deactivated, the runtime system calls a special function in the component providing the level. The component can use this function to adjust to the new level. For example, if all the sensor sampling code is extracted into an energy level, the component could turn the sensor hardware on or off in these functions.

The abstraction of energy levels nicely fits modular development in component-oriented languages like nesC [8]. If an application consists of several software components which define their own energy levels, it might be undesirable that each of them can be deactivated separately. For example, code in one component might depend on functionality of another one's (higher) levels. Therefore, a component can combine levels of several components and thus ensure that they are only active at the same time. For example, in Fig. 1 the energy levels of Components 3 and 4 are combined using this mechanism. Likewise, the overall levels of the complete application can be created by combining the energy levels of its components. In addition, it is possible to insert levels from one component between those of another one. For instance, in Fig. 1 level 1 of Component 2 is mapped between

```
1  module LevelTestM {
2    provides energylevel SendLevel<1>;
3    provides energylevel ComputeLevel<2>;
4    ...
5  }
6  implementation {
7    ...
8    event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr msg){
9      if (ComputeLevel.active) {
10       post computeTask();
11     }
12     return msg;
13   }
14   event result_t Timer.fired() {
15     if (SendLevel.active) {
16       call SendMsg.send(TOS_BCAST_ADDR, 29, &message);
17     }
18     return SUCCESS;
19   }
20   command void SendLevel.activate() {
21     call RadioControl.start();
22   }
23   command void SendLevel.deactivate() {
24     call RadioControl.stop();
25   }
26   ...
```

Figure 2. Code example for energy levels

the levels of Component 1 in the application. However, it is not possible to change the order of the levels of a single component; doing so could break assumptions in the code.

Using this simple mechanism, which closely corresponds to nesC's wiring of interfaces, the overall energy levels of the application in the figure ($l_0, \ldots, l_3$) are formed. This application can deactivate functionality for data analysis, storage, and sensing if necessary. Forwarding functionality, however, is placed on the lowest level l_0, which is always active.

By connecting all required levels to level l_0 the developer has full control of which energy levels have to be active for the current application. All other levels, however, can be deactivated if necessary. Therefore, *Levels* allows for the reuse of components with several energy levels, even if all of them are required for one specific application.

3.4.1 Syntax

We integrated *Levels* into nesC [8], the programming language used by TinyOS [10]. We selected nesC because of the large number of sensor network applications that has already been developed with this programming language. In addition, building upon this general-purpose language helps to achieve general applicability of our abstraction.

Fig. 2 shows a small code example of a component that provides two energy levels. The numbers in the declaration of energy levels determine their local order. However, these numbers do not have to be absolute or globally unique; other levels can still be inserted when wiring the component.

In this example each energy level consists of a single optional code block. If "ComputeLevel", the highest level, is active, the component performs some computation after receiving messages (line 10) and periodically sends some packets itself (line 16). If just "SendLevel" is active, the component will continue sending messages but cease to do the computation. It would also have been possible to add else-branches here that run a less expensive computation task in lower levels, for example. In the implicitly declared de-

```
1  module ProfilingM {
2    provides interface ReceiveMsg;
3    provides interface Timer;
4    ...
5  }
6  implementation {
7    ...
8    void measureReceive()
9        @energy("ReceiveMsg", "receive") {
10     TOS_Msg msg;
11     msg.addr = TOS_LOCAL_ADDRESS;
12     ...
13     signal ReceiveMsg.receive(&msg);
14   }
15   void measureTimer() @energy("Timer", "fired") {
16     signal Timer.fired();
17   }
18   ...
```

Figure 3. Test driver used for energy profiling

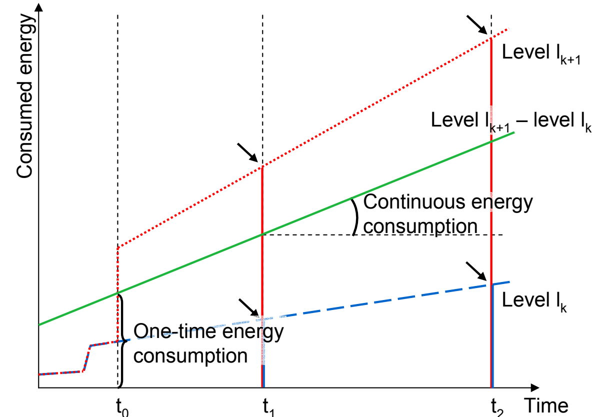

Figure 4. Computing the energy consumption of a code block

fault level l_0, neither "computeTask" nor "send" will be invoked. Note that the code inside these two functions is regarded as a part of the energy level from which it is called.

Whenever a level is being activated or deactivated, one of the corresponding functions will be called. In the example the radio chip is turned on just as long as it is needed (lines 21 and 24).

As this example shows, *Levels* requires only very small changes to existing nesC modules. Furthermore, wiring energy levels (not shown in the figure) is completely analogous to wiring interfaces in nesC. Therefore, we think that using energy levels should be easy for application developers.

4 Energy Profiling

In order to correctly estimate the lifetime of the application, *Levels* has to know how much energy is consumed by each optional code block defined in energy levels. Getting this information on the sensor node itself is not possible since the energy consumed in individual blocks of code is too small to be accurately estimated using the node's built-in voltage sensor. Therefore, we make use of the fine-grained energy models available in simulators.

It should be noted that because of hardware differences the energy consumption of different nodes varies slightly [14]. Currently, profiling can achieve only optimal results if the energy model of the simulator is calibrated to each node. Therefore, from our perspective an important design requirement for future sensor nodes is that they should be created from parts which show only little variations in energy consumption.

Compared to real measurements with instrumented sensor nodes and lab equipment, simulation has the advantage that the additional effort for the application developer is very small. Furthermore, our approach allows to reuse code from unit testing to profile energy consumption.

4.1 Measurement Approach

Unit testing using either tools like JUnit [12] or custom test drivers has already proved to be a valuable technique in different domains including sensor networks. For example, the TinyOS distribution includes several small test applications for many operating system components. In addition, with nCUnit we have developed a JUnit-like testing tool for nesC-based applications. Since nCUnit executes its test suites in a simulator, its approach corresponds to our energy profiling technique. Therefore, the developer can reuse the test code from nCUnit for energy profiling. The only change needed is to tag all relevant functions in the test driver with an "@energy" attribute that tells our build system which functions should be used to measure energy consumption.

Fig. 3 shows example code that can be used to measure the energy consumption of the optional code blocks in Fig. 2. This module has to be wired to the component whose energy consumption should be measured instead of the components normally used to, for example, receive radio messages. Creating these simple functions and wiring the component correctly is the only additional effort needed from the developer. As already mentioned, similar or even the same functions can be used for unit testing.

A pre-compiler generates calls for all measurement functions tagged with the "@energy" attribute. The parameters of this attribute specify the name of the function that should be profiled. For each measurement function the energy profiler is executed several times, where – in order to avoid side-effects – each simulation calls only a single measurement function once. The profiler starts two separate simulation runs for all energy levels and each of their optional code blocks: one with a short duration t_1 and one with a longer duration t_2.

These energy measurements allow the system to compute two kinds of energy consumption for each of these code blocks: energy that is consumed once (i.e., when the code is executed) and energy that is consumed continuously (i.e., by changing the state of a hardware device). For example, sending a message requires only energy once whereas turning on sensors leads to a change in continuous energy consumption. In addition, the measurements allow to remove the overhead introduced for setting up the test case.

Fig. 4 shows how this computation is done for the case when the function under measurement defines just a single optional code block. The function is called at the well-known point of time t_0. To get the energy consumption of level l_{k+1} four measurements are necessary: the energy consumptions of the function under measurement in levels l_k and l_{k+1} at

both t_1 and t_2 (see the arrows in the figure). Then the difference between the two levels is computed by subtracting their values. From the resulting points the slope of the energy difference, which corresponds to the change in continuous energy consumption, and the one-time energy overhead at t_0 can be computed.

This computation assumes that continuous energy consumption is linear. We expect this to be true on average for sufficiently long simulation durations. For example, if a timer is activated to periodically execute some code or if the sensor board is turned on, the average energy consumed will increase linearly with time.

Such a computation is done for each optional code block of all energy levels. If there are several such blocks in the function under measurement, in each run an additional block (from the beginning of a function to its end) is activated. Using the same principle as outlined above now for each code block, the difference in the energy consumed can be computed. Activating code blocks incrementally allows to measure their individual energy consumption while still ensuring that they can rely on the code in preceding blocks to be active, too. At runtime, of course, in the actual application all blocks of an energy level are active at the same time.

After executing all simulations, the energy profiler analyzes its log files and performs the computation outlined above. It then stores the energy consumption of each code block in a central file. This file is later read when compiling the actual application to insert energy values into the code.

Our profiling approach has several advantages. First, reusing unit testing code ensures that the code is executed in a controlled setting where, for example, messages from other nodes, unexpected sensor readings, or interactions with other components do not alter the application flow. Second, these measurements do not only include the energy spent by the CPU to run the code under test but also the energy consumption of other hardware like the radio or flash memory chips. Finally, unlike existing energy profilers [14], which map energy consumption to code blocks, or systems monitoring the energy state of hardware components at runtime [5] our approach allows to include the energy consumption of asynchronously executed code (e.g., TinyOS tasks, timers, split-phase events) in the measurement. This is important to get the total energy consumption originating from the code block: Since this code is executed from a certain energy level, its energy consumption has to be attributed to that level.

4.2 Special Cases

There are two special cases to consider: the energy consumed by the lowest level l_0 and energy consumption that depends on some state of the hardware or software.

First, we do not measure the energy consumed by the default energy level l_0 and rather compute this value at runtime by subtracting the energy of all other levels from the overall energy consumed. This decision helps to keep the runtime overhead of *Levels* small since this level would be present in every single function. Furthermore, because there are no optional code blocks in level l_0, profiling could be done only at a coarse granularity and, therefore, would be probably inaccurate.

Second, for some code the energy consumption can differ depending on the state of the hardware and the application. For example, if – within an optional code block – the application tries to turn on a hardware device that is already enabled, executing this code will not change energy consumption. To address this issue the application developer can provide a condition in the "@energy" attribute that will later be checked at runtime. Therefore, each measurement function can refer to a different state of the component. For example, the condition could check an already existing state machine or read out the status of a hardware device. The system stores separate information about energy consumption for each condition. Depending on which condition applies, *Levels* attributes the correct energy consumption to the code block at runtime. However, the developer should choose these conditions in a way that allows to evaluate them efficiently. Otherwise, the overhead for checking the condition at runtime could outweigh the benefits.

If a level is not active at runtime, evaluating such conditions can be difficult since the deactivated code might have some effects on them. For example, if a function adds sensor readings to a buffer before storing all of them together to flash memory, only one of several calls will result in an energy-expensive write access to the flash chip. To deal with this problem our energy profiler calculates the probability that a code block is called with one of the conditions defined and computes the average continuous energy consumption of each level. For this purpose we have to run the complete application in a realistic scenario rather than execute unit test code. This can be done while testing the whole application. In these measurements we rely on the previously profiled energy consumption of code blocks; we just count how often they are called for each of the conditions given by the application developer. Since the information obtained from simulating the complete application is less accurate than the atomic energy measurements of code blocks, we use it only when necessary, i.e., for code blocks of deactivated energy levels.

5 Runtime System

In this section we describe *Levels*' runtime system. The runtime system has three tasks: estimating the remaining energy from voltage readings, attributing energy consumption to energy level, and adjusting the active levels at runtime.

5.1 Battery Model

To estimate the remaining lifetime it is necessary to know at runtime how much energy is left in the battery. For this purpose we have built a simple battery model that maps voltage values to the remaining usable battery capacity.

Creating such a model has not been the main focus of our research. In fact, if the sensor nodes were equipped with battery monitoring chips that accumulate the current drawn from the battery, better accuracy could be obtained than with this model. However, since the small, low-power devices that we target have only voltage sensors, battery voltage has to be mapped to the remaining energy. To do this mapping we created a battery model for the specific type of alkaline-manganese dioxide batteries [6] that we use in our experiments.

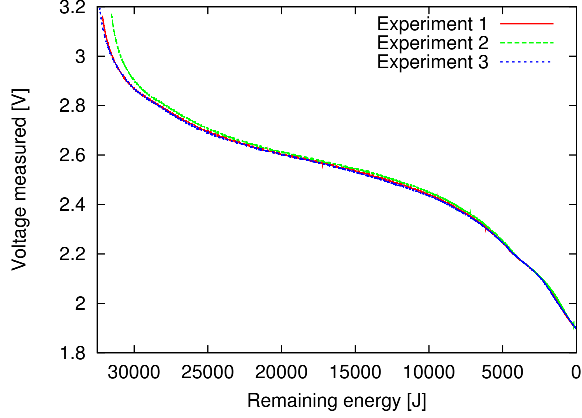

Figure 5. Battery discharge characteristics from three experiments

We opted for a simple but efficient model: for each distinct voltage reading we store the average remaining energy of this value in program memory. The overhead of this approach is minimal since it does not require any computation at runtime. Nevertheless, *Levels* is flexible enough to be combined with more advanced (but possibly more complex) battery models.

Because typical sensor network platforms like Mica2 nodes are not equipped with a voltage boost converter [20], the current draw I depends linearly on the battery voltage U. Thus the resistance R remains constant. From $E = U \cdot I \cdot t$ and $R = \frac{U}{I}$ the effect on energy (and power) is quadratic: $E = \frac{U^2}{R} \cdot t$. However, when creating our battery model we assumed a constant voltage $U_{const} = 3V$ for the computations. Therefore, instead of mapping the actual energy E to the voltage readings, our battery model and all energy values in the rest of this paper refer to values for $E \cdot \frac{U_{const}^2}{U^2} = \frac{U_{const}^2}{R} \cdot t$. This simplifies computations at runtime greatly because the energy consumption of a code block can be assumed to be independent of the current supply voltage of the sensor node. Nevertheless, using the same approach in the creation of the model and at runtime leads to consistent results that allow for accurate computations.

To create the battery model we built an application for Mica2 nodes that periodically measures the voltage and transmits this data via radio until the batteries are drained. Using the energy model of the Avrora simulator [14] we later computed the total energy consumed by this measurement application throughout the node's lifetime. The result of this computation is not necessarily the total energy available in the battery but rather the energy that is actually usable by the sensor node. For our purposes this number is more relevant because this is also the energy available at runtime. Ignoring for simplicity effects like the influence of temperature on the batteries, this data allows to relate voltage measurements with the usable energy left.

We performed several experiments with this application and created the model used at runtime by computing the average voltage reading for each energy value. Fig. 5 shows the discharge behavior of three batteries. Although there are some differences, the curves are almost equal when the batteries are almost empty. Particularly there a good energy estimation is important to accurately meet a lifetime goal.

Since the relationship between voltage and the remaining energy is not linear, the differences in energy values between two consecutive voltage readings can vary significantly (see Fig. 5). This directly affects the accuracy of the mapping. For example, in our battery models the differences vary between 7 J and 412 J for a battery with about 32,000 J usable capacity. Similar differences can exist between the models of several batteries, especially close to their total capacity. Therefore, we make this expected error available at runtime. This makes it possible to defer computations until significantly more energy than this error estimate has been consumed. Hence the influence of the inaccuracies is reduced.

5.2 Attributing Energy Consumption to Energy Levels

The runtime system is responsible for attributing energy consumption to energy levels. First, it is called whenever an optional code block is about to be executed. It then checks if the energy level is active and adds the energy consumption of this code block to the total energy consumed by the corresponding level. Second, periodically (every few seconds) it adds up the energy that has been consumed continuously in the current interval. Finally, periodically (every few hours) it uses this information and computes the optimal level assignment for the time remaining.

If an optional code block of an energy level is about to be executed, the system checks if the level is active. Only if it is active, the code will be executed. Furthermore, the runtime system uses the data obtained with energy profiling (see Section 4) and adds the energy consumption of the current block to the overall energy consumed by the level. If a code block of level l_i is reached by executing code belonging to level l_j with $j > i$, the system correctly attributes the energy consumed by this code to level l_j. In addition, it updates continuous energy consumption if it is changed by the code.

The same information is also updated for blocks belonging to the next higher energy level in the stack, which is actually not executed. This way the system can predict the energy consumption after increasing the current level. However, we do not monitor the energy consumption of even higher levels because it is unclear which of their code will be additionally reached if the levels in between are activated. For example, if currently only level l_i is active, the system cannot tell whether or not the application will reach more code blocks of level l_{i+2} from the code in level l_{i+1}. Therefore, energy consumption of l_{i+2} would not be accurately predicted.

Keeping only information up to the next higher level also restricts the levels that can be selected when adjusting the current level. Therefore, in each adjustment the system can increase the current level by at most one. This problem does not occur with lower levels because their energy consumption can always be accurately predicted. Thus several levels can be skipped when switching to a lower level.

As already mentioned, for each energy level *Levels* keeps information about its continuously consumed energy, e.g., for a hardware component that has been enabled in an en-

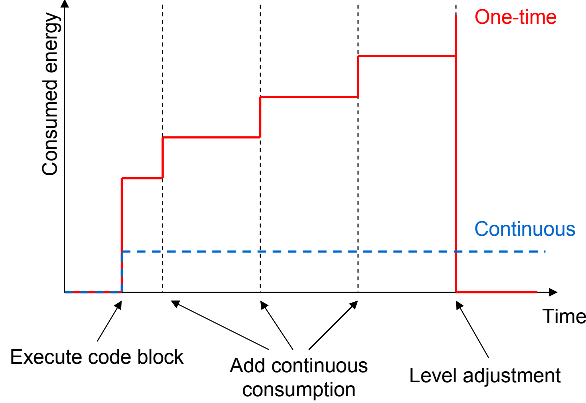

Figure 6. Accumulating the energy consumed by a level

ergy level. The runtime system periodically adds the energy continuously consumed in the last few seconds to the one-time energy consumption of the code. This approach provides finer granularity and, therefore, better accuracy than doing this only when computing the energy level assignment. In addition, it minimizes overhead because it requires less state and computational resources compared to calculating this data whenever continuous energy consumption changes.

Fig. 6 summarizes how the runtime system computes the energy consumed by an optional code block. When the code block is executed, both one-time and continuous energy consumption are updated. The system then periodically adds continuous energy consumption to the energy consumed by the level. After some time this energy consumption is reset when computing a new level assignment.

5.3 Adjustment of Active Energy Levels

Levels uses the information about the energy consumption of energy levels to periodically adjust the currently active level. In each adjustment it tries to maximize the utility of the energy levels for the time remaining while meeting the lifetime goals. Formally this corresponds to the following optimization problem. Given the current lifetime t, the total required lifetime T_{req}, the remaining energy E_{rem}, and the energy levels $l_0,\ldots,l_{n-1}$, which have the utility values $u_0,\ldots,u_{n-1}$ and consume $P_0,\ldots,P_{n-1}$ energy units per time interval, find the durations $t_0,\ldots,t_{n-1}$ that maximize the utility of the energy levels for the remaining lifetime $T_{req}-t$:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=0}^{n-1} u_i \cdot t_i \\ \text{subject to} \quad & \sum_{i=0}^{n-1} t_i = T_{req} - t \\ & \sum_{i=0}^{n-1} P_i \cdot t_i \leq E_{rem} \\ & t_0,\ldots,t_{n-1} \geq 0 \end{aligned}$$

The first equation formalizes the maximization of the utility over time. The constraints then specify that the still needed lifetime has to be met and that enough energy has to be available. Using a linear equation for the energy constraint is only possible because our battery model returns the energy for a (hypothetic) constant voltage instead of actual energy values (see Section 5.1). Finally, the last equation excludes solutions with negative time durations.

The optimization problem can be solved using well-known algorithms from linear programming [4]. In our implementation we use the Simplex algorithm, which is the standard method to solve such problems. Since our implementation uses efficient fixed point arithmetic, the computational overhead of this algorithm is small. Furthermore, we limited the overhead by defining a maximum number of iterations after which the algorithm aborts even if it has not found the optimal solution yet. In practice, however, this limit is reached only seldom. As we show in the evaluation, the computational overhead can almost be neglected even on resource-constrained sensor nodes (see Section 6.3).

The system tries to compute a new level assignment periodically with a low frequency (e.g., every two hours). Repeating this computation is necessary since energy load may vary over time and because of possible inaccuracies in previous adjustments. However, due to the discharge characteristics of batteries (see Section 5.1) the inaccuracies of the measurement might exceed the actual energy consumed, especially for low-power applications. In this case it is not possible to compute meaningful results. Therefore, we use the expected accuracy from the battery model at runtime: only if the energy consumed by code in energy levels is sufficiently greater, the algorithm tries to compute a solution. Otherwise, it waits until the next measurement. Although this reduces the agility of the system, it helps to obtain correct results. In addition, to further reduce the fluctuations because of inaccurate measurements, we use a moving average to smooth the energy values used for computation.

Furthermore, to deal with inaccurately estimated remaining energy and with possibly varying load within an energy level, *Levels* adds a safety factor to the lifetime still required in order to make sure that the node can meet its lifetime goal. This design decision leads to the side effect that the average level achieved is slightly below the optimum because the lifetime goal is usually exceeded. We opted for this conservative policy to ensure that no node runs out of energy early. Furthermore, as the safety factor depends on the remaining lifetime required, this issue is addressed by periodically recomputing the level assignment. The node will – in later computation rounds – switch back to higher levels if sufficient energy is still available.

To minimize state, our system does not change levels between these computations, even though the Simplex algorithm returns a complete level assignment for the remaining time; we just switch to the highest level of the result for optimal application quality. However, depending on the energy consumption of the application and the current accuracy of the battery model, several tries might be needed until the level assignment can be recomputed. Therefore, a level is selected only if the algorithm expects to be executed again before the computed time duration. Otherwise, *Levels* already switches to the next lower level of the solution.

Table 1. Average lifetimes of sample applications for constant energy levels (in days)

Application	Level l_0	Level l_1	Level l_2
FFT	961	375	not used
Flash	961	296	not used
SendLPL	34.6	22.5	16.7
SendLPLRandom	34.6	27.1	22.2
SendRadioOff	948	692	not used
Voltage	7.69	6.65	5.93

6 Evaluation

This section evaluates the benefits and overhead of *Levels*. For this purpose we use both simple applications, which correspond to components found in more complex ones, and real-world applications.

Unless otherwise mentioned, we use the Avrora simulator [23], which accurately emulates Mica2 sensor nodes. The battery voltage that the simulator makes available to the sensor nodes' voltage sensors has been recorded from individual batteries with the voltage sensor of a real sensor node. In contrast, *Levels* running on the sensor node uses a battery model based on the average of several such voltage traces (see Section 5.1). Therefore, this simulation setup corresponds to the situation of real sensor nodes.

6.1 Quality of Level Assignments

In this subsection we evaluate the quality of level assignments. To do this we use several metrics: First, we contrast the actual to the required lifetime. Second, we compare the average utility achieved in simulation with the optimal value possible. Finally, we validate our simulation results with experiments using physical sensor nodes.

6.1.1 Simulation of Small Applications

The small applications that we use for this evaluation represent parts which can also be found in larger ones. FFT periodically computes a Fast Fourier Transform, Flash stores data into flash memory, and SendRadioOff turns the radio chip only on for sending (short) messages but does not listen for any messages itself. Unlike SendRadioOff, SendLPL uses low-power listening [19] and thus sends messages with longer preambles. In addition, it includes another energy level where a second periodic message is sent. SendLPLRandom is similar to that. However, instead of periodically running this code it waits for a random time and then sends a random-length burst of messages. Finally, Voltage periodically sends messages with its current voltage reading and, on the highest energy level, toggles its LEDs every thirty seconds. We used this application also for experiments with real nodes. Because of the time constraints of our experiments we intended to create a particularly energy-intensive application here.

Except for SendLPLRandom all of these applications execute some tasks periodically. This code has been encapsulated in an energy level which can be deactivated if necessary. In this case, however, the applications will no longer perform their actual tasks. SendLPLRandom, in contrast, represents an event-based application. To simulate events, it waits for a random time (up to one hour) before running its optional code.

In most applications the utility values have been set to 0 for Level l_0, 1 for Level l_1, and 2 for Level l_2. For SendLPL and SendLPLRandom, however, the utility of level l_1 and l_2 has been increased slightly to 2 and 3, respectively. Because of the greater gap to l_0 and the small difference to l_2 these two applications preferably switch to l_1 if they cannot stay in l_2 all the time.

Table 1 shows the simulated lifetime of the applications when a constant energy level has been set. We validated some of the shorter lifetimes with real sensor nodes. The results of these experiments differed by at most 2.2 % from the simulated values. We attribute these differences mostly to variations in the capacity of the batteries used and slight deviations in the energy model of the simulator.

The table shows that our evaluation includes very low-power applications with a maximum lifetime of several years like FFT, Flash, and SendRadioOff as well as applications like SendLPL, SendLPLRandom and Voltage which have a high energy consumption. In addition, the lifetime of the nodes varies significantly for different energy levels. Depending on the application, the lifetime can be extended by between 30 % and 225 % if only the lowest level is active.

Levels will not be able to meet a lifetime goal if the lifetime requested cannot be possibly achieved when the application does not already start in the lowest level. For example, for one of our simulated batteries, SendLPL has a maximum lifetime of 50,318 minutes in level l_0. If a lifetime of 50,000 minutes is requested for this application and if the initial energy level is l_2, *Levels* switches to level l_0 as soon as possible. Nevertheless, it can only achieve a total lifetime of 49,348 minutes and, therefore, fails to meet the requested lifetime goal. However, cases like that are somewhat artificial since using *Levels* does not make much sense if the lifetime goal can hardly be met in the lowest level. Therefore, in this evaluation we focus on more realistic lifetime goals where better application quality is actually possible.

We have set such goals for all sample applications and selected the lifetimes such that the first run could be completed in the maximum level with most simulated batteries. Fig. 7 compares the average lifetime when simulating different batteries (including 95 % confidence intervals) with the lifetime requested. In each of these simulations *Levels* was able to meet the lifetime goal. In addition, the size of the confidence intervals is in almost all cases smaller than 4 % of the total lifetime. This is less than the confidence intervals of the simulated battery capacity, whose size is about 6.4 %, since *Levels* adjusts to the actual battery discharge curve.

Furthermore, the nodes did not live much longer than requested. Therefore, the applications were able to provide almost optimal quality subject to the constraints present. The largest differences relative to the lifetime achieved can be observed most often for the simulations with the smallest lifetimes because they have been chosen in a way that the applications can stay in their highest levels for the complete lifetime. Therefore, the node cannot possibly consume all the energy available. For example, in the first simulation run, SendLPL lives about 20 % longer than requested although *Levels* stays in Level l_2 for almost the complete simulation.

Besides that, the relative differences between the required

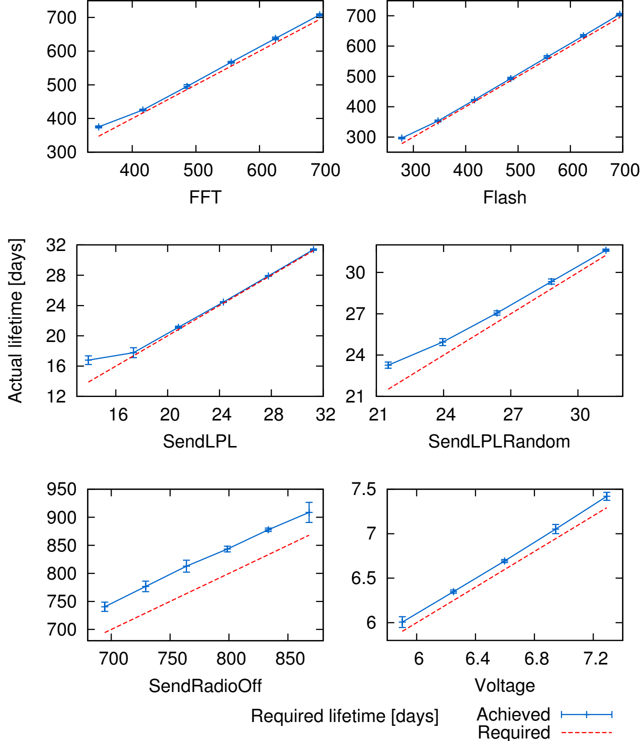

Figure 7. Required lifetime vs. lifetime achieved (including 95 % confidence intervals)

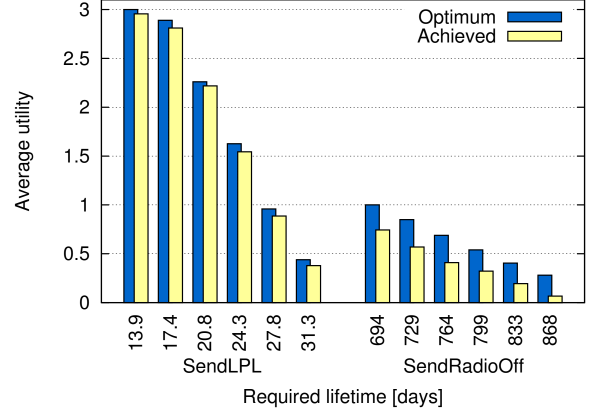

Figure 8. Average utility throughout the required lifetime

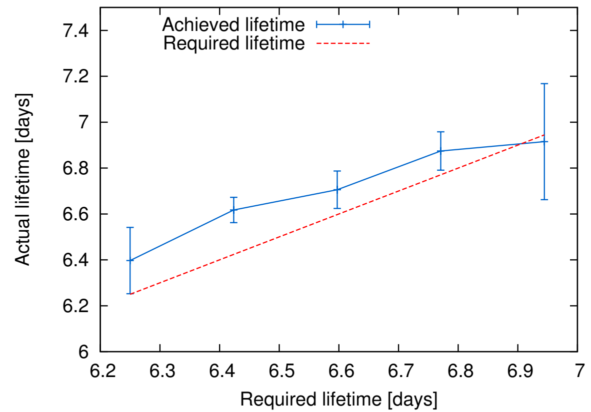

Figure 9. Lifetime of the experiments with Mica2 nodes (including 95 % confidence intervals)

and the actual lifetime are the largest ones for SendRadioOff (5.9 % on average). Due to the inaccuracies in the estimation of the battery's remaining energy, *Levels* defers the computation if only a very small amount of energy has been consumed in optional energy levels. Therefore, *Levels* can only execute a small number of level computations and cannot switch long enough back to a higher level when the energy reserved as a safety overhead becomes available near the end of the required lifetime. Since this application consumes in level l_1 just 0.14 mW more than in level l_0, it can take more than 45 days until a new level assignment is computed, which reduces the agility of level adjustments. This delay could only be reduced significantly with more accurate hardware to measure battery capacity.

SendRadioOff is an extreme example for the amount of energy consumed where meaningful computations are almost impossible. As other long-lived applications show, a small increase in power consumption is enough to obtain considerably better results. For example, FFT, for which the actual lifetime is much closer to the required one, consumes in level l_1 just 0.60 mW more than in level l_0.

This problem of SendRadioOff is also shown in Fig. 8. This figure compares for a single battery the average utility value achieved within the requested lifetime to the optimal average. This optimum has been computed offline by solving the same linear programming problem as *Levels*. However, it has been calculated only once using exact information about the battery capacity and the energy consumption of the different levels.

The figure shows that SendLPL achieves the optimal utility value almost perfectly despite the inaccuracies present on the sensor nodes. For SendRadioOff the difference is greater (about 0.24 utility units) because of the small number of level computations described above.

6.1.2 *Experiments with Mica2 Nodes*

We validated the simulation results of the Voltage application in experiments with real hardware using Mica2 sensor nodes. To prevent side-effects from slight variations in energy consumption we calibrated the energy model used for profiling with a multimeter to the specific sensor nodes used and created a separate battery model for each node.

Fig. 9 shows the actual lifetime achieved by the motes when varying the required lifetime. We define as the lifetime the time a neighboring node was able to receive periodically transmitted packets, which were sent irrespective of the current energy level. In most experiments the nodes met their lifetime goal. However, in the last experiments we ran – some of those with a lifetime of 10,000 minutes (6.94 days) – most nodes failed early although in previous experiments they accurately achieved this lifetime. For this time value these failures reduce the average lifetime and increase the size of the confidence interval. We had purchased the batteries used in these (failed) experiments several months after

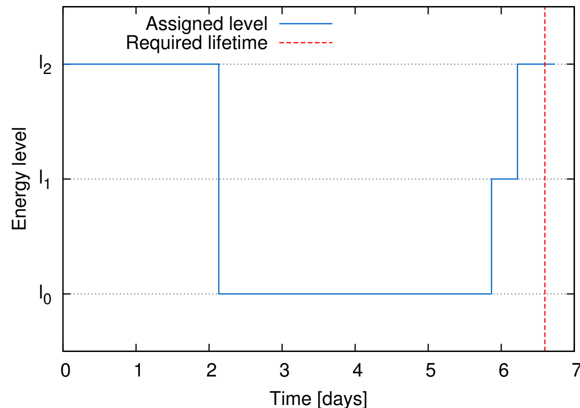

Figure 10. Level assignment over time

the ones used to build the battery model. A detailed analysis of the recorded voltage readings showed that the nodes expected to have significantly more energy left than actually available. We attribute this to differences in the properties of the batteries. In fact, after updating our battery model a lifetime of 10,0000 minutes could be achieved again. This shows that a good representation of the battery characteristics is needed for *Levels* to accurately meet lifetime goals.

The lifetime achieved by the sensor nodes in all other experiments was between 1.1 % and 6.5 % longer than the lifetime requested. Considering variations in battery capacity and the relatively small number of possibilities to adjust the level for this short-lived application these numbers are excellent. Although the variations due to external influences are larger here, the results correspond to our simulations. Therefore, they validate that the models used in the simulator capture the relevant factors of real deployments.

In the perfect case, the sensor nodes start in their highest energy level and only switch to lower levels later if the lifetime goal can no longer be met. Thus they minimize the number of level changes and avoid frequent changes in application quality. Due to inaccuracies on the sensor nodes, this best case cannot be achieved at runtime.

This is shown in Fig. 10 that visualizes the energy levels assigned in one of our experiments with a Mica2 sensor node. As expected, the node starts with the highest level (level l_2) and after two days switches to the lowest level (level l_0). Then, however, near its target lifetime it switches back to higher levels. This behavior is due to the safety factor in our computations: The node detects that its estimation has been too conservative and tries to consume the energy available to provide the best application quality possible.

Although this behavior differs from the ideal case, the number of level changes is still very small; in this experiment just three such changes occurred within several days. Depending on the energy consumption of the application and the requested lifetime more changes can be necessary. Nevertheless, 70 % of all the simulations presented in the previous subsection required 10 or less actual level adjustments.

6.2 Real-World Applications

In this subsection we show how *Levels* can be applied to real-world applications. For this purpose we selected monitoring of volcanoes [26]. In this application there is usually no redundancy in the network topology and the required duration of the experiment is known in advance. Replacing batteries is extremely difficult due to the inaccessible deployment location. Moreover, large parts of each node's energy is used to power the sensor interface board. Therefore, if a node stops sampling data itself but continues forwarding data from other nodes, its lifetime can be extended significantly and network connectivity can be preserved much longer.

As a concrete example of this class of applications we chose the system used at Reventador [26] ("Volcano"). This system is a complex application that has been tested in real-world deployments. It stores sensor readings to flash memory and the base station can then request stored data. In addition, Volcano includes an in-network detection of volcanic eruptions.

Again we use the Avrora simulator for the evaluation. However, since this simulator did not include the custom sensor interface board used by this application, we had to add its energy consumption to the simulator's energy model. From the information available we assumed for the sensor board a current draw of 40 mA. Furthermore, Volcano has been originally created for Tmote Sky nodes while the prototype implementation of *Levels* assumes the Mica family of sensor nodes. Therefore, we ported the application to this hardware family. However, in order to keep changes to the application small we simulated for this evaluation a fictitious Mica2-like node that is – like the Tmote Sky nodes – equipped with more RAM.

The behavior of Volcano depends on the eruptions detected by the sensor nodes. We simulated these eruptions at random intervals such that on average one event occurred every 30 minutes. However, like in the real deployment not all of these eruptions were actually reported by the nodes if they, for instance, stopped sampling to transfer some data.

In the deployment of the application [26] some batteries with higher capacities than those in our battery model were used. Therefore, our simulated lifetimes are significantly shorter than those reported there; this reduction in lifetime is not due to *Levels* and can also be observed when simulating the original application with the parameters of our batteries.

In its original version Volcano does not include code to achieve a user-defined lifetime goal. Therefore, we specified some optional functionality using our energy level abstraction. Since sensing is the largest single energy consumer, we put this code into a separate energy level. If it is deactivated, the nodes turn off the energy-expensive sensor interface boards and stop analyzing, storing, and transmitting their data. However, they still fully participate in routing and thus forward data from other nodes.

We defined energy levels in two nesC modules that were then mapped to a single level in the application. Only minor changes to the existing code were necessary: about 20 lines of code had to be added or modified. Some larger effort was, however, needed to write the profiling functions, since no suitable unit test drivers were available. The size of this module is less than 200 lines of code. In addition, we were able to copy almost the complete nesC wiring from the actual application and reuse it for energy profiling.

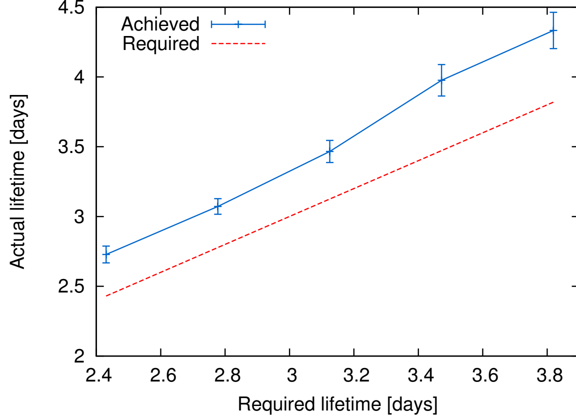

Figure 11. Average lifetime for Volcano (including 95 % confidence intervals)

Fig. 11 shows the average lifetime achieved by this application. In total we simulated 150 sensor nodes and none of them failed before its lifetime goal. However, since in this complex application the behavior of the nodes depends on network packets received and random events detected, accurately predicting future energy consumption from past data becomes more difficult than with the simple applications of Section 6.1.1. Therefore, the variations can be greater for this application. This is shown with the confidence intervals in Fig. 11, whose sizes are between 3.6 % and 6.0 % of the lifetimes requested. In addition, *Levels* is not able to completely consume the energy kept as a safety buffer and the nodes live on average 12.4 % longer than required. However, considering the lesser predictability of this application these results are still encouraging.

Just like for Volcano, *Levels* could also be applied to other existing applications. To get a better understanding of the effort needed for this change we analyzed the structural health monitoring application used at the Golden Gate Bridge [13]. For this application the sensor board is also one of the main energy consumers. Therefore, by applying energy levels just like in Volcano the lifetime of this application could be significantly extended. Similar effort as for Volcano would be needed for this modification, i.e., changing about 20 code lines in the application and writing measurement functions of less than 200 lines.

6.3 Runtime Overhead

Since with *Levels* each node determines its energy level independent from other ones, it does not have to send any radio messages. This helps to make *Levels* usable with low-power applications. Therefore, the only increase in energy consumption can be attributed to computational overhead. There are three sources for this overhead. First, whenever a code block belonging to an energy level is about to be executed, the system has to check if the level is active and has to add the block's energy consumption to its internal variables. Second, it accumulates continuous energy consumption to overall energy consumption every few seconds. Finally, every few hours *Levels* tries to adjust the current energy level with the Simplex algorithm.

Table 2. Runtime overhead

Optional code block	91 μs
Check-only for optional code block	11 μs
Adding continuous energy (2 levels)	30 μs
Adding continuous energy (5 levels)	107 μs
Adding continuous energy (10 levels)	235 μs

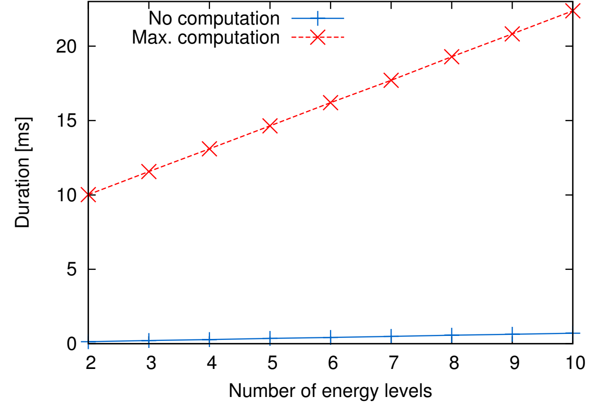

Figure 12. Duration of level adjustment

To evaluate the first kind of overhead, i.e., the overhead associated with each optional code block, we instrumented a simple application that makes use of the energy level abstraction. By simulating this application we were able to measure the CPU overhead in a controlled setting.

Table 2 shows the result of this experiment. The overhead associated with every code block is comparatively small: it is just 91 μs. However, sometimes the energy consumed by this overhead might still outweigh the energy consumed within the code block. In these cases the runtime system just checks if the code should be executed without adding its energy value. This takes only 11 μs. Therefore, even in this case when *Levels* is of less use, its runtime overhead still does not dominate the energy consumption of the code that it controls.

In another experiment we measured the overhead when accumulating continuous energy consumption. Here the results depend on the number of energy levels defined in the application. Although individual software components might all define their own energy levels, we expect that application developers will combine them when creating the overall application. Therefore, most applications will probably have less than five energy levels. As the numbers in Table 2 show, even for applications with twice this number the overhead is just a few hundred microseconds.

Finally, computing a new level assignment incurs the largest overhead. However, since this computation is only executed every few hours, the overhead is less critical. Fig. 12 shows the CPU overhead for two cases. In the first one, the energy consumed is too small compared to the current accuracy given by the battery model. Therefore, the actual computation is not performed. The other one, in contrast, shows the overhead when the computation is done for the maximum number of iterations. With more energy levels the overhead increases because for each level additional

Table 3. Effect of runtime overhead on node lifetime

Application	Lifetime with *Levels*	Reduction
Voltage level l_0	7.687 days	0.0 %
Voltage level l_1	6.648 days	0.0 %
Voltage level l_2	5.932 days	0.0 %
FFT level l_0	944.4 days	1.8 %
FFT level l_1	372.7 days	0.7 %

variables have to be considered both when no result can be computed and for solving the linear programming problem. Although an overhead of some milliseconds might seem significant, this computation is only executed every few hours or even days (depending on the energy consumption). Therefore, it should be acceptable for most applications.

To find out the actual effects of the computational overhead on node lifetime we simulated some of the test applications described in Section 6.1 with and without our runtime system doing its computations. As the results in Table 3 show, for short-lived applications with a lifetime of only a few days, the energy overhead of the computation does not result in a detectable decrease in node lifetime. Even for extremely low-power applications with a lifetime of several months or even years, the CPU overhead of our runtime system leads to reduction in lifetime of less than 2 %.

7 Discussion

This section discusses special application requirements on the network-wide behavior and on keeping energy levels constant for some time.

The level assignment described so far assigns energy levels independently on each node. Therefore, if the load on the nodes is roughly equal, all nodes in the network will change energy levels almost synchronously. Over time the overall quality of the network will be either excellent or poor, but nothing in between. Depending on the application this behavior might not be anticipated by the user.

One way to address this problem is to define the lowest energy level such that it is still useful to the application (e.g., nodes still sample data with lower-power sensors). Alternatively, the user can define the target lifetime in a way that it is actually achievable by sufficient nodes with their full functionality (e.g., at least by the leaves of the routing tree).

If such a definition of energy levels or lifetime requirements is not possible, distributing energy level assignments better over time often requires application-specific knowledge. For example, in some applications neighboring nodes should be fully active at the same time because they cooperate whereas in other applications nodes in high energy levels should be distributed uniformly throughout the network. In addition, for particularly energy-restricted applications even the slightest overhead for coordination might be prohibitive. Therefore, this problem can be addressed best by the application. Nevertheless, *Levels* provides two supporting mechanisms: The application can give hints which level to select and *Levels* can introduce some randomness.

With the first mechanism the application performs its own coordination among nodes. It does not, however, directly assign an energy level because otherwise the lifetime goal could possibly not be met. Instead, it tells *Levels* which energy level from the result of the optimization problem to select first: the lowest or the highest one. Over its lifetime each node will still use all energy levels from the results. Therefore, this approach cannot guarantee that really all nodes selected by the application operate in one specific level. However, it ensures that the lifetime goal can actually be met.

The second mechanism is similar but – instead of using information from the application – with this approach each node selects randomly whether to use the lowest or the highest level from the level assignment first. Although this mechanism might not provide optimal results, it is well-suited for low-power applications since it does not require any coordination among nodes.

Besides distributing level assignments in the network some applications might require that the system cannot change the currently active level while the node is, for example, sensing or analyzing data. *Levels* provides an interface for such applications that temporarily prevents level changes. To avoid that nodes stay too long in the wrong level, however, such periods should be relatively short. If a new level assignment is necessary during this time, it is applied immediately after the application allows level changes again.

8 Conclusions and Future Work

In this paper we have described and evaluated *Levels* which – unlike most approaches that try to maximize network lifetime – helps to meet user-defined lifetime goals for each individual node of a sensor network. It requires only small modifications to existing code and its energy levels provide a flexible and easy-to-use programming abstraction. With only minimal extensions to nesC *Levels* allows to mark code that is not needed to provide some basic functionality like network connectivity or sampling with less energy-intensive sensors. In addition, measuring the energy consumption of parts of an application is easy with our simulation-based approach for energy profiling. Finally, our runtime system shields the application developer completely from low-level issues related to lifetime estimation.

If an accurate battery model and information about the energy consumption of the sensor nodes are available, *Levels* helps to ensure that each node meets its lifetime goal and provides an application quality that is close to the optimum. *Levels* assumes that future energy consumption can be predicted from information about the past. Although we expect this assumption to be true for the long periods between level adjustments, *Levels* might not be able, however, to meet the lifetime goal in all cases if the node's load differs significantly over time.

We have shown in the evaluation that it is possible to benefit from *Levels* in complex applications without changing the code significantly. Moreover, the energy-overhead of the runtime system is so small that it can be almost neglected for both short-lived and long-lived applications.

In conclusion, we expect that *Levels* will help to make the creation of energy-aware sensor network applications much easier. For applications that cannot benefit from redundant nodes like those in our motivating scenarios it will allow to preserve some minimal functionality for the lifetime defined by the user. Although this might somewhat decrease

the quality of the data obtained from the network, we argue that – especially in a sparse network topology – a node is more useful when providing reduced functionality than if it stops working completely.

Regarding future work, we want to provide some application-independent coordination mechanisms for dense networks. In this case some local coordination among redundant nodes could ensure without much overhead that the average energy level of all nodes remains constant over time. In addition, we would like to extend our model to include a prediction of battery recharging when using energy harvesting. Furthermore, currently *Levels* tries to compute level assignments at a fixed rate. Adjusting the time interval between those computations to the energy consumption could possibly further improve application quality. Finally, we want to integrate information obtained from energy profiling into a sensor network IDE in order to make the developer aware of energy-expensive code.

9 Acknowledgments

This work was supported, in part, by Landesstiftung Baden-Württemberg. We thank our shepherd, Richard Han, and the anonymous reviewers for their helpful feedback and comments.

10 References

[1] E. F. Camacho and C. Bordons. *Model Predictive Control*. Springer-Verlag, 2nd edition, 2004.

[2] M. Cardei and J. Wu. Energy-efficient coverage problems in wireless ad-hoc sensor networks. *Computer Communications*, 29(4):413–420, 2006.

[3] A. Cerpa and D. Estrin. ASCENT: Adaptive self-configuring sensor networks topologies. In *Proc. of the Twenty-First Annual Joint Conf. of the IEEE Computer and Communications Societies*, volume 3, pp. 1278–1287, 2002.

[4] V. Chvátal. *Linear Programming*. W. H. Freeman and Company, 1983.

[5] A. Dunkels, F. Österlind, N. Tsiftes, and Z. He. Software-based on-line energy estimation for sensor nodes. In *Proc. of the Fourth Workshop on Embedded Networked Sensors*, 2007.

[6] Duracell Batteries. Duracell Plus alkaline-manganese dioxide battery. `http://www.mdsbattery.co.uk/datasheets/duracell/MN1500PL.pdf`.

[7] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proc. of the Seventeenth ACM Symposium on Operating Systems Principles*, pp. 48–63, 1999.

[8] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of the Conf. on Programming Language Design and Implementation*, pp. 1–11, 2003.

[9] A. Giusti, A. L. Murphy, and G. P. Picco. Decentralized scattering of wake-up times in wireless sensor networks. In *Proc. of the 4th European Conference on Wireless Sensor Networks*, pp. 245–260, 2007.

[10] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 93–104, 2000.

[11] C.-F. Huang, L.-C. Lo, Y.-C. Tseng, and W.-T. Chen. Decentralized energy-conserving and coverage-preserving protocols for wireless sensor networks. *ACM Trans. Sen. Netw.*, 2(2):182–187, 2006.

[12] JUnit web site. `http://www.junit.org`.

[13] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon. Health monitoring of civil infrastructures using wireless sensor networks. In *Proc. of the Int'l Conf. on Information Processing in Sensor Networks*, pp. 254–263, 2007.

[14] O. Landsiedel, K. Wehrle, and S. Götz. Accurate prediction of power consumption in sensor networks. In *Proc. of the Second Workshop on Embedded Networked Sensors*, 2005.

[15] T. Liu, C. M. Sadler, P. Zhang, and M. Martonosi. Implementing software on resource-constrained mobile sensors: Experiences with Impala and ZebraNet. In *Proc. of the Int'l Conf. on Mobile Systems, Applications, and Services*, 2004.

[16] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[17] P. J. Marrón, D. Minder, A. Lachenmann, and K. Rothermel. TinyCubus: An adaptive cross-layer framework for sensor networks. *it – Information Technology*, 47(2):87–97, 2005.

[18] P. J. Marrón, O. Saukh, M. Krüger, and C. Große. Sensor network issues in the Sustainable Bridges project. In *European Projects Session of EWSN 2005*, 2005.

[19] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *Proc. of the Int'l Conf. on Embedded Networked Sensor Systems*, pp. 95–107, 2004.

[20] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proc. of the Int'l Conf. on Information Processing in Sensor Networks – SPOTS*, 2005.

[21] R. Rao, S. Vrudhula, and D. N. Rakhmatov. Battery modeling for energy-aware system design. *Computer*, 36(12):77–87, 2003.

[22] V. Shnayder, M. Hempstead, B.-r. Chen, G. Werner-Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proc. of the 2nd Int'l Conf. on Embedded Networked Sensor Systems*, pp. 188–200, 2004.

[23] B. Titzer, D. Lee, and J. Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proc. of the Fourth Int'l Conf. on Information Processing in Sensor Networks*, pp. 477–482, 2005.

[24] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong. A macroscope in the redwoods. In *Proc. of the Int'l Conf. on Embedded Networked Sensor Systems*, pp. 51–63, 2005.

[25] T. van Dam and K. Langendoen. An adaptive energy-efficient MAC protocol for wireless sensor networks. In *Proc. of the Int'l Conf. on Embedded Networked Sensor Systems*, 2003.

[26] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proc. of the Symp. on Operating Systems Design and Implementation*, 2006.

[27] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. ECOSystem: Managing energy as a first class operating system resource. In *Proc. of the Int'l Conf. on Architectural Support for Programming Lang. and Operating Syst.*, pp. 123–132, 2002.