LiteOS, A Unix-like Operating System and Programming Environment for Wireless Sensor Networks

Sensys Paper #89

Abstract

This paper presents LiteOS, a UNIX-like, multithreaded operating system with object-oriented programming support for wireless sensor networks. Aiming to build an easy-to-use sensor network software platform, LiteOS offers a number of features that are not available in existing sensor network operating systems, including: (1) a built-in hierarchical file system and a wireless shell for user interaction using UNIX-like commands; (2) kernel support for dynamic loading and native execution of multithreaded applications; and (3) an object-oriented programming language that uses a subset of C++ as its syntax with class library support. LiteOS comprises an evolvable kernel for application execution and a compiler for application development, bridged by a suite of expandable system calls. We evaluate it experimentally by measuring the performance of common tasks using this operating system, and demonstrate its programmability through more than twenty application benchmarks.

1 Introduction

This paper introduces the first UNIX-like operating system, called LiteOS, that fits on memory-constrained motes such as MicaZ. This operating system is multithreaded and comes bundled with a UNIX-like file system and a C++ compiler. The authors believe that such an operating system could significantly expand the circle of sensor network application developers by providing a familiar programming environment based on UNIX, threads, and C++. While TinyOS and its extensions have significantly improved programmability of mote-class embedded devices via a robust, modular environment, NesC and the event-based programming model introduce a learning curve for most developers outside the sensor networks circle. The purpose of LiteOS is to significantly reduce such a learning curve. This philosophy is the operating system equivalent of network directions taken by companies such as Arch Rock [1] (that superimpose a familiar IP space on mote platforms to reduce the learning curve of network programming and management).

The rapid advances of sensor networks in the past few years created many exciting systems and applications. Operating systems such as TinyOS [16], SOS [15], Mantis [5], Contiki [9], and t-Kernel [14] provided software platforms. Middleware systems such as TinyDB [21] and EnviroTrack [4] made fast development of specialized applications feasible. Deployed applications ranged from global climate monitoring to animal tracking [18], promising unprecedented sampling of the physical world. Widespread adoption and commercialization of sensor networks is the next logical step.

The most obvious challenge in sensor network development has been to fit within extremely constrained platform resources. Previous work, such as TinyOS, therefore focused on reducing overhead and increasing robustness. Since initial users were researchers, compatibility with common embedded computing environments was not a major concern. Moving forward, to decrease the barrier to widespread adoption and commercialization, leveraging familiar abstractions is advisable. One approach is to build user-friendly applications and GUIs such as SensorMap [25] and MoteView [7]. In this paper, we explore a complementary solution that targets the operating system and programming environment.

We build a familiar software platform by leveraging the likely existing knowledge that common users (outside the current sensor network community) already have: Unix, threads, and C++. We introduce an interactive operating system, called LiteOS, and a programming language, called LiteC++. LiteOS supports a Unix-like environment for multithreaded user applications, and LiteC++ supports a subset of C++ (with one keyword from Java) for application development. They are referred to in general as the LiteOS platform, when no ambiguity arises. Since our primary goal is to increase adoption, we also optimize LiteOS for efficiency and robustness.

LiteOS differs from both current sensor network operating systems and more conventional embedded operating systems. Compared to the former category, such as TinyOS, LiteOS provides a more familiar environment to the user. Its features are either not available in existing sensor network operating systems, such as the shell and C++ compiler, or are only partially supported, such as dynamic loading support. Compared to the latter category (conventional embedded operating systems), such as VxWorks [31], eCos [2], embedded Linux, and Windows CE, LiteOS has a much smaller code footprint, running on platforms such as MicaZ, with a 8MHz CPU, 128K bytes of program flash, and 4K bytes of RAM. Embedded operating systems, such as VxWorks, require more computation power (e.g., ARM-based or XScale-based processors) and more RAM (at least tens of KBytes), and thus cannot be easily ported to MicaZ-class hardware platforms (such as MicaZ, Tmote, and Telos).

A possible counter-argument to our investment in a small-footprint UNIX-like operating system is that, in the near future, Moore's law will make it possible for conventional Linux and embedded operating systems to run on motes. For example, the recent iMote2 [6] platform by CrossBow features an XScale processor that supports embedded Linux. Sun and Intel also demonstrated more powerful sensor network hardware platforms [3, 24]. While it is true that more resources will be available within the current mote form factor, Moore's law can also be harvested by decreasing the form factor while keeping resources constant. For example, the current MicaZ form factor is far from adequate for wearable computing applications. Wearable body networks can have a significant future impact on healthcare, leisure, and social applications if sensor nodes could be made small

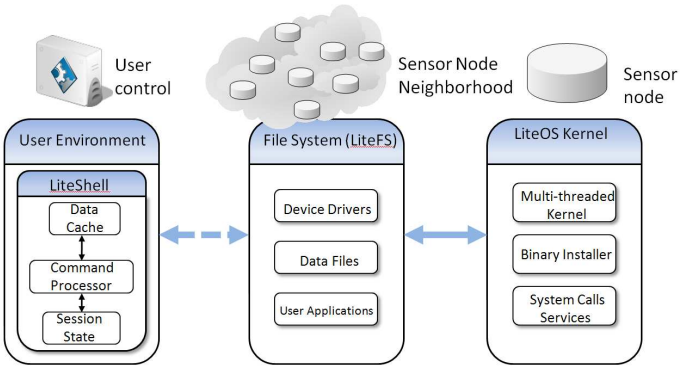


Figure 1. LiteOS Operating System Architecture

enough to be unobtrusively embedded in attire and personal effects. These applications will drive the need for small-footprint operating systems and middleware as computation migrates to nodes that are smaller, cheaper, and more power-efficient. This paper serves as a proof of concept by pushing the envelope within the constraints of a current device; namely, the MicaZ motes.

We have implemented LiteOS on the MicaZ platform. The LiteOS kernel is 8300 lines of C code. The LiteOS shell is 2200 lines of Java code (running on a PC). The kernel compiles into 30822 bytes of binary and consumes 1633 bytes of RAM (for an eight-thread, eight file-handler setting). The LiteC++ compiler is 12100 lines of C code (also running on a PC) and 1200 lines of Perl code. We test the system using 21 application benchmarks written in LiteC++, which themselves constitute 1260 lines of code. Comments and blank lines are not counted. Auxiliary tools, such as those written for debugging and testing the system, and device drivers adapted from other open-source research projects, mostly the CC2420 radio driver adapted from TinyOS 1.1.x, are also not counted. We are scheduled to release LiteOS as an open-source project for the research community in 2007.

The rest of this paper is organized as follows. Section 2 introduces the LiteOS operating system and describes its design and implementation. Section 3 introduces the LiteC++ programming environment through motivating examples, and present implementation details. Section 4 gives an overview of the future directions of LiteOS and its implications. Section 5 concludes this paper.

2 The LiteOS Operating System

In this section, we present the LiteOS operating system. We first present a system overview and design choices, then we describe its three subsystems. We include evaluation results in these subsections to increase readability.

2.1 Architectural Overview

Figure 1 shows the overall architecture of the LiteOS operating system, partitioned into three subsystems: LiteShell, LiteFS, and the kernel. Implemented on a base station, the LiteShell subsystem interacts with sensor nodes only when a user is present. Therefore, LiteShell and LiteFS are connected with a dashed line in this figure.

LiteOS is designed to support single-node functionality. It executes applications in a UNIX-like multithreaded environment. It also provides a wireless *node mounting mechanism* (to use a UNIX term) through a file system called LiteFS. Much like connecting a USB drive, a LiteOS node mounts itself wirelessly to the root filesystem of a nearby base station. Moreover, analogously to connecting a USB device (which implies that the device has to be less than a USB-cable-length away), the wireless mount works only for devices within wireless range. The mount mechanism comes handy, for example, in the lab, when a developer might want to interact temporarily with a set of nodes on a table-top before deployment. While not part of the current version, it is not conceptually difficult to extend this mechanism to a "remote mount service" to allow a network mount. Ideally, a network mount would allow mounting a device as long as a network path existed either via the Internet or via multi-hop wireless communication through the sensor network.

Table 1. Shell Commands

Command List	
File Commands	ls, cd, cp, mv, rm, mkdir, touch, chmod, pwd, du
Process Commands	ps, kill, install, uninstall
Group Commands	foreach, $, |
Environment Commands	history, who, man, echo
Security Commands	login, logout, passwd

Once mounted, a LiteOS node looks like a *file directory* from the base station. The shell, called LiteShell, supports UNIX commands, such as copy and move, executed on such directories. The external presentation of LiteShell is versatile. While our current version resembles closely a UNIX terminal in appearance, it can be wrapped in a graphical user interface (GUI), appearing as a "sensor network drive" under Windows or Linux.

The basic (stripped-down) version of LiteOS is geared for trusted environments. This choice of default helps reduce system overhead when the trust assumptions are satisfied. In more general scenarios, where security concerns are relevant, an authentication mechanism is needed between the base station and mounted motes. Low-cost authentication mechanisms for sensor networks have been discussed in prior literature and are thus not a focus of this paper [26].

2.2 LiteShell Subsystem

The LiteShell subsystem implements a Unix-style shell for MicaZ-class sensor nodes. It supports an expandable suite of commands, including both standard Unix ones, such as **ls** and **cp**, and several new ones, such as **foreach**. Currently, 23 commands, as listed in Table 2, are implemented. They fall into five categories: file commands, process commands, group commands, environment commands, and security commands.

File Operation Commands: File commands generally maintain their Unix meanings, e.g., the **ls** command lists directory contents. Typing **man ls** in the shell returns the following:

```
$ man ls
    LiteOS User Command: LS
    SYNOPSIS:
    List information about the files alphabetically.
    OPTIONS:
    -l:  list full file information
    -u:  disable shell data cache
    -a:  do not hide entries
```

The **-l** option displays detailed file information, such as type,

size, and protection. To reduce system overhead, LiteOS does not provide any time synchronization service, which is not needed by every application. Hence, there is no time information listed. A **ls -l** command returns the following:

```
$ ls -l
    Name    Type    Size    Protection
    usrfile file    100     rwxrwxrwx
    usrdir  dir     ---     rwxrwx---
```

In this example, there are two files in the current directory (a directory is also a file): **usrfile** and **usrdir**. LiteOS enforces a simple multilevel access control scheme. All users are classified into three levels, from 0 to 2, and 2 is the highest level. Each level is represented by three bits, stored on sensor nodes. For instance, the **usrdir** directory can be read or written by users with levels 2 and 3. The **chmod** command can be used to change file permissions.

Once sensor nodes are mounted, a user uses the above commands to navigate the different directories (nodes) as if they are local. The base station PC also has directories, such as drives **C** and **D**. Some common tasks can be greatly simplified. For example, by using the **cp** command, a user can either copy a file from the base to a node to achieve wireless download, or from a node to the base to retrieve data results [1]. The remaining file operation commands are intuitive. Since LiteFS supports a hierarchical file system, it provides **mkdir**, **rm** and **cd** commands.

Process Operation Commands: LiteOS has a multithreaded kernel to run multiple applications concurrently. When an application is first loaded, it is executed as a thread that can spawn new threads. LiteShell provides four commands to control thread behavior: **install**, **uninstall**, **ps**, and **kill**. We illustrate these commands through an application called **Blink**, which blinks LEDs periodically. Suppose that this application has been compiled into a binary file called Blink.lhex[2], and is located under the C drive of the base station. To install it on a node named **node101** (that maps to a directory with the same name) in a sensor network named **sn01**, the user types the following commands:

```
$ cp /c/Blink.lhex /sn01/node101/apps/Blink.lhex
    Copy complete
$ cd /sn01/node101
$ install ./apps/Blink.lhex
    File Blink.ihex installed under bin directory
$ ./bin/Blink.lhex
    Blink.lhex started
$ ps
    Name   PID
    Blink  1
```

As illustrated in this example, LiteOS installs applications into a directory called **bin** by default. The user starts an application by explicitly typing its name, and views current threads by using the **ps** command. Compared to Unix, the use of **install** may be debatable: in Unix, an application can be directly started by invoking its name, without a separate installation phase. While that design simplifies user operations, we provide the **install** command based on the special hardware of MicaZ: the MCU of MicaZ, Atmega128, follows the Harvard architecture, which provides a separate program space (flash) from its data space (RAM). Only instructions that have been programmed into the program space can be executed. Hence, LiteOS has to reprogram part of the flash in the application installation phase by using the SPM instruction of Atmega128. In addition, the installation procedure also involves a binary translation phase that dynamically rewrites parts of the binary, to avoid memory access conflicts of multiple applications. These translation details are described in Section 2.4. Above all, the installation procedure on MicaZ is more complicated and costly. If no separate **install** command is supported, when an application is repeatedly invoked, it needs to be installed multiple times. To reduce the installation cost, we allow the user to explicitly control which programs are going to reside in the program flash.

After an application is installed, it starts running when invoked by the user, as illustrated earlier. Because Blink has only one thread, the **ps** command returns one single thread. Finally, the **kill** command terminates threads, and the **uninstall** command removes an application from the program space. (The application binary is not really erased; only its occupied program space is marked available.)

Loop Command, Pipes, and Regular Expressions: Even though LiteOS is a node-based operating system, we find it convenient to support certain group operations. For example, prior to deployment, all nodes are usually kept together for programming and testing. Operations in this phase are usually group-based by nature, such as "install an application on all nodes". We have implemented two such group mechanisms, the **foreach** command and the **$** variable. When used together with regular expressions and pipes, these two commands loop over a set of nodes (directories) to perform group operations. We illustrate their usage through two examples.

Suppose that we have ten nodes labeled **201**, **202**, up to **210**. We have installed and started an application **Tracking.lhex** on each of them (located in the directory **bin**, and having a PID of 1), and have copied another application **Report.lhex** to each node (in the directory **apps**). To retask each node with the **Report.lhex** application, the user types the following command:

```
$ echo [201-210] | foreach $ \
  {kill 1; uninstall $/bin/Tracking.lhex;\
  install $/apps/Report.lhex; $/bin/Report.lhex}
```

In this example, the **echo** command generates a list of directories using a regular expression. Instead of being displayed by the terminal, these directories are provided to the variable **$** through a pipe (spaces between strings are used as separators), which is in turn used by the **foreach** command. As its name suggests, **foreach** loops over these directories, terminates and uninstalls the **Tracking** application, and installs/invokes the **Report** application.

In the next example, the **$** variable is concatenated with other strings to generate customized commands. The scenario is as follows. Suppose that the **report** application writes data results into a local file called **data.txt** on each node (in its root directory). To retrieve such data, the user types the **cp** command as follows:

[1]The implementation of copy requires that this file is not concurrently opened by applications for writing. Otherwise, the copy operation will return a failure.

[2]LiteOS uses a revised version of the Intel hex format, called lhex, to store binary applications. lhex stands for LitOS Hex.

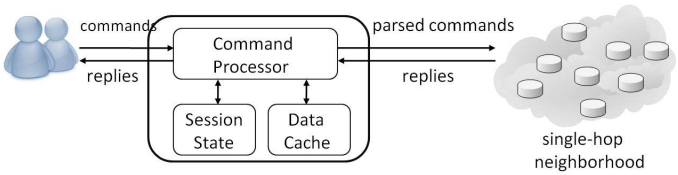


Figure 2. LiteShell Implementation

```
$ echo [201-210] | foreach $ \
  cp $/data.txt /c/sensordata/$.txt
```

Here, the command **cp $/data.txt /c/sensordata/$.txt** customizes the destination file for each node. Because normal file names only allow letters, digits, underscores(_), and dots(.), LiteShell recognizes that **$.txt** needs to be translated before execution. Hence, there will be no confusion.

Environment Commands: The next four commands support environment management: **history** for displaying previously used commands, **who** for showing the current user, **man** for command references, and **echo** for displaying strings. The semantics of these commands are similar to their Unix counterparts.

Security Commands: Finally, three commands, **login**, **logout**, and **passwd** are provided to support user authentication. When first connected to a base station, the user is asked to login by providing a valid username/password pair. LiteShell calculates the MD5 digest of this pair, and broadcasts it to the one-hop neighborhood. Any node that matches this digest will guarantee access rights to the user. Because calculating MD5 is computationally intensive, when username/password pairs were first created, nodes store the MD5 digest of such pairs instead of their values, which were calculated and broadcast by the base station. To bootstrap the whole procedure, an *administrator* username/password pair is stored when LiteOS is first installed, which is used to create new user accounts.

One potential security risk of the basic LiteShell is that it cannot effectively prevent replay attacks. Further, to reduce response time, all messages (other than the login procedure) between the base and the nodes are currently not encrypted. This is consistent with our default assumption of trust, which we focus on in this paper. A secure version of LiteOS is the topic of another publication.

Implementation of LiteShell: Figure 2 illustrates the implementation of LiteShell. Its command processor interprets user commands into internal forms, communicates with the sensor network, and replies to the user. We make two design choices. First, the sensor nodes are stateless. All state information regarding user operations, such as the current working directory, is maintained by the shell, while the sensor nodes only respond to interpreted commands using a simple ACK-based communication model. This design choice is motivated by the resource constraints of sensor nodes. It helps reduce the code footprint on the nodes as well as the communication overhead. Second, to reduce response time, LiteShell implements an internal data cache. For example, if the user types two consecutive **ls** commands, the replies from the network are likely to be identical. Hence, LiteShell relies on the cache to handle such "redundant" user commands without communicating again with the sensor network. Usually, the storage on the base station PC is sufficiently large.

Table 2. Shell Commands Classification

Command List	
Local Commands	pwd, echo, who, man, history
Network Commands	cp, mv, rm, mkdir, touch, chmod du, ps, kill, install, uninstall, login, logout
Both	ls, cd

Hence, we implement a "cache everything" policy, where cached information is assumed never to expire in the same session. To ensure updated information, on the other hand, we implement an **-u**(update) option for most interactive commands, such as **ls**, to forcefully retrieve up-to-date information from the network and update the cache.

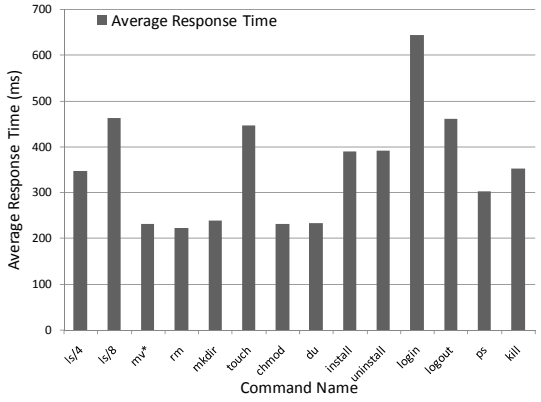
As an additional note, not every command can be successfully completed. For example, when the user installs an application, if the remaining program space and RAM of a node are not sufficient for a binary to be loaded, the node responds with an error message. We build LiteShell to robustly handle such scenarios.

LiteShell can be used in multiple ways. First, prior to deployment, all nodes are usually kept together, within one hop radio range, where LiteShell helps the user to develop and test applications. Second, during experiments, a user can interact with nodes by mounting a data mule, or by physically approaching sensor nodes of interest. As long as there is only one base station in a neighborhood, conflicts and inconsistencies will not arise. Observe that multiple users may have different views of the network depending on mounted nodes and freshness of cache (which is similar to NFS semantics).

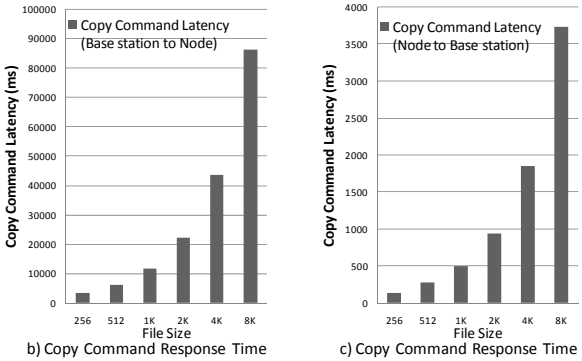
LiteShell Performance Evaluation: We now focus on evaluating the measured average response time of LiteShell as its main performance metric. In the following experiments, we classify shell commands into three categories: those executed locally, those communicating with nodes, and those that do both, as listed in Table 2.

We find that, while commands in the first category are typically finished within one millisecond, commands of the later two categories take much longer to complete. We measured the average response time of these commands by using a script to execute each command 100 times, and recorded the average response time of each command. The results are shown in Figure 3(a). In particular, the *mv** command only moves files on the same node. We observe that the response time of every command is less than one second. While we acknowledge that performance can possibly be further optimized by fine-tuning the communication protocol, the current response time is already quite acceptable. For example, our interactive performance is generally better than common experiences in Web browsing.

One special command that deserves further experiments is the **cp** command, whose response time depends on the size of the file to be transferred over the air. We experimented with different file sizes, and list the results in Figure 3(b) and 3(c). As illustrated, the time to receive data files is much shorter than the time to send data files, partially because in the later case, the base station has to deliver packets first to a node connected through UART, which in turn broadcasts such packets over the radio. Sending packets via UART turns out to be slower, which explains the performance differences. One way to solve this problem is to connect a CC2420 radio



a) Command Response Time (ls/4 means listing a directory with four children nodes; ls/8 means listing a directory with eight children nodes; mv* is for node-wise move only)



b) Copy Command Response Time (Base station to Sensor node)

c) Copy Command Response Time (Sensor node to Base Station)

Figure 3. Average Command Response Time of LiteShell

directly to the base station to improve performance[3].

2.3 LiteFS Subsystem

LiteFS Interfaces: We now describe the LiteFS subsystem. The interfaces of LiteFS resemble Unix closely, providing support for both file and directory operations. LiteFS is not the first effort to provide a file system for sensor networks. Previous well-known efforts include both MicaZ-class implementations, such as MatchBox [11] and ELF [8], and flash-based systems, where additional flash is plugged to sensor nodes, such as Capsule [22]. We do not adopt the former because their interfaces do not match our goals: Matchbox and ELF do not support hierarchical file organizations, only providing basic abstractions for file operations such as reading and writing. On the other hand, we do not require external flash attachment so that users with off-the-shelf MicaZ nodes can directly use LiteOS. The APIs of LiteFS are listed in Table 3.

While most of these APIs resemble those defined in "stdio.h" in C, some of them are customized for sensor networks. For instance, two functions, **fcheckEEPROM** and **fcheckFlash**, are unique in that they return the available space of EEPROM and the serial flash, respectively. Another feature of LiteFS is that it supports simple search-by-filename using the **fsearch** API, where all files whose names

[3]We also tested the response time of copying files between nodes. Not surprisingly, its delay is no different from Figure 3(c).

Table 3. LiteFS API List

API Usage	API Interface
Open file	FILE* fopen(const char *pathname, const char *mode);
Close file	int fclose(FILE *fp);
Seek file	int fseek (FILE *fp, int offset, int position);
Test file/directory	int fexist(char *pathname);
Create directory file	int fcreatedir(char *pathname);
Delete file/directory	int fdelete(char *pathname);
Read from file	int fread(FILE *fp, void *buffer, int nBytes);
Write to file	int fwrite(FILE *fp, void *buffer, int nBytes);
Move file/directory	int fmove(char *source, char *target);
Copy file/directory	int fcopy(char *source, char *target);
Format file system	void formatSystem();
Change current directory	void fchangedir(char *path);
Get current directory	void fcurrentdir(char *buffer, int size);
Check EEPROM Usage	int fcheckEEPROM();
Check Flash Usage	int fcheckFlash();
Search by name	void fsearch(char *addrlist, int *size, char *string);
Get file/directory info	void finfonode(char *buffer, int addr);

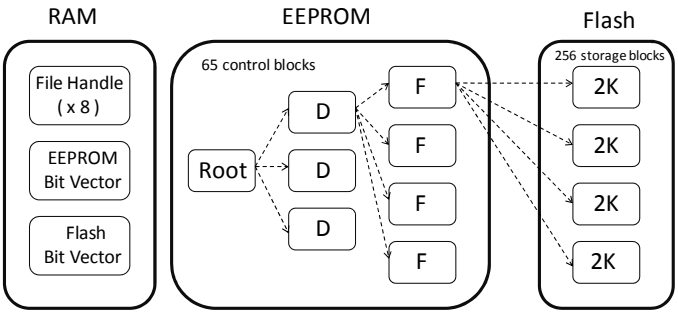


Figure 4. LiteFS Architecture

match a query string are returned. These APIs can be exploited in two ways; either by using shell commands interactively, or by using the LiteFS class provided by LiteC++ which exports this API for application development.

Design Choices: MicaZ nodes have two non-volatile storage space: EEPROM (4K bytes) and the serial flash (512K bytes). While EEPROM can be updated at the byte level, the serial flash has to be updated at the page level, with each page occupying 264 bytes. They also differ in access time. For EEPROM, reading one byte takes 1 CPU cycle (and the CPU halts for 4 cycles after that), while writing one byte takes 8848 cycles. For the serial flash, reading and writing are controlled at the page level using two 264-byte SRAM buffers. Reading a page into this buffer takes less than 250 microseconds, but writing a page takes 14-20ms.

Given these hardware characteristics, we conclude that the serial flash is suitable for sequential data update, while EEPROM is more ideal for data updates at a smaller granularity. We use EEPROM to keep LiteFS directory information. An advantage of this design is that in the future, if the 512K flash is replaced by other flash with much larger capacity, the design of LiteFS can be easily modified to support new flash memory by only changing flash-related functions, while reusing its directory operation implementations.

Implementation of LiteFS: Figure 4 shows the architecture of LiteFS, which is partitioned into three parts. It (i) uses RAM to keep opened files and the allocation information of EEPROM and the serial flash, (ii) uses EEPROM to keep hierarchical directory information, and (iii) uses the serial flash to store files. Just like Unix, files in LiteFS represent different entities, such as data, application binaries, and de-

vice drivers. A variety of device driver files, including radio, sensor, and LED files, are supported. Their read/write operations are mapped to real hardware operations, e.g., writing a message to the radio file (either through the shell by a user or through a system call by an application) maps to broadcasting this message.

In RAM, LiteFS has room for eight (flexible) file handles, with each handle occupying eight bytes. Hence, at most eight files can be opened simultaneously. LiteFS uses two bit vectors to keep track of EEPROM/flash allocation, one with 8 bytes for EEPROM, the other with 32 bytes for the serial flash. A total of 104 bytes of RAM are used to support these bit vectors.

In EEPROM, each file is represented as a 32-byte control block. LiteFS currently uses 2080 bytes (flexible) of the 4096 bytes available in EEPROM to store hierarchical directories, while the remaining EEPROM is available for other needs. These 2080 bytes are partitioned into 65 blocks. The first block is the root block, which is initialized every time the file system is formatted. The other 64 blocks are left for the application needs.

There are four types of control blocks, file, directory, binary application, and device driver. We follow three design choices in these data structures. First, the maximal length of a file name is 12 bytes. Therefore, the *Eight Dot Three* naming system is supported. Second, a single control block addresses at most ten flash pages. Each page holds 2K bytes of data (or 8 physical flash pages). If a file occupies more than 20K bytes, LiteFS allocates another control block for this file, and stores the address of the new block in the *next* field of the old one. Third, all control blocks reside in EEPROM rather than in RAM.

Wear levelling Optimization: Repeated erase-writes to the same location in flash or EEPROM quickly exhaust its lifetime. Hence, it is important that the erase-write locations spread evenly. This process is usually referred to as *wear levelling*. We implemented two techniques for this purpose, one in EEPROM and the other in the serial flash. Compared to previous work on wear-levelling, while several approaches have been proposed for the serial flash [8, 22], wear levelling optimizations for EEPROM have largely been neglected. Besides, some modern flash hardware already implements built-in wear levelling. Therefore, our following descriptions focus on the EEPROM part. For the serial flash, we implement a page-age (stored in page metadata) based scheme, where the driver picks a relatively young page by comparing several page candidates.

To avoid hot spots in EEPROM, LiteFS performs periodic cyclic permutations of all control blocks except the root block (which is not updated except immediately after each permutation). To maintain the relationship between different blocks, LiteFS uses relative addresses in control blocks, rather than absolute addresses. For example, a control block **Parent** (at address A) keeps the address of another block **File** (at address B) as its offset $(B-A+64) \pmod{64}$. During permutations, the relative distances between blocks remain constant. Only the root block needs to modify its pointers to its child blocks. This is not a problem because the root only maintains one pointer to the real root, therefore, its lifetime is sufficiently long. The cost of one permutation constitutes writing 2048 bytes to EEPROM, and takes around 2 seconds. To trigger such permutations, LiteFS maintains a counter (two bytes) for the number of write operations. In the worst case, all write operations map to a same location. To avoid hot spots under such an extreme scenario, LiteFS triggers a permutation every 25000 writes.

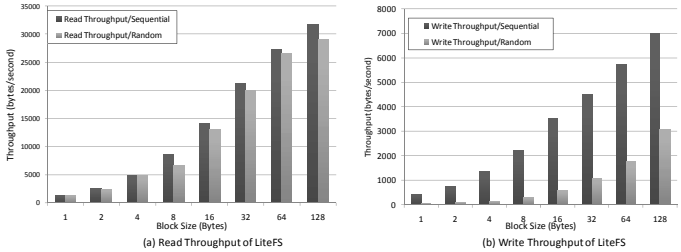


Figure 5. LiteFS Throughput

LiteFS Performance: Due to space limitations, we focus on throughput performance of LiteFS, as shown in Figure 5. In this experiment, we wrote a simple application that repeatedly called the **fread** and **fwrite** APIs, and measured the time to complete I/O operations. By default, we used one file with 128K bytes, and experimented with different block sizes. LiteFS allows modifying the middle of a file with the help of the **fseek** API. Therefore, we performed both sequential and random reads and writes.

We observe a reading throughput up to 30 KBytes/s, much higher than the writing throughput. Another observation is that when the block size is small, the random writing throughput degrades exponentially. The reason is that it is very likely for a page-erase operation to occur for every writing operation, because the file pointer has been modified to a random position. This is a special type of thrashing, where the cache of the serial flash (264-byte SRAM) experiences excessive misses. Because of this phenomenon, the random throughput of LiteFS is lower, compared to throughput results reported in the ELF file system [8], which followed a different setting to test random throughput, where the file pointer was modified only once and no thrashing occurred.

2.4 LiteOS Kernel Subsystem

Threads and Events: Threads provide basic parallelism in modern computers. In sensor networks, there are design tradeoffs between using threads and events. Both approaches have been implemented in previous research efforts. TinyOS, SOS, and Contiki [4] are representatives of using events, while Mantis and TinyThreads choose threads.

The kernel subsystem of LiteOS takes the thread approach. We implement two different scheduling policies: priority-based scheduling, where the thread with the highest priority always gets dispatched, and round-robin scheduling, where each thread gets a slice of CPU, whose length is decided by its priority. By default, eight threads are supported. Unlike TinyThread, which implements cooperative scheduling, the LiteOS kernel takes back the CPU every 0.1 seconds. Of course, the correct execution of this design choice relies

[4]Contiki supports a special type of lightweight threads called protothreads [10], which are different from conventional threads. Hence, we still consider Contiki as event-based.

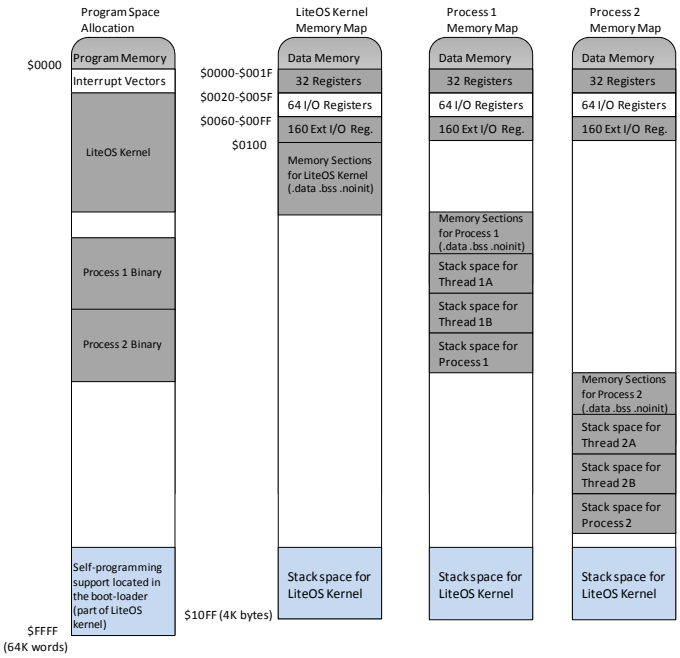


Figure 6. LiteOS Memory Organization

on our earlier assumption that user threads are not malicious, and will not disable interrupts to enter an infinite loop.

In response to user commands, the kernel allows dynamic loading of user threads. To facilitate this, it maintains a map of system resource allocation, including both its program flash and RAM. To dispatch a thread, it copies thread information, such as its entry address and priority, into a free control block. When a thread terminates, it frees allocated resources for this thread, by marking both the occupied program flash and RAM as available. It also forcefully closes previously opened file pointers by this thread, if there are any, in case this thread forgets. Because LiteOS does not support dynamic memory allocation, all resource allocation and collection are static.

While resource management in LiteOS is relatively simple to implement, there are two major challenges that directly affect user experiences using LiteOS threads. The first is to support dynamic loading and un-loading of user applications, running them natively, with a very low overhead. The second is to provide a clear boundary between the kernel and user applications, and to maintain compatibility during system updates. These are described next together with related work.

Previous work: Dynamic loading is taken for granted in almost every modern operating system. Conventionally, dynamic loading is closely related to virtual memory: to avoid conflicts between memory accesses of multiple processes, the operating system maps them to different physical addresses with the help of page tables. While virtual memory support in MicaZ-class sensor networks has been implemented [20, 14], we observe that such efforts unanimously slow down program execution, because of the lack of hardware support for page tables and the limited computation power of the CPU. While the direction of support dynamic loading with the help of virtual memory may still be possible, several previous efforts have tried alternative approaches to implement dynamic loading in sensor networks.

Representative efforts to provide dynamic loading without virtual memory include TinyOS (using XNP), SOS, and Contiki. In XNP, a boot loader provides loading service by reprogramming part of the flash and jumping to its entry point to dispatch it. While this approach is practical, it cannot load more than one application, due to potential conflicts in memory accesses. SOS, on the other hand, proposes to use modules. Its key idea is that if modules only use relative addresses rather than absolute addresses, they are relocatable. However, compiling such relocatable code has size limitations: the compiler for the AVR platform (MicaZ) only supports binaries up to 4K bytes. Such limitations cause scalability problems for large applications. Contiki takes yet another approach, where it parses ELF files to patch all unsolved symbolic links, and to perform binary relocation with the help of a symbol table in its kernel. A critical difference between Contiki and LiteOS, however, is that Contiki is an event-based operating system. Its dynamic linking and relocation only apply patches to symbolic links of kernel variables and function interfaces in single-threaded application binaries. LiteOS, on the other hand, allows multithreaded operations, where each thread owns separate memory segments and stacks. The approach of patching ELF files is no longer sufficient, and our experiments show that such an approach cannot be ported to the MicaZ platform due to memory constraints.

Goals: Our primary goal is to support dynamic loading and *native* execution of multiple multithreaded applications with *minimal* overhead. We emphasize the words *native* and *minimal*, because if the kernel introduces excessive overhead in either execution speed or memory consumption, it is less likely to be useful.

Our second goal is to achieve software compatibility. The LiteOS kernel is likely to evolve in the next few years, providing new functionalities and performance optimizations. Just like we can still run old applications on new versions of mainstream PC operating systems, we do not want recompilations of old user applications with every small revision to the LiteOS kernel.

Approach Overview: Our approach leverages both the LiteOS kernel and the LiteC++ compiler. It remotely echoes the *differential patching* idea [28, 17] in application distributions over networks. In that idea, to update an application, the user compiles a list of differences between the old and new versions of the application, and only sends such differences as an update, so that communication cost can be reduced. However, this idea does not directly work in LiteOS, because it requires (unreasonably) that the user has to obtain the source code of all applications.

Our approach differs from differential patching in that instead of obtaining differential information separately, it encodes such information into application binaries, by fitting differential patches with mathematical models, and distributing these models together with binary images. With such models, the kernel is able to rewrite part of the binary image for relocation. With appropriate mathematical models, such relocations guarantee that no memory access conflicts

will arise. Figure 6 illustrates such an ideal case, where six threads (besides the kernel) are executed concurrently. Note that the memory sections of different threads do not overlap, and threads are executed natively.

Our development of mathematical models for differential patches are driven by practical observations. The models we construct, which are linear, are based on analyzing binary images generated by the specific compiling environment of LiteC++, which implements whole-program optimizations using GCC. By being customized, these models are easier to be implemented on MicaZ motes with very low overhead, which is critical for LiteOS performance. For compilers other than GCC, or other versions of GCC, the model details, such as the types of instructions that need to be translated, are likely to change. However, generalizing the models to support compiling environments other than LiteC++ is out of the scope of this paper.

Observations on Differential Patches of Binary Image: We start presenting the details of our approach by observations on binary image instructions. When the kernel loads a binary to different memory locations, the potential factors that affect the image can be written as a vector $(\mathbb{S}, \mathbb{M}, \mathbb{T})$, where $\mathbb{S}$ is the start address of the binary executable in the program flash, $\mathbb{M}$ the start address of allocated memory, and $\mathbb{T}$ the stack top. All three parameters can be set in GCC and other common compilers. Observe that the difference between $\mathbb{T}$ and $\mathbb{M}$ is the actual memory space allocated for this executable, whose minimum requirement can be statically decided using a technique called stack analysis (as long as the application has no recursive functions). Several such tools are available [27, 23]. Therefore, we assume that the stack top has been statically decided [5].

Obviously, if $(\mathbb{S}, \mathbb{M}, \mathbb{T})$ is static and known in advance, the user just needs to compile the source code once. Since they are volatile, we have to find a way to encode their impact. Experimentally, we observe that as the vector changes, it only affects the immediate operands of a few instructions. More specifically, of the 114 instructions provided by the Atmega128 processor (on MicaZ), under the LiteC++ compiling environment, only six instructions are affected, listed in Figure 7. We call them *differential instructions*.

Mathematical Models of Address Translation: We use an example to illustrate how we model address translation, where we sample the light sensor for 100 times, at a frequency of once per second, and write readings into a local file in LiteFS, so that the user may extract such data using LiteShell. Using the LiteC++ compiler, this application compiles to 358 instructions consuming 790 bytes of binary. We compiled it with three $\mathbb{M}$ settings, $0x500$, $0x504$, and $0x600$, and compared the compiled code in the assembly level. We found a total of 12 different instructions out of these 358 instructions. Semantically, they are grouped into three difference clusters, shown in Figure 8.

We list a total of 14 instructions, including two instructions (⑤ and ⑩) that are not differential, but are useful for our following explanations. The underlined numbers highlight their differences. Our primary observation over these differential patterns is that the values taken by immediate operands in each differential instruction are closely related to the values of $\mathbb{M}$. For instance, the immediate operands in instructions ③ and ② collectively represent a 16-bit number with a value of $\mathbb{M}$, while instruction ① takes the value of $\mathbb{M}$ shifted eight bits in the right direction.

[5]This task is actually performed by the LiteC++ compiler described in the next section. We integrate a revised stack analyzer from TinyThread into the compiler, so that it automatically allocates sufficient stack space for threads.

Instruction	Semantics	Syntax
LDI	Load immediate into register	LDI Rd, K (16 ≤ d ≤ 31, 0 ≤ K ≤ 255)
LDS	Load direct from data space to register	LDS Rd, K (0 ≤ d ≤ 31, 0 ≤ K ≤ 65535)
STS	Store direct to data space from register	STS K, Rr (0 ≤ r ≤ 31, 0 ≤ K ≤ 65535)
JMP	Jump	JMP K, 0 ≤ K ≤ 64K
CALL	Long call to a subroutine	CALL K, 0 ≤ K ≤ 64K
CPI	Compare register with a constant	CPI Rd, K (16 ≤ d ≤ 31, 0 ≤ K ≤ 255)

Figure 7. Differential Instructions

Section	Address	M = 0x500	M = 0x504	M = 0x600
_do_copy _data	① 19098	ldi r17, 0x05	ldi r17, 0x05	ldi r17, 0x06
	② 1909a	ldi r26, 0x00	ldi r26, 0x04	ldi r26, 0x00
	③ 1909c	ldi r27, 0x05	ldi r27, 0x05	ldi r27, 0x06
	④ 190ac	cpi r26, 0x0C	cpi r26, 0x10	cpi r26, 0x0C
	⑤ 190ae	cpc r27, r17	cpc r27, r17	cpc r27, r17
_do_clear _bss	⑥ 190b2	ldi r17, 0x05	ldi r17, 0x05	ldi r17, 0x06
	⑦ 190b4	ldi r26, 0x0C	ldi r26, 0x10	ldi r26, 0x0C
	⑧ 190b6	ldi r27, 0x05	ldi r27, 0x05	ldi r27, 0x06
	⑨ 190bc	cpi r26, 0x14	cpi r26, 0x18	cpi r26, 0x14
	⑩ 190be	cpc r27, r17	cpc r27, r17	cpc r27, r17
main	⑪ 190d2	ldi r22, 0x00	ldi r22, 0x04	ldi r22, 0x00
	⑫ 190d4	ldi r23, 0x05	ldi r23, 0x05	ldi r23, 0x06
	⑬ 190d6	ldi r24, 0x02	ldi r24, 0x06	ldi r24, 0x02
	⑭ 190d8	ldi r25, 0x05	ldi r25, 0x05	ldi r25, 0x06

Figure 8. Binary Image Difference Example

Before illustrating what we can do, we start with what we cannot do. Generally, it is almost impossible for us to predict what code the compiler will produce, or the semantic relationships between instructions. For example, in Figure 8, instructions ① to ⑤ belong to the data segment initialization section. Similarly, instructions ⑥ to ⑩ belong to a section that clears the bss segment for storing global variables. Such kind of implicit relationships only emerge with application specific analysis, but are far beyond the capability of mathematical modeling.

Because of these difficulties, we take an application independent approach, shown in Figure 9. We analyze the differences between compiled images under different memory settings, generate linear models to fit such differences, and test the models with training data. The models are based on the observation that for differential instructions, their values under different $(\mathbb{S}, \mathbb{M}, \mathbb{T})$ vectors demonstrate a set of linear displacements. That is, if V is the value of an immediate operand in a differential instruction, it can be written as $\mathbb{V} = f(\mathbb{S}, \mathbb{M}, \mathbb{T})$, where f is function that exhibits linearity properties. This follows directly from the format of machine code for various addressing modes. In the previous exam-

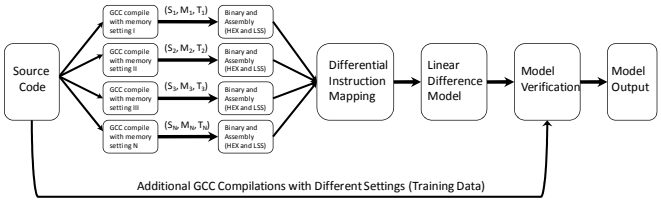


Figure 9. Mathematical Model Generation

ple, except non-differential instructions ⑤ and ⑩ , f can be written as follows:

$$\mathbb{V} = ((a\mathbb{M} + b) >> L) << R \quad (1)$$

In this equation, a, b, L and R are undecided variables, $>>$ is the right-shifting operation, and $<<$ is the opposite. To solve a, b, L and R, we usually need four different compiler settings. After solving these variables for the previous example, the f function of ① is determined as $\mathbb{M} >> 8$ ($a = 1, b = 0, L = 8$, and $R = 0$).

While our discussions so far are only about $\mathbb{M}$, the same approach applies to $\mathbb{S}$ and $\mathbb{T}$, though we observe different variations of the f expressions. The final form of f is always a superposition of different linear equations for $\mathbb{S}$, $\mathbb{M}$, and $\mathbb{T}$, and is sufficiently simple to be encoded into binary images. Due to space limitations, we do not elaborate on $\mathbb{S}$ and $\mathbb{T}$ displacement models. To ensure that our approach works correctly, we always generate *training* binary images, serving as verifications of the mathematical models.

Implementation Details: We have implemented the above binary translation procedure as one step in the LiteC++ compiler. One problem is how to practically integrate such models into binary executables. We modified the Intel HEX format to encode such models, and call our format the LiteOS HEX format (with a suffix of .lhex). Figure 10 shows the differences between these two formats. Similar to the Intel HEX format, LiteOS HEX format is record-based, and uses ASCII characters to represent binary data. Different from the Intel format, LiteOS HEX removes the *Address* field in the records, since binary codes in LiteOS are relocatable. Another difference is that LiteOS introduces three new types of records: explanation records, for storing essential information of the binary executable, such as its size and memory consumption; equation specification records, for keeping mathematical model information; and differential instruction records. LiteOS Hex files are automatically generated by the LiteC++ compiler and are used to reprogram sensor networks.

System Calls and Software Compatibility: To distribute applications using the LiteOS HEX format, one important problem is software compatibility. To reduce redundancy, LiteOS provides system resources, such as LiteFS, drivers, and thread services for user applications. If the LiteOS kernel is modified, the addresses of its internal data structures will change. If not handled carefully, such updated kernels no longer support binaries compiled for older versions of LiteOS. While this problem can be detected easily (through version numbers), we are interested in how to avoid such incompatibilities.

We introduce lightweight system calls to address compat-

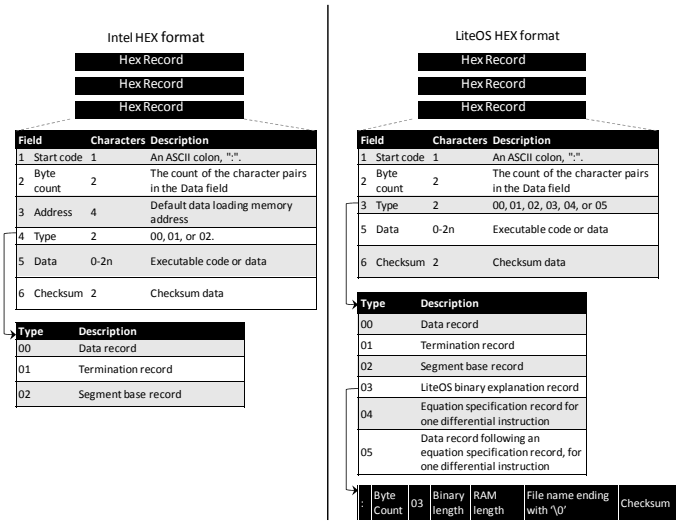


Figure 10. Comparison between Intel HEX and LiteOS HEX

ibility. Because the MicaZ CPU does not support soft interrupts or traps, our implementation is based on revised *callgates*, a special type of function pointers. These callgates are the *only* access points through which user applications access system resources. Therefore, they implement a strict separation between the kernel and user applications. As long as the system calls remain supported by future versions of LiteOS, user binaries do not need to be recompiled.

At the system kernel side, each system call gate takes 4 bytes, with 1024 bytes of program space allocated for at most 256 system calls. So far, we have used 52 of them, and this number is expandable. Compared to directly invoking kernel functions, each system call adds 5 instructions (10 CPU cycles), a low overhead to be supported on MicaZ. At the user side, interactions with system calls are encapsulated into LiteC++ classes. For example, part of the **send** function provided by the **Radio** class is implemented as follows:

```
   //Mutex to control Radio access
1   mutex* msend;
2   thread** current_thread;
   //Get radio mutex system call
3   msend = getRadioMutexAddress();
   //Get current thread address system call
4   current_thread = getCurrentThread();
   //Try to lock mutex or block
5   Mutex.lock(msend);
   //Set up radio send information
6   (*current_thread)->data.radiostate.port = port;
7   (*current_thread)->data.radiostate.address = address;
8   (*current_thread)->data.radiostate.length = length;
9   (*current_thread)->data.radiostate.msg = msg;
10  {
11   void (*radiosendfp)() = (void (*)(void))RADIO_SEND_SYSCALL_GATE;
     //direct system call to the radio send function
12   radiosendfp();
13  }
     //Set the timeout as 10 millisecond
   //May be awake before this timer ends
   //if the radio finishes before 10 milliseconds elapse
   //sleep thread system call
14  Thread.sleepThread(10);
15  Mutex.unlock(msend);
```

In this example, the **Radio.send** function in the user thread obtains the radio mutex (if it fails, the thread blocks until the mutex is released by another thread), sets up its data

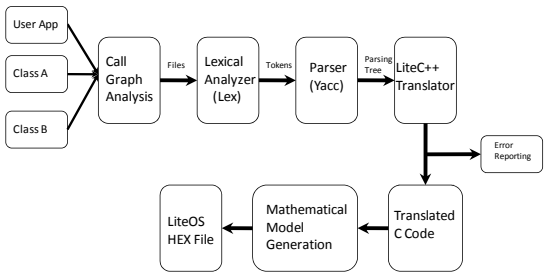


Figure 11. LiteC++ Compiler Architecture

structure, invokes a system call (at line 12), and sleeps for 10ms. This thread is either woken up by the kernel if the radio action finishes successfully within 10ms, or, under very rare cases, woken up by the sleep timer if the radio driver fails.

3 The LiteC++ Environment

We develop LiteC++, an object-oriented programming language that features a subset of C++ syntax (with one keyword, **import**, from Java). We choose C++ for its popularity: most programmers already know it. Hence, we reduce the learning difficulty. Constrained by hardware resources available on MicaZ-class sensor nodes, we only implement a subset of C++ features, including class and type polymorphism, for information hiding and encapsulation. More advanced, but less useful language features for sensor network applications, such as templates, are not implemented. Consistent with the LiteOS kernel, LiteC++ is thread-based, and interacts with the kernel only through system calls.

To reduce development cost, LiteC++ provides a suite of nested class abstractions with different levels of granularity, which we refer to as *LiteC++ Base Classes*. Instead of assuming a specific application model, they allow different "programming models" to be mixed. For example, a class that supports EnviroTrack style functionality can be used to implement cross-node middleware services, while another class can represent Abstract Regions [29] or Hood-style abstractions [30]. The reliance on objects as the fundamental abstraction mechanism does not dictate their semantics, which are decided by developers and published to the broad research community to reduce development overhead.

This section is organized as follows. We first present the design of LiteC++, followed by details of its class library, and finally, its performance evaluation through 21 benchmark applications.

The following basic principles underlie LiteC++'s design.

LiteC++ as a static language and a subset of C++: LiteC++ is implemented using a subset of C++ grammar rules (a superset of C). Different from C++, LiteC++ does not implement dynamic memory. Instead, all classes are static, and translated into C by the LiteC++ compiler. Put another way, we only allow class (static) methods in programs, making the call graph fully known at compile time. Such static nature of LiteC++ is necessary for us to implement whole-program analysis and optimization to reduce code footprint.

LiteC++ integrates LiteOS system calls: The classes in LiteC++ reflect the design of LiteOS, by using system calls to support common operations. For example, sending out a packet over the radio is needed by many applications. By implementing this service as a system call and coordinating its access by multiple applications using synchronization primitives such as mutex, LiteC++ avoids functional redundancies and reduces code footprint.

Figure 11 presents the architecture of the LiteC++ compiler. It uses Lex and Yacc in early stages, and implements a type-checking subsystem for reporting errors before invoking the GCC compiler to compile translated C code. Finally, the generated binary images from GCC are analyzed using the aforementioned mathematical models to create relocatable LiteOS Hex files.

3.1 Grammar Features

Classes: A LiteC++ application is built by compiling multiple classes. Unlike NesC [13], which relies on wiring to specify module interactions, LiteC++ enforces simple rules to avoid confusions: each class file must have the same name as the class it implements. For instance, **Radio.cpp** implements the **Radio** class, and the compiler uses file names to locate classes. Hence, no two classes should have the same name in the LiteC++ programming environment. (If there are, the compiler will generate warnings.) Just like C++, a class could also define public or private variables/functions, and only public variables/functions are visible. Classes can invoke system calls, as demonstrated by the **Led** class:

```
//class Led for Led operations, partial listing.
public class Led{
  //toggle the red led.
  public void redToggle() {
  //use function pointer to call the led system call
  void (*redtogglefp)() = (void (*)(void))RED_TOGGLE_SYSCALL_GATE;
  redtogglefp();  }
}
```

A user application uses **import** to use a class. For instance, a sensing-and-sending application looks as follows:

```
1 import Led, Radio, Sensor, Thread;
//Main function
2 int main(){
3 int i,reading;
 //send out the reading 100 times with a frequency of roughly 1Hz
4 for (i=0;i<100;i++)
5 {
6  Led.redToggle();
   //Sensor class reads sensors using system calls
7  reading = Sensor.getLightSensor();
   //Radio class sends messages
8  Radio.send(i);
9  Radio.send("The reading is" + reading + "\n");  }
  //Thread class allows sleeping the current thread
10 Thread.sleep(1000);
11 }
12 return 0;  }
```

This example shows us several things. First, observe that LiteC++ follows a thread-based programming style, which has been previously explored in Mantis and TinyThread. This style of programming removes the **sendDone** event handler. Different from Mantis/TinyThread, however, LiteC++ is object-oriented, leading to more structured and easier-to-understand source code.

This example also illustrates function and operator overloading. Observe that **Radio.send()** takes different parameter signatures, first a two-byte integer, then a string. In fact, part of the **Radio** class is implemented as follows:

```
public class Radio{
  public int send(uint16_t  value) { ... }
```

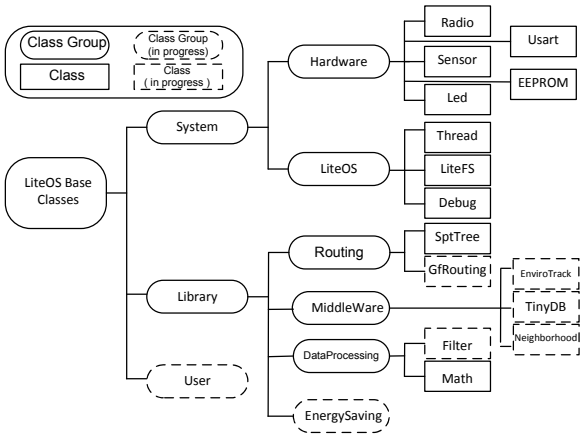


Figure 12. LiteOS Base Classes

Method	Semantics	Implementation
Thread Class		
sleep	Sleep for a period of time	System call using a timer in the kernel
wakeup	Wakeup a thread	Set the state of the thread to be ACTIVE
yield	Give up CPU immediately	Context switch
create	Create a new thread	The kernel sets up the thread information using an available thread control block
terminate	Removes a thread	The kernel marks the control block and the consumed resources of this thread as available
Radio Class		
send	Send out a message	System call, and the kernel sends the radio message
receive	Receive a message	System call, and current thread is suspended. It specifies a listening port. When a message for this port arrives, this thread resumes
Debug Class		
assert	Fault-tolerance checkpoint	Checks if a condition is true or not. If false, the current thread is terminated and the kernel logs an error
setLogFile	Set the output log file for the current thread	Creates a log file pointer for the current thread for writing log information
logData	Log traces of the program execution	Writes trace information to the output log file

Figure 13. Class API Examples

```
  public int send(uint8_t*  string){ ... }
  public int send(uint8_t*  string, uint8_t size){ ... }
  public int send(uint16_t  value,  uint8_t port)  { ... }
  public int send(uint8_t*  string, uint8_t port){ ... }
  public int send(uint8_t*  string, uint8_t size, uint8_t port){ ... }
}
```

Recall that LiteC++ translates C++ code into C. Because C does not provide polymorphism support, the above feature is implemented as follows. The LiteC++ compiler analyzes all type information, and rewrites overloaded function names using their parameter signatures. For instance, the **public int send(uint16_t value)** is rewritten to be **public int send_1_unsignedinteger(uint16_t value)**, where **1** stands for the number of parameters, and **unsignedinteger** stands for its parameter type signature. As long as overloaded functions have different parameter type signatures, no confusion will be introduced.

Another observation concerning the previous example is operator overloading, as illustrated in line **9** of the previous example. Note that the operator + connects strings to integers, where the integer is *promoted* into a string type in this case. This feature is also implemented by analyzing data types, and rewriting the source code using string operations.

3.2 LiteC++ Base Classes

Most software is not written from scratch. Instead, the development of modern software is usually supported with libraries. In LiteC++, we provide such a library for the user, called LiteOS Base Classes (LBC). In this section, we briefly introduce how it is organized, and give a short tour of several interesting classes, including **Radio**, **Thread** and **Debug**.

Figure 12 shows the architecture of LBC. Each rounded rectangle maps to a file directory, and each rectangle maps to a class file. For instance, the **Radio** class is implemented by the **Radio.cpp** file in the library. The whole architecture is broadly classified into **System** (for operating system related classes), **Library** (for standard libraries), and **User**, for extended user classes. Currently ten classes have been built and tested.

A partial list of APIs of three example classes, **Radio**, **Thread**, and **Debug**, is shown in Figure 13. Some of these functions are blocking. For example, if a user thread writes:

```
//block myself for a message
Radio.receive(MYPORT, msg, length);
```

then it is blocked until the message comes. Some other functions, such as those of the **Led** class, are non-blocking. While most classes are limited to operations on a single node, some encapsulate interactions between multiple nodes. Thus, the abstraction level is raised. One such network level class we implemented is the class **SptTree**. It has the following API interface:

```
public class SptTree
{
//set the current node as the root of
//a new shortest path tree with id
  public void setRoot(int id) {...}
//check if the current node is the root of
//a shortest path tree with id
  public bool isRoot(int id) {...}
//build the tree with id
  public void build(int id){...}
//send a message to the parent using a SPT tree
  public void sendToParent(...) {...}
//set callback function if the current node is a root
//and a message has been received. The callback function
//is called to further handle this message
  public void setCallBack(int id, void (*tp)(void)\\
  , uint8_t *length, uint8_t **address) {...}
}
```

As shown above, the class **SptTree** maintains its private data structures to keep multiple trees with different id numbers, creates threads to send and receive messages, and invokes callback functions to interact with other modules of an application. To the user, how a tree is implemented is irrelevant. It appears as if the class is local, although its implementation may span the entire sensor network.

3.3 Programming Model

We use *programming model* to refer to a developer's view of a programming environment. In TinyOS/NesC [13, 12], the view is such that an application is driven by events, which can optionally post long-term tasks to a FIFO scheduler. For example, a simple Blink application is driven by a timer fire event, whose event handler blinks the LED. A more complicated application typically needs state machines, because TinyOS does not directly support execution contexts. For instance, in writing reliable communication stacks, the developer sometimes wants to do the following: after send-

ing a packet, a node waits for a period $T_{timeout}$ (after which the previous packet is considered lost), or stop waiting if an acknowledgement is received. In the programming model of TinyOS, such a task is decomposed into two events: a timeout event, and a radio receive event. Because the order of these two events is not predictable, a developer introduces two states in the state machine. If the number of states grows large, handling state transitions usually becomes complicated. Such difficulties motivated several research efforts to simplify application development, such as the OSM model [19] and Abstract Regions [29].

A key observation that motivated the LiteOS programming model is regarding events. There are actually two types of events: those triggered by user applications, and those triggered by external factors. For instance, a **sendDone** event always follows a packet sending operation. A radio receive event, however, is triggered by external events, and is not predictable. Unlike previous work, which does not differentiate these two types of events, LiteOS treats them separately. Observe that the first type of events has already been handled by using threads, where a sendDone operation is no longer visible to a programmer. It is the second type of events that deserves special attention.

A typical second type event is the radio receive event. LiteOS provides two solutions to handle this event. The first is to create a new thread using the **Thread.create** system call, which blocks until the message arrives[6]. Return to the previous reliable communication example. Suppose that the application thread creates a child thread to listen to possible acknowledgements. If such a message is received before $T_{timeout}$, the child thread wakes up its parent thread using **Thread.wakeup**. Otherwise, if its parent thread wakes up without receiving an acknowledgement, this child thread is terminated by its parent thread using the **Thread.terminate** method.

While this thread-based approach is feasible for handling second type events, it introduces the overhead of creating and terminating threads. This is typically not a problem because it wastes a few hundred CPU cycles, less than 0.1ms. For computationally intensive applications, however, user applications want to reduce overhead. LiteOS provides another primitive for this purpose: callback functions.

A callback function registers to a second type event, which could range from radio events to detections of targets, and is invoked when such an event occurs. In the implementation of **SptTree**, a callback function is used to tell other modules that a message has been received. The **thread** class also provides a callback function, **registerRadioEvent**, to handle radio events, whose prototype is defined as follows:

```
//This function belongs to the thread class
  public void registerRadioEvent(uint8_t port, uint8_t *msg,\\
  uint8_t length, void (*callback)(void));
```

This interface requires the user thread to provide a buffer space for receiving incoming messages. After the kernel copies the message to this buffer, it invokes the callback function. Based on this mechanism, the previous reliable communication example can be implemented as follows:

[6]LiteOS classifies radio messages using **ports**, where the kernel delivers messages to threads listening on matching ports.

Application	Functionality
Simple applications (demonstrations of simple APIs, total: 11)	Blink, BlinkTask, CntToLeds, CntToLedsAndRfm, CntToRfm, CountRadio , GlowLeds, GlowRadio, ReverseUSART, RfmToLeds, SenseToLeds
Pong	Two nodes exchange messages
Sense	Sense the environment and report
SenseTask	Sense the environment and report, using tasks
GenericBase	Base station to receive messages
Oscilloscope	Displaysdata on PC
OscilloscopeRF	Displaysdata on PC, through the radio
SenseLightToLog	Read sensors and store the value in flash
SenseToRfm	Read sensors and report through the radio
SimpleCmd	Basic command interpretation
Surge	Basic multi-hop routing program

Figure 14. Benchmark Applications

Part I: Application

```
//Main thread, assuming we want to listen at MYPORT, and
//allocates a buffer to receive incoming packets
 bool wakeup = FALSE;
 uint8_t currentThread;
//Set up the current thread number
 currentThread = Thread.getCurrentThreadIndex();
//Register the event
 Thread.registerRadioEvent(MYPORT, msg, length, packetReceived);
//Sleep
 Thread.sleep(T_timeout);
//Has waken up and remove the register function
 Thread.unregisterRadioEvent(MYPORT);
//Am I waken up by a packet?
//Yes, by a packet
 if (wakeup == TRUE) {...}
//No, no acknowledgement yet
 else {...}
```

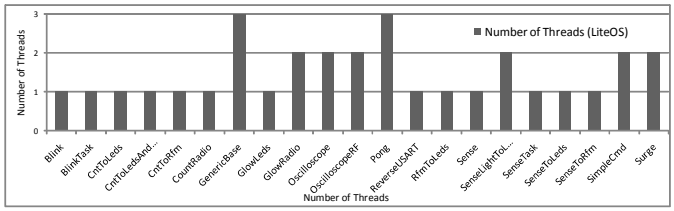
Part II: Callback function

```
void packetReceived()
{
 //the variables wakeup and currentThread are accessed by multiple
 //threads, hence atomic operations
  Thread.atomic_start();
  wakeup = TRUE;
  Thread.wakeup(currentThread);
  Thread.atomic_end();
}
```

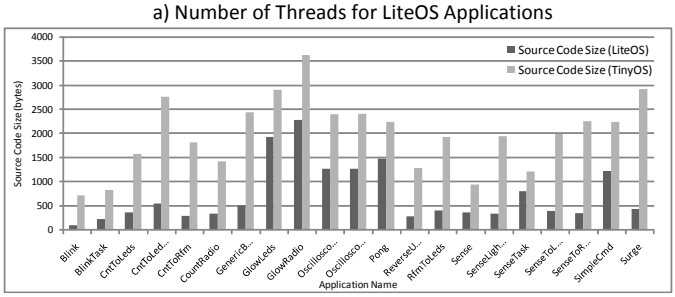
Generally, different from the TinyOS model, the LiteOS programming model primarily comprises threads and callback functions. Threads allow maintaining execution contexts and sequential programming, while callback functions provide an inexpensive way to handle external events. A potential risk of this model is race conditions. Due to space limitations, we only give a sketchy description on how we avoid this problem. Observe that in LiteOS, race conditions only exist between variables in user applications, because the variables in the kernel are not directly accessible by user applications. Hence, a programmer must ensure that every variable that is accessed asynchronously by multiple threads is protected by atomic operations. Otherwise, if a thread that is modifying this variable gets preempted, and another thread modifies this variable again, race conditions occur. One improvement is to automatically detect potential race conditions in the compiler, as the NesC compiler does.

3.4 Performance Evaluation

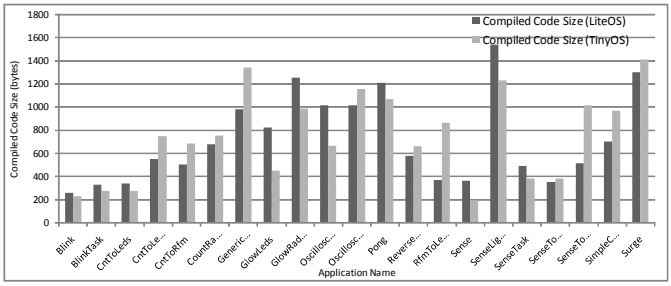
To evaluate the performance of LiteC++, we implemented a suite of 21 benchmark applications, ranging from simple API usage examples, such as Blink, to complete application skeletons, such as Surge. The details of such applications are shown in Figure 14. To compare LiteOS and TinyOS,



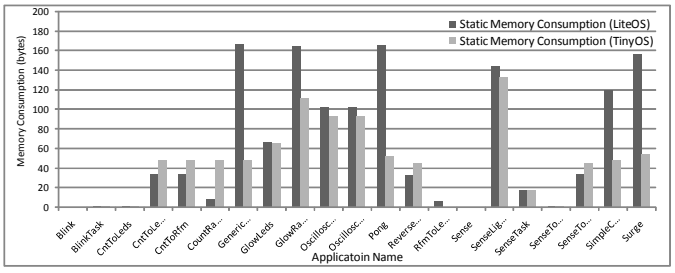
a) Number of Threads for LiteOS Applications



b) Source Code Length Comparison



c) Compiled Code Size Comparison



d) Static Memory Consumption Comparison

Figure 15. Benchmark Evaluation

our choices of benchmarks are limited to those available in standard TinyOS distributions (version 1.1.x) [7].

We used three metrics in the following experiments: code length as measured in bytes after removing all comments and blank lines (we did not use LOC because different programming languages may differ in the average line length), compiled number of bytes, where we use default settings for both LiteOS and TinyOS, and static RAM consumption of user applications [8]. We didn't include the LiteOS kernel in the comparison because the functionalities it provides, such as its support for the shell and file system, are not available in the TinyOS version of benchmark applications. To make fair comparisons, we only included *user side applications*, where the LiteC++ classes and TinyOS system modules are not included for the analysis.

While the LiteC++ applications directly compile into binaries that are measured, TinyOS applications are compiled into single binary images where system support and user applications are combined. To extract the user application part, we used the **module_memory_usage** script (in the **contrib** directory of TinyOS) to analyze its modules and only count those written by user applications. We do not modify any of the TinyOS applications in the experiments.

When writing LiteC++ applications, we implemented *exactly the same* functionalities as their TinyOS counterparts that were measured. One exception is the Surge application, where we only implemented the multi-hop spanning tree and data retrieval, while the message queuing and command broadcast functionality in the TinyOS version of Surge have not been implemented. Therefore, we compared the LiteOS version to a *partial* Surge application in TinyOS for fair comparison. Figure 15 shows the comparison results. Figure 15(a) shows the number of threads for each LiteOS application, while TinyOS applications are all single-threaded. Given that LiteC++ removes event handlers, wiring, and interfaces, it is not surprising that, as shown in 15(b), the source code size for LiteOS applications are typically smaller than their TinyOS counterparts. While such a reduction may not necessarily means that programming in LiteC++ is easier, it is at least promising for reducing the development cost of sensor network applications.

Figure 15(c) shows the compiled code size (consumption of program flash) comparison. Observe that while the LiteOS images include system calls, their code sizes are comparable to TinyOS, indicating that the design of classes and system calls does not introduce too much overhead.

Figure 15(d) shows the static memory usage comparison. Observe that several applications consume more memory in LiteOS than in TinyOS, because their LiteOS versions are multithreaded. In fact, there is a strong correlation between the number of threads and the consumed RAM, by comparing figures (a) and (d), because extra threads consume additional RAM for their own stacks. While it is possible to reduce the number of threads for some applications using callback functions, we choose not to optimize this way because we want to show that, based on the benchmarks, the *worst case* RAM consumption of multithreaded applications is still acceptable: even the **GenericBase** application with three threads consumes fewer than 200 bytes of RAM.

4 Perspective

In retrospect, several design choices become evident after finishing a working version of LiteOS. First, a really powerful advantage of LiteOS is that it supports interactive use. We believe an interactive system is much more productive, and our experiences using LiteOS confirm this. In fact, a few features immediately become our next-step goals, like supporting debugging by exporting state of running processes, retasking by switching between processes, and writing scripts

[7] LiteC++ also implements a postTask system call to implement the same functionality as the post keyword in TinyOS for applications like BlinkTask.

[8] We don't compare stack usage because the stack consumption of TinyOS is attributed to both system services and user applications. It is hard to measure stack consumption for user applications alone in TinyOS applications.

to automate complicated network-scale reconfigurations.

Second, our joint design of both the operating system and its programming environment gives us great leverage in finding the most appropriate design choices . We are not limited by software constraints other than GCC (but we find GCC is flexible enough not to be called a constraint). Controlling both the OS and the programming environment, we are able to experiment with radically different approaches, remove functional redundances across the system, and design everything from a clean slate.

How will LiteOS affect the way we interact with sensor networks? With a graphical shell, it can serve as a "sensor network drive"; with network support, it can implement "sensor net control" over web browsers. There are probably too many possibilities to enumerate in this paper. With an entirely new platform, such possibilities are opened doors to the future.

The remaining question is, will LiteOS be used? After building a working system and using it to develop a suite of applications (not limited to those used as benchmarks), we would like to speculate on whether such a system will be widely adopted. After all, users are not only concerned with what a system can do, but are also concerned with its limitations. So what are the potential obstacles for LiteOS to be widely used?

The primary current obstacle of LiteOS is that it is not compatible with TinyOS, or any other current OS for sensor networks. Therefore, we cannot directly use the established TinyOS code base. While we find it not difficult to manually port them, it is certainly desirable to have a translator from NesC to LiteC++. This problem may not be too difficult because NesC is translated into C before being compiled. Therefore, a translator only needs to extract the application-specific part from the generated C file, and change its dependencies from TinyOS modules to LiteC++ classes.

Another obstacle is conceptual: in sensor networks, TinyOS has become a de facto standard. There have been four to five additional operating systems that are already implemented. Why do we need yet another operating system? We believe the reason not only stems from the novel approaches and concepts proposed by LiteOS, but also from its affinity to UNIX and other APIs most familiar to mainstream embedded systems programmers. The goal of LiteOS is to "recruit" newcomers into our community. It offers an easy transition path for beginning programmer who is not yet familiar with event-based programming, wiring, and state machines. LiteOS and TinyOS therefore fill complementary needs, as far as sensor network development is concerned. Like TinyOS, LiteOS combines operating system development and compiler support into one package, which significantly enhances opportunities for performance optimization.

5 Conclusions

In this paper, we presented LiteOS, a UNIX-like, multithreaded operating system and programming environment for wireless sensor networks. We described the different subsystems of LiteOS and conducted a performance evaluation. We also presented performance evaluation of several macrobenchmarks developed in LiteC++, the programming language bundled with LiteOS. We are hopeful that the familiar abstractions exported by LiteOS will make it appealing to a larger category of mainstream embedded systems programmers, hence expanding the sensor network software developers beyond the current community. The authors are currently working on security and extensions (including wide-area and cross-platform extensions) to LiteOS.

References

[1] Arch rock corporation. http://www.archrock.com.

[2] ecos system. http://ecos.sourceware.org/.

[3] Sun spot project. http://www.sunspotworld.com/.

[4] T. Abdelazaher et al. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In *IEEE ICDCS*, 2004.

[5] S. Bhatti et al. Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms. In *ACM/Kluwer Mobile Networks and Applications (MONET), Special Issue on Wireless Sensor Networks*, 2005.

[6] CrossBow. Imote2 platform. http://www.xbow.com.

[7] CrossBow. Moteview software. http://www.xbow.com.

[8] H. Dai, M. Neufeld, and R. Han. Elf: an efficient log-structured flash file system for micro sensor nodes. In *ACM SenSys*, 2004.

[9] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of Emnets-I*, 2004.

[10] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *ACM Sensys*, 2006.

[11] D. Gay. Design of matchbox, the simple filing system for motes. Available at http://www.tinyos.net/tinyos-1.x/doc/matchbox-design.pdf.

[12] D. Gay, P. Levis, and D. Culler. Software design patterns for tinyos. In *Proceedings of the ACM LCTES*, 2005.

[13] 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 *Proceedings of PLDI*, 2000.

[14] L. Gu and J. A. Stankovic. t-kernel: Providing reliable os support to wireless sensor networks. In *ACM SenSys*, November 2006.

[15] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *Proceedings of Mobisys*, 2005.

[16] 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.

[17] J. Jeong and D. Culler. Incremental network programming for wireless sensors. In *IEEE SECON*, 2004.

[18] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. Peh, and D. Rubenstein. Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet. In *Proceedings of ASPLOS-X*, October 2002.

[19] O. Kasten and K. Römer. Beyond event handlers: Programming wireless sensors with attributed state machines. In *ACM IPSN*, 2005.

[20] A. Lachenmann, P. J. Marron, and K. Rothermel. Efficient flash-based virtual memory for sensor networks. In *Technical Report 2006/07, Universitat Stuttgart, Faculty of Computer Science*, 2006.

[21] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The Design of an Acquisitional Query Processor for Sensor Networks. In *ACM SIGMOD*, 2003.

[22] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Capsule: An energy-optimized object storage system for memory-constrained sensor devices. In *ACM Sensys*, November 2006.

[23] W. P. McCartney and N. Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *ACM Sensys*, 2006.

[24] L. Nachman, R. Kling, R. Adler, J. Huang, and V. Hummel. The intel mote platform: a bluetooth-based sensor network for industrial monitoring. In *Proceedings of IPSN*, 2005.

[25] S. Nath, J. Liu, and F. Zhao. Challenges in building a portal for sensors worldwide. In *First Workshop on World-Sensor-Web: Mobile Device Centric Sensory Networks and Applications (WSW)*, 2006.

[26] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and D. Tygar. Spins: Security protocols for sensor networks. In *Proceedings of Mobicom*, 2001.

[27] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. In *Proceedings of EMSOFT*, October 2003.

[28] N. Reijers and K. Langendoen. Effcient code distribution in wireless sensor networks. In *In Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, 2003.

[29] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proceedings of NSDI*, pages 29–42, 2004.

[30] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of Mobisys*, 2004.

[31] WindRiver. Vxworks operating system. http://www.windriver.com.