Virtual Battery: An Energy Reserve Abstraction for Embedded Sensor Networks

Qing Cao, Debessay Fesehaye, Nam Pham, Yusuf Sarwar, and Tarek Abdelzaher
Department of Computer Science
University of Illinois at Urbana-Champaign
{qcao2, dkassa2, nampham2, mduddin2, zaher}@uiuc.edu

Abstract

This paper introduces the abstraction of energy reserves for sensor networks that virtualizes energy sources. It gives each of several applications sharing a platform the illusion of having its own private energy source. Energy virtualization is the next logical step in embedded systems after virtualizing communication links and CPU capacity. Energy virtualization has not been addressed in past sensor network literature because most current wireless sensor networks feature single-user applications. To amortize deployment costs, future sensor networks, deployed in remote or hard-to-access areas, will likely be leveraged by scientists from different disciplines, each having their independent application for their individual research purposes. Platforms, planned for such deployment, will be fitted with the union of sensors needed, but independent applications will share the remaining resources such as in-field storage and communication bandwidth, calling for quotas and isolation mechanisms. The most expensive resource shared in sensor networks is energy. This paper provides an energy isolation mechanism, called the virtual battery*, that logically divides energy among applications to provide each its private energy reserve. An application can manage its private energy independently as if it were running alone on the platform. The application is terminated when its reserve is depleted. We implement and evaluate this abstraction on MicaZ motes running LiteOS. Our results show that the virtual battery mechanism succeeds at exporting the private reserve abstraction accurately and at a low overhead.*

1 Introduction

Energy is generally recognized as a key bottleneck for embedded sensor nodes. This bottleneck is exacerbated by the disparity between the rapidly growing processing speed and the slowly improving battery capacity of computing systems. Energy virtualization is therefore of increasing importance to partition the bottleneck resources appropriately, when multiple independent applications share a single platform.

Prior research that addressed the energy bottleneck focused on energy conservation approaches in wireless sensor networks that minimize energy consumption. This problem formulation inherently assumed cooperative applications, motivating a global optimization approach.

In contrast, in this paper, we consider sensor networks that serve as common platforms for scientific research, where the concern is with implementing isolation as opposed to cooperative sharing. The immediate motivation of our work comes from an outdoors sensor network testbed, currently being deployed at the University of Illinois as a general platform for research in environmental science. The network will serve as a common resource for multiple research teams to use subsets of available sensors for their individual research purposes. It will provide the necessary infrastructure including batteries, solar energy, Internet access, in-field processing capacity, and in-field storage. This "public" usage model is likely to proliferate, motivated by scenarios where the infrastructure cost needs to be amortized. For example, a sensor network for monitoring polar ice caps might be used by independent research teams to address different scientific observation-based questions enabled by the available access and sensing modalities. For another example, a network deployed in a tropical rainforest might be shared by projects that study changes in species populations using acoustic traces and ones that monitor climate change effects using environmental sensors. Not unlike high-power telescopes and other unique scientific instrumentation, sensor networks deployed in remote areas may be "rented" by different research teams to accomplish their tasks. A more cost-efficient usage is achieved when more than one team can leverage the network at a time, deploying their own application-specific in-situ data filtering and (pre)processing code as opposed to shipping all raw data to base by default.

When multiple applications are deployed concurrently, they should be properly isolated from each other, so that the execution of one does not affect resources "rented" to another. Resource virtualization and performance isolation become key concerns. Most prior work on virtualization ad-

1052-8725/08 $25.00 © 2008 IEEE
DOI 10.1109/RTSS.2008.41

dressed partitioning of communication resources (e.g., using weighted fair queueing or TDMA [19, 21]) and partitioning of CPU resources (e.g., using processor capacity reserves [4, 15, 17]). In contrast, motivated by the energy-constrained nature of sensor networks, we focus on virtualization and partitioning of battery capacity. The new abstraction is thus different from previous energy management research in the same sense that CPU capacity reserves offer a view different from priority scheduling.

The abstraction of energy reserves gives each application the illusion of having its own private energy as if it were executing alone on the platform. The application is allowed to manage its energy at will, after it is allocated with a share of the physical battery based on factors such as how much the application developers are willing to pay or the priority of the deployed tasks. For example, the application may turn off the radio or duty-cycle the CPU to conserve its energy. Since, in fact, the application is not alone, such resource management operations are also virtualized. The energy virtualization engine manages the physical resources accessed by multiple applications, giving each application the illusion that their resource management calls succeeded and hence charging their reserves only for those (virtual) resources that consume energy at the time. We show that this illusion is implementable even on resource-constrained sensor nodes and, in fact, results in energy savings in that true energy expended is generally less than the sum total charged to applications per the above abstraction. For example, when two applications keep their CPU idling when not in use, both reserves are liable to be charged for the idle energy (which they would have consumed if they were alone), yet this energy is physically expended only once.

Our paper makes the following main contributions. First, it provides the first abstraction designed exclusively to support energy isolation and related resource virtualization in wireless sensor networks. Our implementation is efficient, based on modifying low level operating system code, and hence introduces limited overhead.

Second, we demonstrate by an actual implementation that the virtual battery abstraction is feasible even on extremely resource-constrained platforms, such as MicaZ nodes with only 4K bytes of RAM. Energy reserves will be released as a standard feature in a future version of LiteOS. Having said so, we do not intend this abstraction to be platform or operating system dependent. Instead, due its low overhead, we believe that it can be easily integrated into other representative platforms and operating systems.

Third, we perform a systematic evaluation of our implementation, demonstrating that it is sufficient to provide the isolation needed to support sensor networks as common research platforms shared by multiple groups of users. Allowing different types of applications to run concurrently without energy interference is a crucial step towards the widespread commercialization and adoption of wireless sensor network systems.

Finally, a note is due on what our abstraction is not. Our goal is to virtualize energy. Any pitfalls or energy drains that tax an application's battery when running (physically) alone are allowed to tax it when running on a private virtual battery. For example, energy drained due to overhearing of extraneous traffic, when the application keeps its radio on, will also be drained from an application's reserve, unless the application turns off its (virtualized) radio. Hence, the semantics of running on a virtual battery are the same as those of running on a real one; no better and no worse.

The rest of this paper is organized as follows. Section 2 describes the related work. Section 3 describes the design choices of virtual battery. Section 4 describes the implementation details of virtual battery on the LiteOS operating system. Section 5 provides evaluation results. Section 6 concludes this paper.

2 Related Work

In the realm of wireless sensor networks, energy conservation has been the focus of a lot of contemporary research work. Motivated by the disparity between CPU capacity and battery lifetime, research on minimizing energy consumption has proposed techniques including putting nodes or radio into sleep mode, also known as duty scheduling [5, 23, 25], low-power communication protocols [18, 20], and system level support [11, 13]. These solutions have generally assumed cooperative, single-user applications for the proposed energy conservation optimizations.

This paper considers the type of sensor networks that serves as a common platform for scientific research, where implementing isolation between applications is crucial for multiple users. This application model is different from previous scientific deployments, such as the Great Duck Island deployment [7], the volcano monitoring project [24], and the redwood tree deployment [22]. These deployments feature single purpose applications based on the TinyOS operation system [14], where energy isolation for multiple applications is not considered because only one thread is supported.

Similar to our approach is the CPU reserve abstraction [4, 15, 17] and communication link sharing and scheduling [12, 19, 21]. The CPU reserve abstraction provides predictability for applications by isolating them from each other in timing, similar to the way a memory protection mechanism isolates their memory accesses. Based on the reserve abstraction, one task can reserve a portion of CPU capacity, and is guaranteed its availability if such a request succeeds. Communication link sharing, on the other hand, studies the resource scheduling problem for communication links, aiming to provide applications certain quality-of-service (QoS) guarantees, such as guaranteeing reserved communication bandwidth.

3 Design

Core to the energy reserve abstraction is an interface called *battery capacity reserve*. This interface allows applications to reserve a portion of the battery. If reserved successfully, this energy portion can be managed by the application at will. The application terminates once its energy reserve is depleted. The design details of this interface are described in the following sections.

3.1 Design Requirements

To support a pure virtual battery interface, the design should allow an application to specify only how much energy it reserves in total. This energy reserve becomes its virtual battery. No other limitations or parameters are imposed. However, to aide the application with energy management, our energy reserve abstraction exports a few more parameters that are useful for energy-constrained systems, but are not typically implemented in regular physical batteries.

First, physical batteries (to a first degree of approximation) make all their stored energy available for use at one time. This is analogous to receiving one lump payment into a bank account. The user is free to spend as much or as little of it as they wish. In practice, some may prefer an annuity over the lump sum payment. Hence, a virtual battery could help the application "pace itself" by making its allotted energy available in smaller installments over time (into a logical energy account). The unspent account balance will accumulate if not used. Hence, an application may define a desired lifetime and a payment interval. The total energy due over the lifetime is then divided by the number of payments and deposited into the application's reserve at the period indicated. Note that, the above does not mean that the application must spend equal amounts of energy over time. The application may choose to save its energy for some time, then spend the savings at a faster rate.

Continuing with the financial analogy, another related issue is account overdraft. What happens if the application spends more energy than is currently available in their account? Some applications may prefer overdraft protection (i.e., no overdraft). Others may prefer a credit line that allows them to borrow from future payments. Hence, our virtual battery abstraction allows a configurable credit line to be established. An application may spend its energy until the balance of their account becomes negative by an amount equal to the credit line.

Finally, an application may have some idea about their peak energy consumption. If peak consumption is exceeded, it would be useful to have a protection mechanism that enforces the maximum burn rate and alerts the application. The virtual battery provides a mechanism for such enforcement.

The design also calls for systematic support for virtualizing activities that involve energy management, to provide the illusion that each task is executed exactly the same as if it were alone. We call this illusion, *energy isolation*. For example, if tasks manage their energy by turning off the radio or duty-cycling the CPU, such activities should still succeed. Since the task is, in fact, not alone, a virtualization engine should manage the physical resources accessed by multiple applications, and virtualize them to support concurrent, potentially conflicting, requests. The energy isolation abstraction is defined has the following two requirements:

- *Portability:* First, an application that performs energy management (when running alone on a real battery) should not have to change its code when ported over to run on a shared platform with virtual batteries. Hence, calls such as those that put the processor to sleep must not have to change when running on the shared platform (although their behavior may need to change since, on a shared platform, other applications also need the resource).

- *Energy compatibility:* Second, the virtualized energy management calls on the shared platform must result in *at most* the same energy consumption that the application would have incurred if it were executing them on a dedicated platform alone. This ensures that virtualization does not degrade energy savings.

Below, we describe virtual batteries in more detail and present mechanisms that maintain energy reserves and enforce energy isolation.

3.2 The Energy Reserve Abstraction

To address the above design requirements, the energy reserve interface comprises of a five-component tuple. For a task T_i, its tuple is denoted by $(W_i, N_i, L_i, C_i, B_i)$, where:

- W_i is the percentage of total (physical) battery energy reserved by this task.

- N_i is the number of energy installments (annuity payments) requested.

- L_i is the expected task lifetime.

- C_i is the credit line rate, which defines the maximum amount of energy the task can borrow from future installments, as a fraction of the total amount of remaining installments due.

- B_i is the maximum energy burn rate.

Given a total physical battery capacity, E, application i is thus allotted a total amount of energy W_iE (allocated in lump sum or in regular annuity installments). The expected lifetime L_i divided by the number of installments requested, N_i, gives the period of installment payments, which we call

epoches, $P_i = L_i/N_i$. The first installment occurs at system start time. If $N_i = 1$, a lump sum allocation is requested.

When $N_i > 1$, the credit line, C_i, decides the maximum amount of energy a task can overdraft its account by. Note that, unlike the case with banking transactions, a task cannot *return energy* to the battery. Hence, it can only borrow from its own future energy installments. The parameter C_i is therefore defined as the *fraction* of remaining energy due to that task that the task can overdraft its account by. If the task received n_i payments out of N_i, its remaining payments add up to $W_i E(1 - n_i/N_i)$ and the maximum allowable negative balance of its account becomes $C_i W_i E(1 - n_i/N_i)$. If the balance drops below that point, the task will be suspended. If C_i is 0, no negative balance is allowed. If C_i is 1, all remaining energy can be used at any time. The overdraft is evaluated at epoch boundaries. A task suspended in one epoch might become eligible to resume when the next energy installment arrives at the next epoch.

The last parameter of the tuple, B_i, is the maximum burn rate, which limits the maximum rate at which a task can consume energy. For an epoch of length P_i, it translates into a maximum amount of energy, equal to $B_i P_i$, that the task can spend. Once that amount is reached within an epoch, the task is suspended until the end of the epoch, and a flag is set to indicate that the maximum burn rate was exceeded. The task may inspect this flag when it is allowed to resume.

Our energy management software maintains the energy balances for all tasks. For each task, T_i, an energy account balance, $Energy_i$, is maintained, and the number of installments, n_i, that the task already received is counted. The maximum allowable overdraft, $Maxover_i$, is also maintained for each task. At the beginning of an epoch for task T_i, these variables are updated as follows. Here, n_i is always limited by N_i:

$$n_i \leftarrow n_i + 1; \quad (1)$$

$$Energy_i \leftarrow Energy_i + W_i * E/N_i \quad (2)$$

$$Maxover_i = C_i W_i E(1 - n_i/N_i) \quad (3)$$

Let δ_i be a running counter of the amount of energy task T_i spends in its current epoch. A task is suspended if:

$$\delta_i \geq B_i P_i, \quad or \quad (4)$$

$$Energy_i - \delta_i \leq -Maxover_i \quad (5)$$

The virtual battery abstraction supports flexible energy management policies chosen by applications. Figure 1 shows several examples. In this figure, task A chooses lump sum energy allocation. Therefore, this task receives all its reserved energy at the beginning of its lifetime (and, in this example, quickly consumes it). Tasks B, C, and D all choose multiple installments. Additionally, task B requests a non-zero credit line, while task C sets its credit rate to 0. Task B always uses the maximum amount of credit, while in contrast, task C spends energy more evenly. Finally, task D accumulates its received energy for the first several epoches, and later consumes it in a relatively short period of time. The figure is strictly for conceptual illustration, while measurement based data are shown in Section 5.

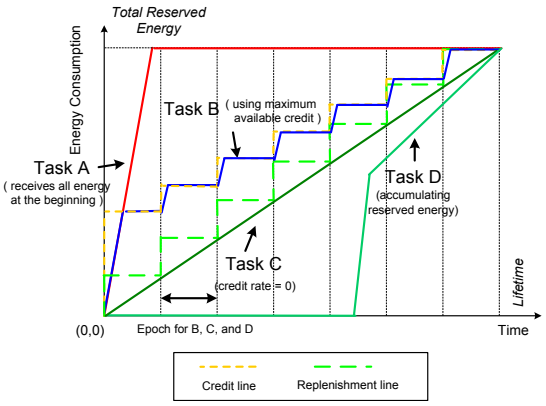


Figure 1. Examples of Energy Tuple Settings

3.3 Addressing Design Requirements

To support energy reserves, our design consists of two subsystems: (i) the energy reserve manager for performing accounting and enforcement per the energy reserve abstraction described above, and (ii) the virtualization engine for enforcing energy isolation. Their relationship is illustrated in Figure 2.

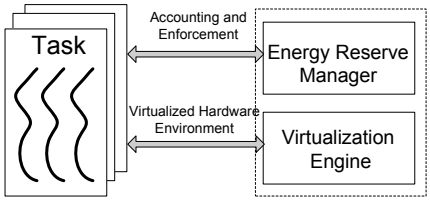


Figure 2. Design Architecture for Supporting Energy Reserve Abstraction

The energy reserve manager enforces the battery capacity reserve interface by accounting, replenishing, and if needed, suspending application tasks. Based on modifications to critical points in the task execution, such as context switches and device driver invocations, the manager enforces that tasks do not consume more energy than reserved. The manager associates energy reserves with logical *tasks* instead of *threads*, even though in most operating systems, threads are the basic units of scheduling. Here, a logical task (henceforth, simply a "task") refers to the traditional notion of operating system processes. Associating

reserves with tasks instead of threads has two advantages. First, a task may consist of multiple threads that collectively serve one application purpose. Therefore, tasks are more meaningful units to be associated with energy quotas. Second, sometimes one thread is executed on behalf of different tasks. For example, a routing protocol thread may be used by multiple applications to deliver packets at different times. Binding such threads with its own energy quota, therefore, is meaningless.

The second subsystem, the virtualization engine, provides a virtualized hardware environment to applications. It converts original device driver operations, such as duty-cycling CPU or turning off the radio, into functions that may or may not change the state of the physical hardware. For example, one task should not turn off the radio as long as the radio is still being used by another task.

Conceptually, virtualization occurs as follows. Consider a device, such as the CPU or the radio, that has multiple energy modes (e.g., on and off). Calls that control the energy mode of such a device are re-implemented by the virtualization engine. A default mode is adopted for each device. If the application has not made an explicit call to set the mode, it is said to have requested the default mode. When applications request conflicting modes for a device (e.g., one application keeps the radio on by default, while another requests to turn it off), the mode with the largest energy consumption is adopted and the application that requested it is charged. An implicit assumption we make is that performance improves with energy consumption. Hence, the mode with the largest energy consumption gives the best performance of all those actually requested. In other words, it errs on the safe side performance-wise. The same principle also applies for devices with more than two power modes.

For example, in an implementation that supports only two energy modes, i.e., if a device is either on or off, the device is physically turned off only when everyone requested that it be turned off. Otherwise, it is kept on and those applications that requested it to be on share the cost. If an application turns the device off, it is not allowed to use this device regardless of whether the device has in fact been turned off. This preserves the semantics of device calls causing application behavior to be similar to that when running on a dedicated platform. Hence, in our implementation, we also virtualize device calls so that these device calls will fail when the device is logically off when in fact it is physically on.

One key observation on our virtualization scheme is that it satisfies the two requirements of energy isolation described in Section 3.1. First, by construction of the API, our virtualization engine satisfies portability. In its implementation based on LiteOS, applications can run either on a dedicated (single user) version of LiteOS or on a shared (multi-user) version. The energy mode management calls in both are the same, hence, user applications don't need to be modified. But the calls in the multi-user version do not actually set the requested modes. Rather, they behave as discussed above.

Second, energy compatibility is achieved because when multiple applications request conflicting energy modes, one of the conflicting requests is always granted and the application is charged for that mode. Since the energy of that mode is, in fact, incurred only once, the other applications need not be charged at all. Hence, from any one application's perspective, it is either charged correctly or not charged at all. Therefore, the energy consumption from its reserve is no more than if it has in fact been running alone.

Observe that it is not the intent of energy isolation to virtualize the resources completely. For example, we do not address the fact that when more tasks share the same resource, such as the CPU, the performance degrades. This issue has been addressed in previous publications on resource virtualization and is not the concern of this paper [4, 15, 17]. Another example is the situation when two tasks, for example, try to configure the radio to operate on two different frequencies. These calls are not allowed on our multi-user platform as they interfere with application performance. For example, if one application sets the radio to a new channel, another might not be able to receive messages. Direct calls to set radio frequency are thus not allowed. Virtualizing these calls is not the concern of *energy* isolation.

Finally, we do not address issues of malicious applications in this paper. After all, the applications can still directly access hardware using low-level APIs and by-pass our abstractions. When different parties submit source code to execute on the shared platform, a central compile-time check can be performed to ascertain that the new application is not malicious (i.e., meets certain safety properties). If the compiler cannot determine with certainty that the code meets such properties (i.e., is not accessing restricted interfaces), it is returned for possible non-compliance. Verification of safety properties [1] has been an active area of research and, as such, is not the focus of this paper.

4 Implementation

We have implemented the virtual battery system on LiteOS [6], a thread-based operating system that provides Unix-like abstractions for operating and programming resource-constrained sensor nodes. It supports multiple applications to be concurrently executed as threads, which are bridged with the kernel through a suite of system calls.

Our implementation consists of two parts, the energy reserve manager, and the virtualization engine. For the former, we describe how we implement accounting and energy reserves. For the latter, we describe its virtualization of device driver operations.

4.1 Implementing the Energy Reserve Manager

4.1.1 Energy Control Blocks

The key data structure we implement to support energy reserves is energy control blocks (ECBs), illustrated in Figure 3. Associated to application tasks, ECBs encompass all energy consumed by their activities. Such activities may involve reading sensors, data processing, writing to files, etc. In addition to the five component tuple $(W_i, N_i, L_i, C_i, B_i)$, ECBs also keep up-to-date information on the energy consumption of tasks, such as remaining energy and the elapsed time, as part of the current state of tasks. Observe that one ECB block may be associated with multiple threads through runtime bindings, allowing us to control their aggregate energy consumption.

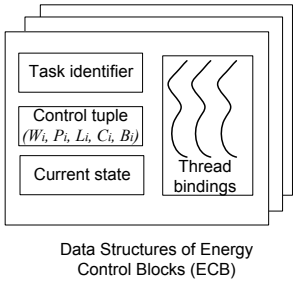


Figure 3. ECB Block Structure

4.1.2 The LiteOS Execution Model

The way that the energy reserve manager accounts for energy consumption of tasks is closely related to the execution model of LiteOS, as shown in Figure 4. In this model, the kernel is a priority based scheduler that loops over a queue of posted LiteOS jobs. When no new job is available, the kernel is put into sleep mode. Otherwise, it processes existing jobs following the default scheduling policy.

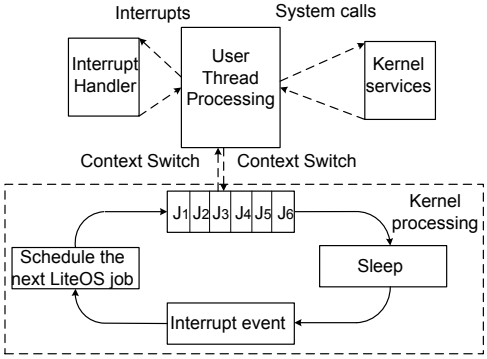


Figure 4. LiteOS Execution Model

To execute a thread, a job named `threadtask` is posted, which invokes the `switch_to_thread` function.

Kernel Task List		
Name of the Operation	Parameters need for accounting	Context switch
Led operations	None	No
Create a file	File Name	Yes
Read a file	Length of the file	Yes
Write a file	Length of the file	Yes
ADC sensor sampling	Sensor types	No
Radio send operation	Message length	Yes
Radio receive operation	Message length	Yes
Serial port send operation	Message length	Yes
Serial port receive operation	Message length	Yes
Read data from EERPOM	Length of data	No
Write data to EERPOM	Length of data	No

Table 1. Kernel Services provided through System Calls

This function saves the current execution context, switches to the user thread, and yields the CPU. During the execution of a thread, it may access kernel services, such as device driver operations, via system calls. Some device driver operations require the thread to temporarily yield CPU back to the kernel, so that the kernel can perform actions on behalf of itself. The following code example shows such an operation, where a thread reads data from an external file.

```
1   void mfread(MYFILE *fp, void *buffer, int nBytes)
2   {
3       current_thread = getCurrentThread(); /* get thread handle */
4       currentthreadindex = getCurrentThreadIndex();
5       mfile = getFileMutexAddress();
6       Mutex_lock(mfile);
7       /* Next set up the control parameters */
8       (*current_thread)->filedata.filestate.fileptr = (uint8_t*)fp;
9       (*current_thread)->filedata.filestate.bufferptr = (uint8_t*)buffer;
10      (*current_thread)->filedata.filestate.bytes = nBytes;
11      readFileSysCall();
12      Barrier_block(7, 3); /* wait for the kernel to perform file read */
13      Mutex_unlock(mfile); /* the kernel returns CPU to the thread */
14      return;
15  }
```

As the kernel may perform activities on behalf of application threads, accurate accounting of thread energy consumption should extend to the kernel. The file reading operation is not the only one that requires kernel service in LiteOS. Table 1 shows a list of representative activities performed through system calls, and whether this activity extends to the kernel through context switches.

4.1.3 Energy Accounting

The reserve manager implements energy accounting in software, where it instruments critical functions in the kernel to gather energy related information. It then converts such information into energy consumption in joules. Table 2 shows the different types of energy accounting, the instrumented functions, the collected information, and the energy conversion details.

Accounting for Consumed CPU Cycles To estimate energy consumption for CPU cycles, the reserve manager counts how many CPU cycles are consumed and for what

Type of activities	Instrumented functions	Collected information	Energy cost (Measurement details in Section 5.1)
CPU cycles for application processing	Critical kernel functions, e.g. `switch_to_thread()`, `yield_thread()`, `thread_task()`	Number of CPU cycles through timestamps	3.26μJ per 1000 cycles
Sensor samplings	Included in CPU cycle accounting	-	-
EEPROM operations	Included in CPU cycle accounting	-	-
Serial port operations	Included in CPU cycle accounting	-	-
CPU cycles for kernel processing	Critical kernel functions and interrupt handlers	Number of CPU cycles through timestamps	3.26 μJ per 1000 cycles
Energy cost for file operations	LiteOS API functions, e.g., `mfopen()`,`mfclose()`, `mfread()`, `mfwrite()`	Number of read/write bytes	$\sim$0μJ per byte read, and 11.81μJ per byte written
Energy cost for radio operations	LiteOS API functions, e.g., `radioSend()`, `radioReceive()`	Number of send/receive bytes	1.67μJ per byte sent, and 1.8 μJ per byte received
Hardware energy cost	Provided by the virtualization engine, accounted by associating soft state to tasks and timestamping device driver operations for calculating the energy cost	Device driver invocations	Depending on device

Table 2. Energy Accounting Instrumentation for the Kernel on the MicaZ node

purposes, then converts them into energy cost. There are two generic purposes of consumed CPU cycles: those for kernel processing of jobs and context switches, and those for executing threads, where each thread belongs to an accountable entity that is charged for energy consumption. Depending on type, interrupts also fall into one of these two purposes. To account for consumed cycles, the manager instruments the kernel at critical execution points, such as the start and the end of context switches, and keeps timestamps when these points are reached. The intervals between consecutive critical points are associated with accountable entities, which are in turn charged by converting CPU cycles into joules. This method of using timestamps has been proved useful and accurate for energy accounting in the literature [10].

We first describe how the manager maintains timestamps. The default timer provided by LiteOS is 8-bit with a maximum clock frequency of 32.768KHz, insufficient to provide the high resolution accounting needed for counting CPU cycles. We implemented a separate high resolution timer that provides a 48-bit cycle-accurate global timer through three 16-bit counters. Technically, the microcontroller hardware (Atmega128) provides two 16-bit timers (Timer 1 and Timer 3), but one of them (Timer 1) is already used by the CC2420 radio of MicaZ. Our implementation is based on the remaining one, Timer 3. One implementing challenge of this timing service is to accurately read out counter values, as the Atmega128 controller does not support atomic reads of 16-bit registers. In fact, every read operation of the counter consumes multiple CPU cycles, during which the counter is still counting. Our implementation therefore consists of a software-based adjustment module that ensures the accuracy of the timing service.

Note that the estimate on consumed CPU cycles already takes into account certain device driver invocations that do not require context switches, including reading ADC sensors, EEPROM operations, and serial port operations. Such device operations are not addressed separately.

Accounting for Device Driver Operations The second source of energy consumption is device driver operations that *require* context switches, such as sending and receiving packets through the radio. Such operations consume additional energy by drawing more energy with the use of external circuits, whose cost not only depends on the parameters of the operations, such as the length of data packets, but also on device configurations, such as the level of transmitting power for radio activities.

Our key insight to simplify energy accounting for device driver invocations is that their operations usually consist of a series of *atomic* actions, each with constant energy consumption. For example, a file write operation comprises of a series of identical, byte level serial flash writes. By premeasuring the energy consumption of writing one byte, the reserve manager can estimate accurately the energy cost by simply counting how many bytes are passed as the parameter.

Following this approach, we instrumented the LiteOS library APIs for such device driver invocations, as shown earlier in Table 1, to estimate energy consumption based on their parameters. Note that such instrumentation should only modify sufficiently low level APIs that can be decomposed into actions with constant energy cost. As a counterexample, the sending function provided by a MAC layer protocol should not be used because packets may be sent multiple times by the MAC layer, consuming variable amount of energy.

Accounting for Hardware Usage The third source of energy consumption that we account for is energy consumed by hardware. Such accounting is performed jointly with the virtualization engine, based on the usage profile of hardware devices by tasks. For example, if only one task requires the radio to be turned on, it will be the only task whose reserve will be charged for energy consumption.

4.2 Implementing the Virtualization Engine

As the virtual battery abstraction allows tasks to manage their own energy reserve to conserve energy, another subsystem we implement is the virtualization engine that provides an illusion that the energy saving protocols used by applications continue to be effective. Such protocols have been extensively studied in sensor networks, but most of them are based on the assumption that an application is executed alone on the platform. For multiple applications sharing the same platform, their energy management protocols may be in conflict with each other, calling for isolation and virtualization of their resource management calls.

In our current implementation on LiteOS, we address two types of energy management calls: duty-cycling the CPU and duty-cycling the radio. The union of these calls are sufficient to express most energy conservation protocols designed for sensor networks.

4.2.1 CPU Duty Scheduling Virtualization

The duty scheduling management provided by LiteOS allows the microcontroller to enter power-save mode for a flexible period of time. When the watchdog timer is disabled, this mode consumes 8μA of energy, or 0.1% of the 8mA consumed in active mode, revealing a great potential for energy management protocols to increase system lifetime.

Because duty scheduling calls only assume one of two states, their virtualization is simple. The virtualization engine instruments the duty scheduling calls from multiple applications by keeping the device in the mode with the largest energy consumption, in this case, keeping the device in the on state as long as at least one task requests so. The device is turned off when it is no longer requested by any active task.

Besides virtualizing duty scheduling, the engine also provides accounting information to the reserve manager, based on requests made by tasks. If one task has logically put the CPU into the power-save mode, it is no longer charged for the CPU energy even if, physically, the CPU is still active, whose energy cost is charged against only those tasks that require it to stay active.

4.2.2 Radio Duty-cycling Virtualization

Another primary source of energy consumption is radio operations. The CC2420 radio installed on MicaZ supports multiple modes of operation with different current draws, such as voltage-off mode (0.02μA), power-down mode (20μA), idle mode (426μA), transmit mode ($8.5-17.4$mA), and receive mode (18.8mA). It is therefore advisable for tasks to turn the radio into power down or voltage off mode to save energy when the radio is not being used.

Virtualized radio duty-cycling is implemented by keeping the radio in the mode that consumes maximum amount of energy when multiple requests are received. When one task invokes the `RadioVerfOff()` function to turn off the radio voltage regulator, the physical radio is not turned off until every task has invoked the same function. On the other hand, one task is no longer charged of energy cost after it invokes `RadioVerfOff()`, effectively putting its virtualized radio into off state. All packets sent by this task following a `RadioVerfOff()` request are ignored by the engine to provide an illusion that the semantics of the task remain the same.

5 Evaluation

In this section, we systematically evaluate the virtual battery abstraction in two parts. First, we empirically choose the parameters we use for energy accounting, such as the energy cost of CPU cycles, radio operations, and file system operations. Then, we show that the virtual battery mechanism succeeds at exporting the private reserve abstraction accurately for both CPU-bound tasks and I/O-bound tasks.

5.1 Measuring energy consumption for system operations

For accurate accounting energy for tasks, we need the energy consumption of various system operations. In addition to consulting data sheets of MicaZ [9], Atmega128 [2] and CC2420 radio chip [8], we also carry out extensive experiments with a power meter and an oscilloscope to measure (and to verify) their energy consumption. The results have been summarized earlier in Table 2, and we describe our main procedure as follows. To measure the energy cost of certain operation (e.g., writing to flash), we run the same operation (for example, writing to the flash by a low level system call `atmelflashread()`) repeatedly with a very short period (say, 50ms), and observe the power consumption by the mote for this operation. We connect the mote with a 3 volt DC battery, in series with a tiny resistor (6.1Ω). Since the operation is periodic, the mote almost consumes a constant current, although there are some flicks when it consumes additional power by writing to flash. To get a stable reading, we attach a big capacitor (3.5F) in parallel with the resistor. Figure 5 shows the setup of this experiment.

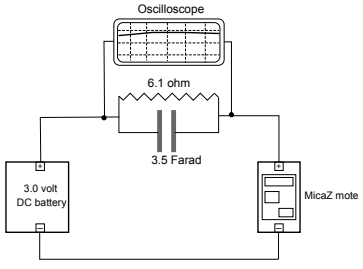
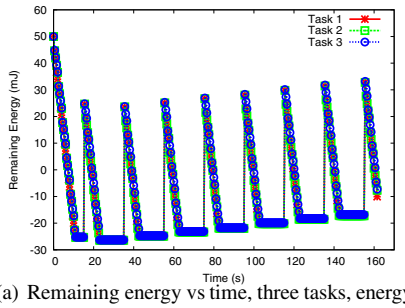
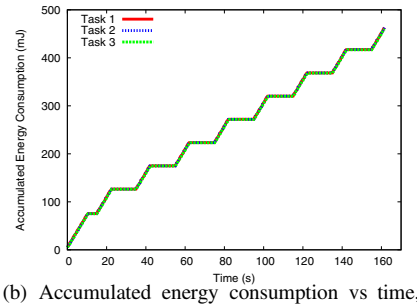


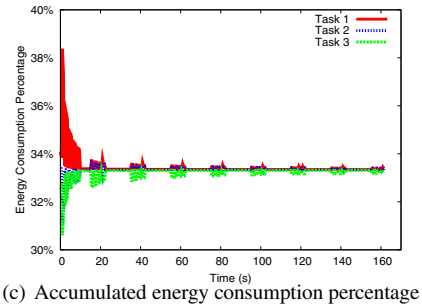
Figure 5. Experiment setup for measuring energy costs.



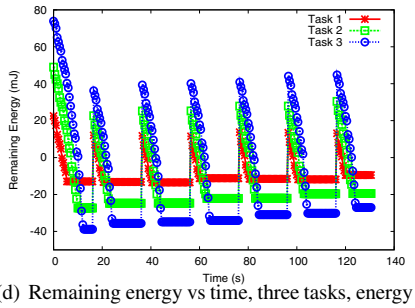
(a) Remaining energy vs time, three tasks, energy reserve ratio = 1:1:1, credit line ratio = 2.5%



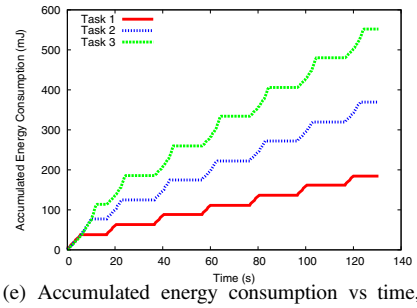
(b) Accumulated energy consumption vs time, same settings as part (a)



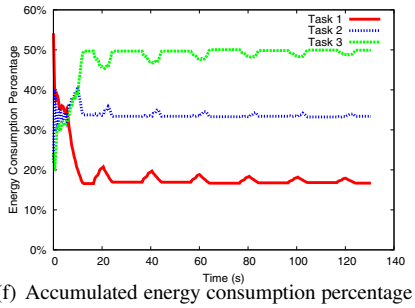
(c) Accumulated energy consumption percentage vs time, same settings as part (a)



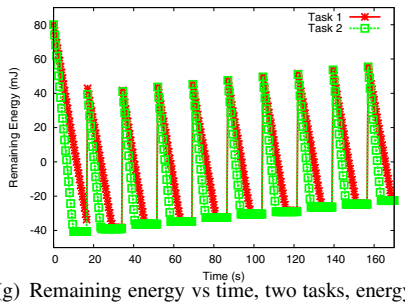
(d) Remaining energy vs time, three tasks, energy reserve ratio = 1:2:3, credit line ratio = 2.5%



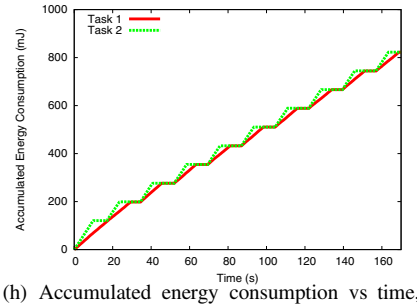
(e) Accumulated energy consumption vs time, same settings as part (d)



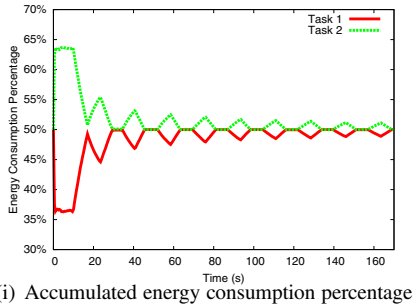
(f) Accumulated energy consumption percentage vs time, same settings as part (d)



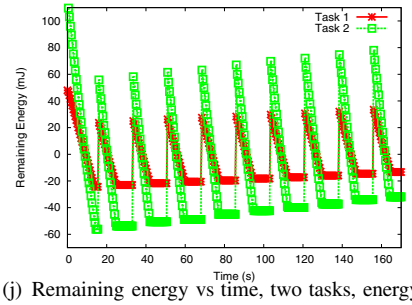
(g) Remaining energy vs time, two tasks, energy reserve ratio = 1:1, credit line ratio = 2.5%



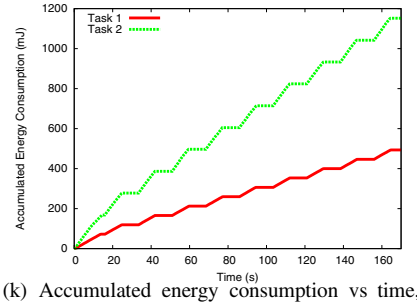
(h) Accumulated energy consumption vs time, same settings as part (g)



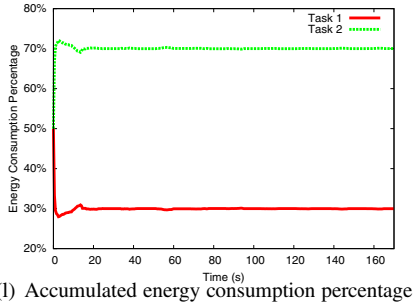
(i) Accumulated energy consumption percentage vs time, same settings as part (g)



(j) Remaining energy vs time, two tasks, energy reserve ratio = 3:7, credit line ratio = 2.5%



(k) Accumulated energy consumption vs time, same settings as part (j)



(l) Accumulated energy consumption percentage vs time, same settings as part (j)

Figure 6. Evaluations of Energy Reserves for both CPU-bound and I/O-bound Applications

CPU

According to MicaZ data sheet, when in idle mode and the radio off, the mote's CPU draws 8mA current, meaning it consumes $8 \times 3 = 24$mW power. In turn, for 1000 cycles, it consumes $\frac{8\times3\times1000}{7.3728\times10^6} = 3.26\mu$J. In experiment we measured its current to be 8.85mA, which is pretty close. In turn, for 1000 cycles, it consumes 3.43μJ in experiments.

Radio operation

For radio energy consumption, we deduce most values from CC2420 data sheet and we tried to verify the values by the real lab experiments. For measuring sending cost, we repeated send packets of size 50 bytes for every 50ms, and we measured the current going into the mote. Experiments show that the mote draws 21.31mA current. Hence the radio consumption is $21.31 - 8.85 = 12.46$mA (CPU consumption is deducted). But the data sheet says while sending, radio chip draws 17.4mA at 0dBm. Hence, our result does not exactly match with the data sheet. In receiving mode (idle listening), the mote takes 19.7mA according to the data sheet, and experiments find $27.86 - 8.85 = 19.01$mA, which is very close. It's interesting that CC2420 consumes more energy for receiving than sending.

Assuming 12.46mA consumption for sending bytes, we can compute the per byte energy cost for sending packets. The chip CC2420 sends at rate 250kbps, for every byte it consumes $\frac{12.46\times3\times8}{250\times10^3} = 1.20\mu$J, whereas according to data sheet it is 1.67μJ. Similarly, for receiving, we get 1.82μJ/byte from the experiment and 1.80μJ/byte from data sheet. In our following experiments, because our results are not sensitive to the particular chosen metrics, we still decide to use the values from the data sheet.

Flash read/write

To measure energy cost of read/write operation to flash, we turn the radio off. While reading from the flash, we repeatedly read consecutive flash pages (each page is of 264 bytes) with a 50ms interval. We do not read the same page again and again, because in that case the page will be served from a cache in the serial flash hardware instead of flash itself. In our experiment, we see no extra current drawing other than CPU for this read operation. Whatever amount and rate we read from the flash, we always see the constant CPU current 8.85mA. In the literature, [16] reported the measurement results as 0.26μJ per byte, supposedly using a different approach. In the data sheets [3], the energy consumption for read operation with 3V voltage is not provided.

For writing to the flash, we did the same experiment – write bytes to consecutive flash pages with 100ms intervals. When we write 1 page (264bytes) in every 100ms and we see the current 19.67mA. Therefore, the mote consumes $19.67 \times 3 = 59.01mJ$ in every second. This energy includes both CPU and flash write. Because in one second, we write 10 pages, i.e., 2640bytes, and CPU takes $8.85 \times 3 = 26.55\mu$J, we can calculate the energy per byte as $\frac{59.01-26.55}{2640} = 12.29\mu$J. In another experiment, we write 2 pages in every 100ms, and obtain a result of 11.27μJ/byte (current 28.69mA). In the third experiment, we write 3 pages, and get 11.86μJ/byte (current 40.16mA). Therefore, on the average, the energy cost for writing to flash is 11.81μJ/byte.

Other activities

There are three types of operations whose cost is already included in CPU cycles. These activities are sensor readings, serial port communication, and EEPROM operations. Their cost is not accounted separately. Note that if the virtual battery mechanism is extended to other platforms, the sensing board may draw additional energy, calling for separate accounting.

5.2 Evaluating the efficacy of the virtual battery abstraction

We instrumented the LiteOS kernel with the energy reserve manager and the virtualization engine. The kernel without instrumentation compiles to 83196 bytes of code and 2311 bytes of RAM. After instrumentation (including code for our experiments), the kernel compiles to 94326 bytes of code and 2648 bytes of RAM. We consider the increase in memory footprint to be moderate, given that we have modified the kernel extensively to provide energy isolation and virtualization. All experiments are based on LiteOS 0.3.3 running on MicaZ node.

We run multiple concurrent applications through the LiteOS shell with different energy reserves, and observe their energy consumption over time. To check the state of energy consumption, we periodically send the ECB contents of tasks over the serial port to the computer, and analyze the data to profile the task behavior. The difference between readings on remaining energy reflects energy consumption of tasks over time.

In the first experiment, we run three CPU-bound applications that perform intensive computing. We set the epoches to be the same for these applications. For these three tasks, we use two different settings. In the first, the energy reserves of tasks are equal, that is, with ratio of (1:1:1). Each task reserve 33.3% of the total energy. In the second, the reserves are set with a ratio of (1:2:3), where tasks reserve 16.7%, 33.3%, and 50% of the total energy, respectively. We set the credit line rate as 2.5% for tasks. We intentionally use a relatively small number of epoches to illustrate the change of available credit with time. In real systems, application lifetime will be much larger and the change of credit line in a short period of time may not be observable.

More specifically, the energy reserve tuples for the three CPU-bound tasks are $(W = 33.3\%, N = 20, L = 400s, C = 2.5\%, B = 100mJ/s)$. The total energy E is 3000mJ. While this is much smaller than typical battery can provide, it is sufficient for our evaluation purposes. A

small E also helps us better observe the trend of the credit line. The maximum energy burn rate B is chosen to be sufficiently large in the experiments. The results for this evaluation are shown in parts a–f of Figure 6. First, observe the energy reserves are enforced well, especially after the applications enter steady state. Also, the credit line decreases with time, because it is defined as proportional to the total remaining energy. Finally, the figures show that in the steady state, the energy consumption fraction of task T_i converges to W_i.

In the second experiment, we run two I/O-bound applications. The first application repeatedly sends packets containing a "hello, world" message through the radio for every 20ms. The second application repeatedly sends a constant length string over the serial port for every 20ms. We experiment with two energy reserve ratios, (1:1) and (3:7), respectively. The total energy E is 3200mJ. In the first setting, the energy reserve tuples for both tasks are $(W = 50\%, N = 20, L = 350s, C = 2.5\%, B = 100mJ/s)$. In the second setting, the energy reserve tuples are $(W = 30\%(task1)/70\%(task2), N = 20, L = 350s, C = 2.5\%, B = 100mJ/s)$. The results are shown in parts g–l of Figure 6. Observe that again, the energy reserves for both tasks are well enforced, the credit line decreases, and the energy consumption fractions of tasks converge to W.

6 Conclusions

In this paper, we presented the energy reserve abstraction for embedded sensor networks. To our knowledge, it is the first energy isolation and virtualization mechanism that provides energy reserve abstraction for resource-constrained sensor networks. It allows applications to reserve energy for their private use, and guarantees the availability of energy for successful reservations. By virtualizing the physical battery, it provides similar abstractions to CPU capacity reserves. We implemented a prototype of this abstraction on the LiteOS operation system running on MicaZ nodes. Our implementation and evaluation results demonstrate that it succeeds at exporting the private reserve abstraction accurately and at an acceptable system overhead.

Acknowledgements

This work is funded in part by NSF grants, NSF 05-54759, and NSF 06-26342.

References

[1] R. Alur, T. A. Henzinger, and P. hsin Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22:181–201, 1996.

[2] Atmel Corporation. *Atmega128 data sheet*, 2007. Available at http://www.atmel.com.

[3] Atmel Corporation. *Serial flash data sheet model AT45DB041D*, 2007. Available at http://www.atmel.com.

[4] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI*, pages 45–58, 1999.

[5] Q. Cao, T. Abdelzaher, T. He, and J. Stankovic. Towards Optimal Sleep Scheduling in Sensor Networks for Rare Event Detection . In *Proceedings of IPSN*, 2005.

[6] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He. The LiteOS Operating system: Towards Unix-like Abstractions for Wireless Sensor Networks. In *Proceedings IPSN*, 2008.

[7] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao. Habitat Monitoring: Application Driver for Wireless Communications Technology. In *Proc. of the 2001 ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean*, April 2001.

[8] Chipcon. *CC2420 Data Sheet*. http://www.chipcon.com/.

[9] CrossBow. *MicaZ data sheet*, 2007. Available at http://www.xbow.com.

[10] A. Dunkels, F. Osterlind, N. Tsiftes, and Z. He. Software-based online energy estimation for sensor nodes. In *EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors*, pages 28–32, New York, NY, USA, 2007. ACM.

[11] A. Eswaran, A. Rowe, and R. Rajkumar. Nano-RK: An energy-aware resource-centric rtos for sensor networks. In *RTSS*, 2005.

[12] T. Facchinetti, L. Almeida, G. C. Buttazzo, and C. Marchini. Real-time resource reservation protocol for wireless mobile ad hoc networks. In *RTSS*, pages 382–391, 2004.

[13] T. He, S. Krishnamurthy, L. Luo, T. Yan, R. Stoleru, G. Zhou, Q. Cao, P. Vicaire, J. A. Stankovic, T. F. Abdelzaher, J. Hui, and B. Krogh. VigilNet: An Integrated Sensor Network System for Energy-Efficient Surveillance. *ACM Transaction on Sensor Networks*, 2007.

[14] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. In *Proceedings of ASPLOS-IX*, 2000.

[15] M. B. Jones, D. Rosu, and M.-C. Rosu. Cpu reservations and time constraints: Efficient, predictable scheduling of independent activities. In *SOSP*, pages 198–211, 1997.

[16] G. Mathur, P. Desnoyers, D. Ganesan, and P. J. Shenoy. Ultra-low power data storage for sensor networks. In *IPSN*, pages 374–381, 2006.

[17] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *ICMCS*, pages 90–99, 1994.

[18] J. Polastre and D. Culler. Versatile Low Power Media Access for Wireless Sensor Networks. In *Second ACM Conference on Embedded Networked Sensor Systems (SenSys 2004)*, November 2004.

[19] S. Prabh and T. F. Abdelzaher. On scheduling and real-time capacity of hexagonal wireless sensor networks. In *ECRTS*, 2007.

[20] K. Seada, M. Zuniga, A. Helmy, and B. Krishnamachari. Energy efficient forwarding strategies for geographic routing in lossy wireless sensor networks. In *The Second ACM Conference on Embedded Networked Sensor Systems*, 2004.

[21] D. Stiliadis and A. Varma. Rate-proportional servers a design methodology for fair queueing algorithms. *IEEE/ACM Trans. Netw.*, 6(2):164–174, 1998.

[22] G. Tolle, J. Polastre, R. Szewczyk, N. Turner, K. Tu, S. Burgess, D. Gay, P. Buonadonna, W. Hong, T. Dawson, and D. Culler. A macroscope in the redwoods. In *The 3rd ACM Conference on Embedded Networked Sensor Systems*. ACM Press, 2005.

[23] X. Wang, G. Xing, Y. Zhang, C. Lu, R. Pless, and C. Gill. Integrated Coverage and Connectivity Configuration in Wireless Sensor Networks. In *First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, November 2003.

[24] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *OSDI*, pages 381–396, 2006.

[25] T. Yan, T. He, and J. A. Stankovic. Differentiated surveillance for sensor networks. In *First ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2003.