The LiteOS Operating System: Towards Unix-like Abstractions for Wireless Sensor Networks

Qing Cao, Tarek Abdelzaher
Department of Computer Science
University of Illinois at Urbana-Champaign
{qcao2, zaher}@cs.uiuc.edu

John Stankovic
Department of Computer Science
University of Virginia
stankovic@cs.virginia.edu

Tian He
Department of Computer Science
University of Minnesota
tianhe@cs.umn.edu

Abstract

This paper presents LiteOS, a multi-threaded operating system that provides Unix-like abstractions for wireless sensor networks. Aiming to be an easy-to-use platform, LiteOS offers a number of novel features, including: (1) a hierarchical file system and a wireless shell interface for user interaction using UNIX-like commands; (2) kernel support for dynamic loading and native execution of multithreaded applications; and (3) online debugging, dynamic memory, and file system assisted communication stacks. LiteOS also supports software updates through a separation between the kernel and user applications, which are bridged through a suite of system calls. Besides the features that have been implemented, we also describe our perspective on LiteOS as an enabling platform. We evaluate the platform experimentally by measuring the performance of common tasks, and demonstrate its programmability through twenty-one example applications.

1 Introduction

This paper introduces LiteOS, an operating system that provides Unix-like abstractions to wireless sensor networks. LiteOS maps a sensor network into a UNIX-like file system, and supports extremely resource-constrained nodes such as MicaZ. It supports C programming natively, and allows online debugging to locate application bugs. We believe that such an operating system could potentially expand the circle of sensor network application developers by providing a familiar programming environment. 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 superimposes a familiar IP space on mote platforms to reduce the learning curve of network programming and management).

Our key contribution is to present a familiar, Unix-like abstraction for wireless sensor networks by leveraging the likely existing knowledge that common system programmers (outside the current sensor network community) already have: Unix, threads, and C. By mapping sensor networks to file directories, it allows applying user-friendly operations, such as file directory commands, to sensor networks, therefore reducing the learning curve for operating and programming sensor networks.

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 the hierarchical file system, or are only partially supported. Compared to the latter category (conventional embedded operating systems), such as VxWorks [29], eCos [2], embedded Linux, and Windows CE, LiteOS has a much smaller code footprint, running on platforms such as MicaZ, with an 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 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 software is available for download at the website http://www.liteos.net, including its manual, programming guide, application notes, and source code.

The rest of this paper is organized as follows. Section 2 describes the related work. Section 3 introduces the LiteOS operating system and describes its design and implementation. Section 4 introduces its programming environment and application examples. Section 5 presents the evaluation results. Section 6 outlines more features that LiteOS currently supports. Section 7 discusses the potential of this platform. Section 8 concludes this paper.

2 Related Work

The rapid advances of sensor networks in the past few years created many exciting systems and applications. Operating systems such as TinyOS [16], Contiki [9], SOS [15], Mantis [5], t-Kernel [14], and Nano-RK [10] 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.

In additional to proposing several novel features, LiteOS also draws from previous research efforts. Its shell subsystem is directly inspired by Unix, by adopting Unix-like commands. The advantage of such command-line shells is that they are well-established mechanisms to interact with complicated systems, providing powerful scripting and automation support. Previous systems, such as Mantis and Contiki, have implemented shell interfaces on resource-constrained motes. In contrast, LiteOS implements the shell on the resource-rich PC side (basestation). This design choice allows us to implement more powerful commands, because the shell is no longer constrained by sensor node resources.

There have also been efforts to provide file systems for MicaZ-class sensor nodes, such as MatchBox [11] and ELF [8], as well as flash-based systems, where additional flash is plugged to sensor nodes, such as Capsule [22]. To implement the file system of LiteOS, we did not adopt the former because their interfaces do not match our goals: Matchbox and ELF do not support hierarchical file organizations, and only provide basic abstractions for file operations such as reading and writing. On the other hand, we did not adopt the latter because it requires an external flash attachment. In contrast, we want users with off-the-shelf MicaZ nodes to be able to directly use LiteOS.

The third subsystem of LiteOS, its kernel, features dynamic loading, a feature that is taken for granted in almost every modern operating system. One approach to implement dynamic loading is based on virtual memory. While virtual memory support in MicaZ-class sensor networks has been implemented [14, 20], we observe that such efforts 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 supporting 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.

The dynamic loading mechanism of LiteOS follows the line of several previous efforts in the literature that did not involve virtual memory such as 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

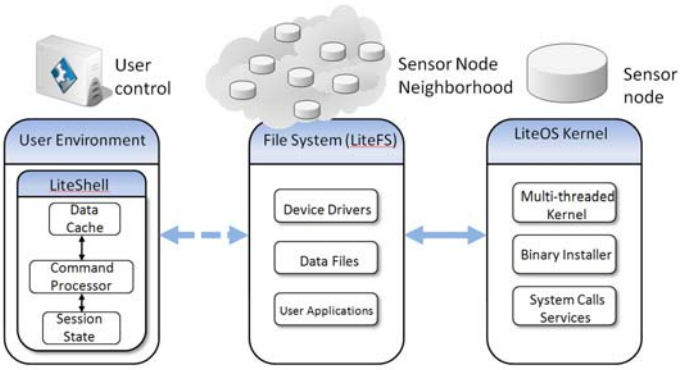


Figure 1. LiteOS Operating System Architecture

has size limitations: the compiler for the AVR platform (MicaZ) only supports such binaries up to 4K bytes. Contiki takes yet another approach, where it parses ELF files to patch all unsolved symbolic links, and performs binary relocation with the help of a symbol table in its kernel. The approach in LiteOS does not handle ELF files directly, instead, it uses modified HEX files, which are smaller in size, to store relocation information.

3 The LiteOS Operating System

In this section, we present the LiteOS operating system. We first present an overview and its design choices, then we describe its three subsystems.

3.1 Architectural Overview

Figure 1 shows the overall architecture of the LiteOS operating system, partitioned into three subsystems: LiteShell, LiteFS, and the kernel. Implemented on the base station PC side, 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 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), our wireless mount currently supports 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, rm, mkdir, touch, pwd, du, chmod
Process Commands	ps, kill, exec
Debugging Commands	debug, list, print, set breakpoint, continue, snapshot, restore
Environment Commands	history, who, man, echo
Device Commands	./DEVICENAME

Once mounted, a LiteOS node looks like a *file directory* from the base station. A sensor network, therefore, maps into a higher level directory composed of node-mapped directories. The shell, called LiteShell, supports UNIX commands, such as copy (cp), 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].

3.2 LiteShell Subsystem

The LiteShell subsystem provides Unix-like command-line interface to sensor nodes. This shell runs on the base station PC side. Therefore, it is a front-end that interacts with the user. The motes do not maintain command-specific state, and only respond to translated messages (represented by compressed tokens) from the shell, which are sufficiently simple to parse. Such an asymmetric design choice not only significantly reduces the code footprint of the LiteOS kernel that runs on motes, but also allows us to easily extend the shell with more complicated commands, such as authentication and security.

Currently, we have implemented the commands listed in Table 3. They fall into five categories: file commands, process commands, debugging commands, environment commands, and device commands.

3.2.1 File Operation Commands

File commands generally maintain their Unix meanings. For example, the **ls** command lists directory contents. It supports a **-l** option to display 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. As an example, a **ls -l** command may return 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 1 and 2.

Once sensor nodes are mounted, a user uses the above commands to navigate the different directories (nodes) as if they are local. 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. The remaining file operation commands are intuitive. Since LiteFS supports a hierarchical file system, it provides **mkdir**, **rm** and **cd** commands.

3.2.2 Process Operation Commands

LiteOS has a multithreaded kernel to run applications as threads concurrently. LiteShell provides three commands to control thread behavior: **ps**, **exec**, 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[1], and is located under the C drive of the user's laptop. 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 may use the following commands:

```
$ pwd
   Current directory is  /sn01/node101/apps
$ cp /c/Blink.lhex Blink.lhex
    Copy complete
$ exec Blink.lhex
    File Blink.lhex successfully started
$ ps
    Name   State
    Blink  Sleep
```

As illustrated in this example, we first copy an application, **Blink.lhex**, into the **/apps** directory, so that it is stored in the LiteFS file system. We then use the **exec** command to start this application. The implementation of **exec** is as follows. The processor architecture of 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 reprograms part of the flash to run the application.

Once the Blink application is started, the user may view its thread information using the **ps** command. Finally, the **kill** command is used to terminate threads.

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

3.2.3 Debugging Commands

We now describe the debugging commands. Eight commands are provided, including those for setting up the debugging environment (**debug**), watching and setting variables (**list**, **print**, and **set**), adding/removing breakpoints (**breakpoint** and **continue**), and application checkpoints (**snapshot** and **restore**). Note that all debugging commands keep information on the front-end, i.e., the PC side. In fact, there is no debugging state stored on the mote, which means that there is no limit on the maximum number of variables (or the size of variables) can be watched, or how many breakpoints can be added. We now briefly explain these commands. Detailed documentation of these commands can be found in the LiteOS manual.

The user first invokes the **debug** command to initiate the environment. This command takes the source code directory of the application as its parameter. For example, if supplied with the kernel source code location, it allows debugging the kernel itself. Once invoked, this command parses the source code as well as the generated assembly to gather necessary information, such as memory locations of variables. Such information is then used by other debugging commands for diagnosis purposes. For instance, it is used by the command **list** to display the current variables and their sizes, commands **print** and **set** to watch and change variable values, and commands **breakpoint** and **continue** to add and remove breakpoints. Once a breakpoint is added, the command **ps** tells whether a thread has reached the breakpoint.

We now explain the commands **snapshot** and **restore**. **Snapshot** allows adding a checkpoint to an active thread, by exporting all its memory information, including variable values, stack, and the program counter, to an external file. **Restore**, on the other hand, allows importing such memory information from a previously generated file, essentially restoring a thread to a previous state. Combined use of these two commands allows replaying part of an application by *rewinding* it, and is particularly useful for locating unexpected bugs.

3.2.4 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 meanings of these commands are similar to their Unix counterparts.

3.2.5 Device Commands

The LiteOS shell provides an easy way to interact with the sensors. Every time the file system is initialized, a directory **dev** is created, which contains files that map to actual device drivers. On MicaZ, the **dev** directory contains the following files:

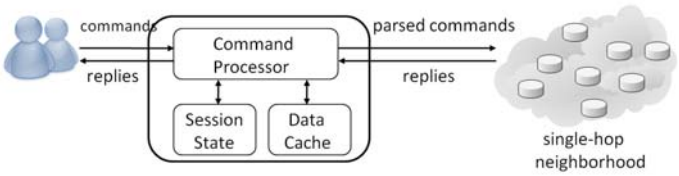


Figure 2. LiteShell Implementation

```
$ls
led, light, temp, magnet, accel, radio
```

In this directory, led refers to the LED device. There are four sensors, light, temperature, magnetic, and accelerator, respectively. There is also the radio device, which sends and receives packets. An example of reading 100 data samples from the light sensor at a frequency of 50 milliseconds is written as follows, where the first parameter is the frequency and the second parameter is the number of readings.

```
./light 50 100
```

3.2.6 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. To reduce overhead, 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 an ACK-based reliable communication protocol.

As an additional note, not every command can always be successfully completed. For example, when the user installs an application, if the remaining program flash and RAM of a node are not sufficient for a binary to be loaded, the node responds with an error message.

3.3 LiteFS Subsystem

We now describe the LiteFS subsystem. The interfaces of LiteFS provide support for both file and directory operations.The APIs of LiteFS are listed in Table 2.

While most of these APIs resemble those declared 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 on EEPROM and the data flash, respectively. Another feature of LiteFS is that it supports simple search-by-filename using the **fsearch** API, where all files whose names match a query string are returned. These APIs can be exploited in two ways; either by using shell commands interactively, or by using application development libraries.

3.3.1 Implementation of LiteFS

Figure 3 shows the architecture of LiteFS, which is partitioned into three modules. It uses RAM to keep opened files

Table 2. 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);

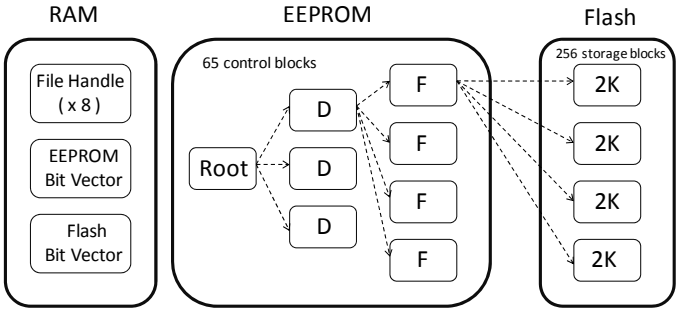


Figure 3. LiteFS Architecture

and the allocation information of EEPROM and the data flash in the first module, uses EEPROM to keep hierarchical directory information in the second, and uses the data flash to store files in the third. Just like Unix, files in LiteFS represent different entities, such as data, application binaries, and device drivers. A variety of device driver files, including radio, sensor, and LED, are supported. Their read/write operations are mapped to real hardware operations. For example, 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, our current version of LiteOS supports eight file handles (this number is adjustable according to application needs), where each handle occupies 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 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

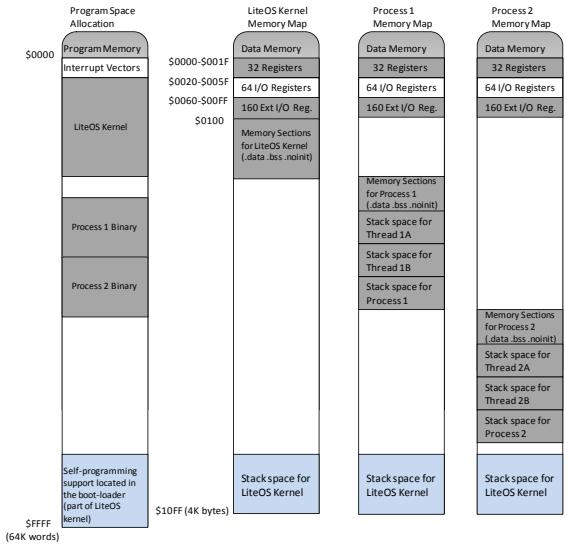


Figure 4. LiteOS Memory Organization

the file system is formatted. The other 64 blocks are either directory blocks (specified with D) or file blocks (specified with F) according to application needs. Just like Unix, files represent data, binary applications, and device drivers.

We follow three design choices in LiteFS. First, the maximal length of a file name is 12 bytes. Therefore, the *Eight Dot Three* naming system is supported. Second, a file control block addresses at most ten logical 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 old one. Third, all control blocks reside in EEPROM rather than in RAM.

3.4 LiteOS Kernel Subsystem

3.4.1 Kernel Overview

Threads are the most commonly used concurrency mechanism in modern general purpose operating systems. In sensor networks, there are design tradeoffs between using threads and events. Both approaches have been implemented in previous research efforts. TinyOS [16] and SOS [15] are based on events, Mantis [5] and TinyThreads [23] choose threads, and Contiki [9] provides support for both.

The kernel subsystem of LiteOS takes the thread approach, but it also allows user applications to handle events using callback functions for efficiency. We implement both priority-based scheduling and round-robin scheduling in the kernel. We also support dynamic loading and un-loading of user applications, as well as a suite of system calls for the separation between kernel and applications. We now describe these topics in more detail.

3.4.2 Dynamic Loading in LiteOS

LiteOS provides two approaches for dynamic loading of user applications. These approaches allow multiple threads to be concurrently executed without memory access conflicts. Figure 4 illustrates such an example, where six threads (besides the kernel) are executed. Note that the memory sections of different threads do not overlap, and threads are executed natively.

In the first loading approach, we assume that the source code is available. Every time the user needs to load the binary to a different location, the source code is re-compiled with different memory settings. This approach introduces no extra overhead, and generates smaller binary images.

In the second approach, we do not assume the source code is available for the user. This approach remotely echoes the *differential patching* idea [17, 28] in application distributions over networks. Different from that idea, however, we directly encode relocation 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 images for relocation.

Our development of mathematical models for differential patches is driven by practical observations, and is customized for our GCC-based toolchain. We start presenting its details by observations of binary 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. 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 [23, 27]. Therefore, we assume that the stack top has been statically decided.

Because $(\mathbb{S}, \mathbb{M}, \mathbb{T})$ is volatile, we have to find a way to encode its 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 our customized GCC compiling environment, only the following six instructions are affected: **LDI**, **LDS**, **STS**, **JMP**, **CALL**, **CPI**. The detailed meanings of these instructions can be found in the datasheet of Atmega128. We call them *differential instructions*.

We use an example to illustrate the major steps of our address translation procedure. In this example, we sample the light sensor for 100 times, at a frequency of once per second, and write readings into a local file in LiteFS. Using the GCC compiler, this application compiles to 358 instructions consuming 790 bytes of binary. We compiled it with

Section	Address	M = 0x500	M = 0x504	M = 0x600

_do_copy
_data
	① 19098
	② 1909a
	③ 1909c
	④ 190ac
	⑤ 190ae
		ldi r17, 0x05
		ldi r26, 0x00
		ldi r27, 0x05
		cpi r26, 0x0C
		cpc r27, r17
			ldi r17, 0x05
			ldi r26, 0x04
			ldi r27, 0x05
			cpi r26, 0x10
			cpc r27, r17
				ldi r17, 0x06
				ldi r26, 0x00
				ldi r27, 0x06
				cpi r26, 0x0C
				cpc r27, r17

_do_clear
_bss
	⑥ 190b2
	⑦ 190b4
	⑧ 190b6
	⑨ 190bc
	⑩ 190be
		ldi r17, 0x05
		ldi r26, 0x0C
		ldi r27, 0x05
		cpi r26, 0x14
		cpc r27, r17
			ldi r17, 0x05
			ldi r26, 0x10
			ldi r27, 0x05
			cpi r26, 0x18
			cpc r27, r17
				ldi r17, 0x06
				ldi r26, 0x0C
				ldi r27, 0x06
				cpi r26, 0x14
				cpc r27, r17

main	⑪ 190d2
	⑫ 190d4
	⑬ 190d6
	⑭ 190d8
		ldi r22, 0x00
		ldi r23, 0x05
		ldi r24, 0x02
		ldi r25, 0x05
			ldi r22, 0x04
			ldi r23, 0x05
			ldi r24, 0x06
			ldi r25, 0x05
				ldi r22, 0x00
				ldi r23, 0x06
				ldi r24, 0x02
				ldi r25, 0x06

Figure 5. Binary Image Difference Example

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 5.

We list a total of 14 instructions, including two instructions (⑤ and ⑩) that are not differential, but are useful for our following explanations. Before illustrating what we can do, we start with what we cannot do. Generally, it is almost impossible to leverage the semantic relationships to construct models. For example, in Figure 5, instructions ① to ⑤ are part of the initialization of the data segment before the main program runs. The same is true for instructions ⑥ to ⑩ , which serve the function of clearing the bss section (for storing global variables). Such kind of implicit relationships only emerge by application specific analysis, but are beyond the capability of our mathematical models.

In our design, we take an application independent approach to model the impact of $(\mathbb{S}, \mathbb{M}, \mathbb{T})$. 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. For differential instructions, their values under different $(\mathbb{S}, \mathbb{M}, \mathbb{T})$ vectors demonstrate a set of linear displacements. This follows directly from the format of machine code for various addressing modes. In the previous example, 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, the values of a, b, L and R are to be solved, $>>$ is the right-shifting operation, and $<<$ is the opposite. 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 $\mathbb{T}$. The final form of f is typically a superposition of different linear equations for $\mathbb{S}$, $\mathbb{M}$, and $\mathbb{T}$, and is sufficiently simple to be encoded into binary images. To ensure that our approach works correctly, we always generate *training* binary images, serving as verifications of the mathematical models.

3.4.3 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 an updated kernel will 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 compatibility. 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. Compared to directly invoking kernel functions, each system call adds 5 instructions (10 CPU cycles), a sufficiently low overhead to be supported on MicaZ. At the user side, interactions with system calls are provided as libraries. The details of call gates are therefore hidden from user applications.

4 The Programming Environment

In this section, we briefly describe the programming environment of LiteOS. This section is organized as follows. We first present a concrete programming example, then we describe the programming model details.

4.1 The "Hello, World" Example

As a concrete example of how to program in LiteOS, we use the classical "Hello, World" example. The source code of this application is shown as follows. Lines starting with **#include** are not listed.

```
int main()
{
  while (1) {
     radioSend_string("Hello, world!\n");
     greenToggle();
     sleepThread(100);
     }
  return 0;
}
```

4.2 Programming Model

We use *programming model* to refer to a developer's view of a programming environment. In TinyOS/NesC [12,

13], the view is that an application is driven by events, which can optionally post long-running tasks to a FIFO scheduler. A complicated program 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 sending a packet, a node waits for a period $T_{timeout}$ (after which the previous packet is considered lost), or stops 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, and handles their possible transitions. 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 protothreads [9] and the OSM model [19].

In contrast to the event based programming model adopted by TinyOS, LiteOS uses threads to maintain execution contexts. For example, in the previous "Hello, World" example, every time the function **radioSend_string** is called, the thread is suspended until the radio operation finishes. Threads, however, do not completely eliminate the use of events for efficiency reasons. Observe that there are two types of events: internally generated events, and externally generated events. For instance, a **sendDone** event always follows a packet sending operation, and is therefore internally generated. A radio receive event, however, is triggered by external events, and may not be predictable. LiteOS treats these two types of events separately. Generally, internally generated events are implicitly handled by using threads, where events like sendDone are no longer visible. It is the externally generated events that deserves special attention.

We illustrate how externally generated events are handled using the radio receive event as an example. LiteOS provides two solutions to handle this event. The first is to create a new thread using the **createThread** system call, which blocks until the message arrives[2]. Consider a 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 the **wakeupThread** method. Otherwise, if its parent thread wakes up without receiving an acknowledgement, this child thread is terminated by its parent thread using the **terminateThread** method.

While this thread-based approach is usually sufficient for handling externally generated 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 an event, which could range from radio events to detections of targets, and is invoked when such an event occurs. For instance, the function **registerRadioEvent** tells that a message has been received. Its prototype is defined as follows:

```
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:

Part I: Application

```
1   bool wakeup = FALSE;
2   uint8_t currentThread;
3   currentThread = getCurrentThreadIndex();
4   registerRadioEvent(MYPORT, msg, length, packetReceived);
5   sleepThread(T_timeout);
6   unregisterRadioEvent(MYPORT);
7   if (wakeup == TRUE) {...}
8   else {...}
```

Part II: Callback function

```
9   void packetReceived()
10  {
11  _atomic_start();
12  wakeup = TRUE;
13  wakeupThread(currentThread);
14  _atomic_end();
15  }
```

We briefly explain this example. First, the thread listens on MYPORT, and allocates a buffer for receiving incoming packets (line 4). This thread then sleeps (line 5). When it wakes up, it checks if it has been woken by the callback function (line 7), or by a timeout event (line 8). The thread then handles these two cases separately.

A potential risk of this model is race conditions, which only exist between variables in user applications, because the variables in the kernel are not directly accessible. 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.

5 Evaluation

5.1 LiteShell Performance Evaluation

We now evaluate the performance of LiteShell, focusing on response time. In the following experiments, we classify

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

Table 3. Shell Commands Classification

Command List	
Local Commands	pwd, echo, who, man, history
Network Commands	cp, rm, mkdir, touch du, ps, kill, exec, ls

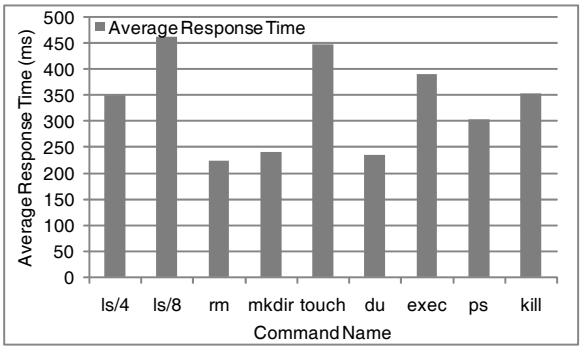


Figure 6. Average Command Response Time of LiteShell

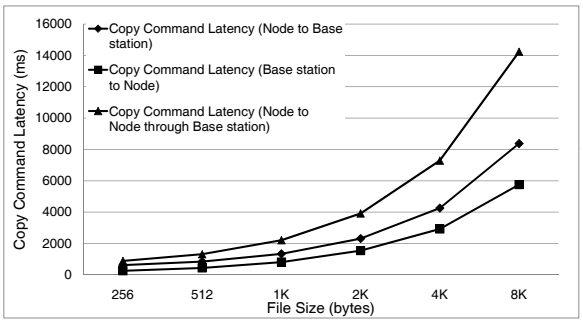


Figure 7. Average Copy Time using LiteShell

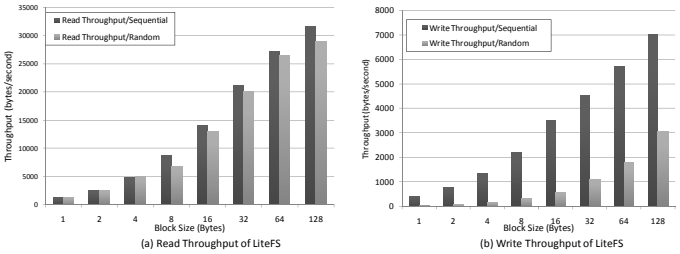


Figure 8. LiteFS Throughput

representative shell commands into two categories: those executed locally, and those communicating with nodes, as listed in Table 3.

We observe that local commands are typically finished within one millisecond. Therefore, it is the second category of commands that affects users' experience. 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 6.

As measured in this experiment, the average response time of these commands is typically less than half a second. While we can further optimize their performance 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 7. As illustrated in this figure, the transfer rate is acceptable for common uses.

5.2 LiteFS Performance

To evaluate the performance of LiteFS, we focus on its throughput. The experiment results are shown in Figure 8. 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 read/write 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 form 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.

5.3 Comparison of Programming Environments

We have implemented a suite of libraries for user programming in LiteOS. To evaluate its performance, we select 21 benchmark applications shown in Figure 9. To compare LiteOS and TinyOS, our choices of benchmarks are limited to those available in standard TinyOS distributions (version 1.1.x). All LiteOS applications are compiled into the modified HEX format (lhex format). We do not choose the ELF format because it has a much larger size (usually 5-10 times larger) compared to the HEX format.

We used three metrics in the following experiments: code length as measured in lines of code after removing

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	Display data on PC
OscilloscopeRF	Display data 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

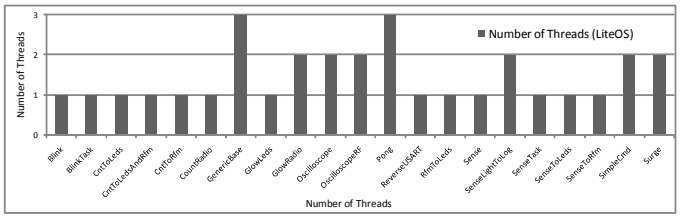
Figure 9. Benchmark Applications

all comments and blank lines, compiled number of bytes, where we use default settings for both LiteOS and TinyOS, and static RAM consumption of user applications[3]. 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 TinyOS system modules are not included for the analysis (we used the **module_memory_usage** script in the **contrib** directory of TinyOS to extract such modules).

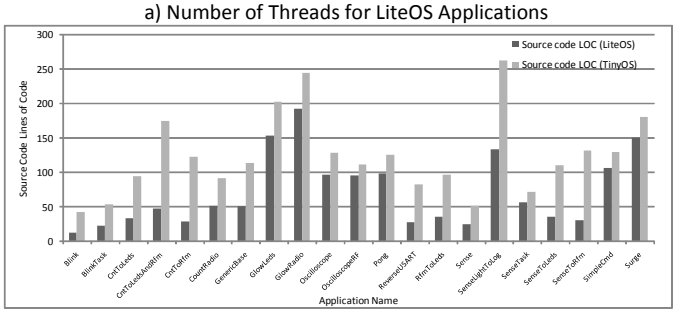
When writing benchmark applications under LiteOS, we implemented *exactly the same* functionalities as their TinyOS counterparts that were measured. One exception is the Surge application, where we only implemented the multi-hop spanning tree and data retrieval, while the message queuing and command 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 10 shows the comparison results. Figure 10(a) shows the number of threads for each LiteOS application, while TinyOS applications are all single-threaded. Given that LiteOS removes event handlers, wiring, and interfaces, it is not surprising that, as shown in 10(b), the source code sizes of LiteOS applications are typically smaller than their TinyOS counterparts. While such a reduction may not necessarily mean that programming in LiteOS is easier, it is at least promising for reducing the development cost of sensor network applications.

Figure 10(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 LiteOS programming model does not introduce too much overhead.

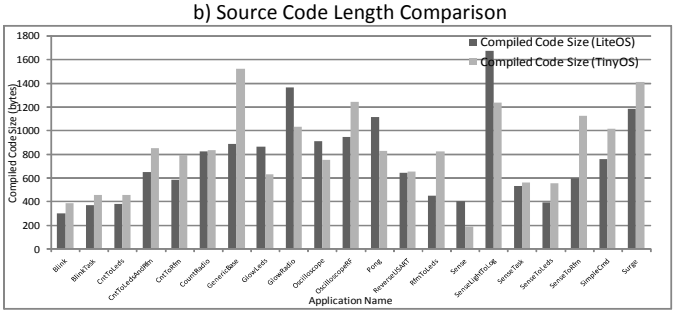
[3]We don't compare stack usage because the stack consumption of TinyOS is attributed to both system services and user applications. It is hard to measure stack consumption for user applications alone in TinyOS applications.



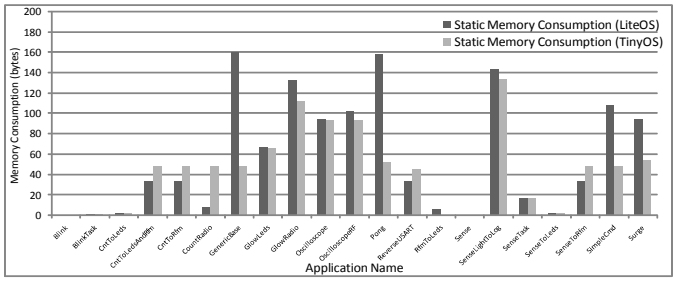
a) Number of Threads for LiteOS Applications



b) Source Code Length Comparison



c) Compiled Code Size Comparison



d) Static Memory Consumption Comparison

Figure 10. Benchmark Evaluation

Figure 10(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.

6 Other Features of LiteOS

In this section, we describe three additional features f the LiteOS platform. Detailed documentation of these features is available at www.liteos.net.

6.1 File Assisted Communication Stacks

The design of flexible and powerful communication stacks has been a critical challenge to sensor network applications. In LiteOS, we treat communication stacks, such as routing and MAC protocols, as threads. Hence, different communication protocols can be exchanged easily, and dynamically loaded just like any user application.

To use a protocol, an application organizes the header of packets in such a way that the communication protocol can correctly parse it. Packets are then sent to the port which the protocol is listening on. For example, a flooding protocol uses a different listening port compared to a multi-hop routing protocol. In the following example, our multi-hop routing protocol listens on port 10, and the application sends a packet of 16 bytes through this protocol using the following code sample:

```
//10 is the port number, 0 means local node protocol
//16 is the message length, and msg is the message pointer
 radioSend(10, 0, 16, msg);
```

Our implementation of this multi-hop protocol (geographic forwarding) maintains a neighbor table of 12 entries, and contains around 300 lines of code. Its binary image consumes 1436 bytes of program flash, and 260 bytes of RAM, including stack. Its overhead is so low that it can co-exist with other protocols, such as flooding. The source code of this protocol is available in LiteOS 0.3.

6.2 Dynamic Memory

LiteOS supports dynamic memory at the kernel level. The dynamic memory is provided as library APIs (**malloc** function and **free** function) to user applications through system calls. The dynamic memory grows in the opposite direction from the LiteOS stack, and resides completely in the unused memory between the end of the kernel variables, and the start of user application memory blocks. Therefore, the size of dynamic memory can be adjusted after deployment according to user applications' needs.

6.3 Event Logging

Visibility is a key challenge for wireless sensor network applications. Deployed on the extremely resource-constrained mote platform, such applications may fail unexpectedly, or exhibit behavior different from their intended goals. To help understand why such problems occur, we design and implement an event trace logger in LiteOS, which allows us to partially reconstruct application behavior after execution, such as which path it took for an IF statement, its invocation history of the kernel system calls, and the dynamics of its behavior across nodes.

Briefly speaking, the event logging service of LiteOS is implemented as follows. We keep an internal buffer (its size decided by the user) to record the most recent application events. We only log one application at a time. Therefore, we only need one byte for most events, up to 256 different types, including all the system calls, certain kernel events such as context switches and driver invocations, and application specific events inserted by the user. Every time an event triggers, a corresponding byte is written into the buffer. When the buffer is full, we write its content into a file stored in the external flash. Such recorded traces are obtained after experiments and are translated into a recognizable sequence of events. Therefore, the sensor node is no longer a black box. Instead, we now have valuable insight on why an application fails, and what we should do to correct unexpected software glitches.

7 Perspective

In retrospect, several design choices become evident after finishing a prototype version of LiteOS. One advantage of LiteOS is that it supports interactive use. We believe an interactive system (e.g., one that offers an interactive shell to users and programmers) leads to improved productivity at development time. Our experiences using LiteOS confirm this thesis. How will LiteOS affect the way we interact with sensor networks? With a graphical shell, it can serve as a sensor network drive that can be mounted to a PC and controlled over Web browsers. Sensor networks can then become just another common peripheral.

Apart from promoting interactive use, there is a conceptual question on the need for LiteOS. In sensor networks, a few other operating systems have already been implemented. Why do we need yet another one? To better understand the differences and similarities between LiteOS and other operating systems, we compare some of them in Figure 11 for reference. The design of LiteOS is partially inspired by Unix, as the title suggests. We believe that it is precisely its affinity to UNIX that could make LiteOS easier to be adopted by mainstream system programmers. One goal of LiteOS is to hopefully offer an easy transition path for those beginning sensor networks programmers who are experienced in conventional systems programming, but not proficient with event-based programming, wiring, and state-machine abstractions of program execution. LiteOS and the previous sensor network operating systems, such as TinyOS, therefore, fill complementary needs. We hope that this diversity brings us one step closer to making sensor network programming attainable to system programmers without steep learning curves.

	LiteOS	TinyOS	Mantis	Contiki	SOS
Current license	GPL	BSD	BSD	BSD	Modified BSD
Website	www.liteos.net	www.tinyos.net	mantis.cs.colorado.edu	www.sics.se/contiki	projects.nesl.ucla.edu/public/sos-2x/doc/
Remote scriptable wireless shell	Yes (on the base PC, Unix commands supported)	No (application specific shell such as SimpleCmd exists)	No (on-the-mote shell is supported)	No (on-the-mote shell is supported)	No
Remote file system interface for networked nodes	Yes	No	No	No	No
File system	Hierarchical Unix-like	Single level (ELF, Matchbox)	No (will be available in 1.1)	Single Level	No
Thread support	Yes	Partial (through TinyThreads)	Yes	Yes (also supports ProtoThreads)	No
Event based programming	Yes (through callback functions)	Yes	No	Yes	Yes
Remote Debugging(e.g. watch and breakpoints)	Yes	Yes (through Clairvoyant)	Partial (through NodeMD)	No	No
Wireless reprogramming	Yes (application level)	Yes (whole system image replacement)	No (will be available in 1.1)	Yes	Yes (module level)
Dynamic memory	Yes	Yes (in 2.0 or through TinyAlloc for 1.x)	No	Yes	Yes
First publication/release date	2007	2000	2003	2003	2005
Platform support	MicaZ and AVR series MCU	Mica, Mica2, MicaZ, Telos, Tmote, XYZ, Iris (among others)	Mica2, MicaZ, Telos	Tmote, ESB, AVR series MCU, certain old computers	Mica2, MicaZ, XYZ
Simulator	Through AVRORA	TOSSIM, PowerTossim	Through AVRORA	Netsim, Cooja, MSPSim	Source level Simulator/ Through AVRORA

Note: [1] Only features in the current version of these operating systems as of the publication of this paper (April 2008) are compared
[2] Only open-source sensor network OS projects that have websites are compared
[3] All OS systems can be directly debugged using JTAG or GDB. Such comparison is not included.

Figure 11. Comparison with Other Operating Systems

8 Conclusions

In this paper, we presented a software development platform called LiteOS, and conducted its performance evaluation. We also presented performance results of several benchmark applications. We are hopeful that the familiar abstractions exported by LiteOS will make it valuable as a new research platform, and appealing to a larger category of systems programmers.

Acknowledgements

We gratefully acknowledge the anonymous reviewers and our shepherd, Adam Dunkels, for their insightful comments. We also thank Simon Han and Richard Han for their comments on the OS comparison table. We thank our colleagues and the users of LiteOS who have provided feedback and help to make this system possible. This paper is supported in part by NSF grants, CNS 05-54759, CNS 06-15318, CNS 06-26342, and CNS 06-26825, and a fellowship from Vodafone Foundation.

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. Eswaran, A.Rowe, and R. Rajkumar. Nano-rk: an energy-aware resource-centric rtos for sensor networks. In *IEEE RTSS*, 2005.
[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] WindRiver. Vxworks operating system. http://www.windriver.com.