Enix: A Lightweight Dynamic Operating System for Tightly Constrained Wireless Sensor Platforms

Yu-Ting Chen
Department of Computer Science
National Tsing Hua University
g9662595@oz.nthu.edu.tw

Ting-Chou Chien
Department of EECS
University of California, Irvine
tchien@uci.edu

Pai H. Chou
Dept. of EECS, UC Irvine and
National Tsing Hua Univ.
phchou@uci.edu

Abstract

Enix is a lightweight dynamic operating system for tightly constrained platforms for wireless sensor networks (WSN). Enix provides a cooperative threading model, which is applicable to event-based WSN applications with little run-time overhead. Virtual memory is supported with the assistance of the compiler, so that the sensor platforms can execute code larger than the physical code memory they have. To enable firmware update for deployed sensor nodes, Enix supports remote reprogramming. The commonly used library and the main logical structure are separated; each sensor device has a copy of the dynamic loading library in the Micro-SD card, and therefore only the main function and user-defined subroutines should be updated through RF. A lightweight, efficient file system named EcoFS is also included in Enix. The code size and data size of Enix with full-function including EcoFS are 8 KB and 512 bytes, respectively, enabling Enix to run on many RF-enabled systems-on-chip that cannot run most other WSN OSs.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems; D.4.7 [**Operating Systems**]: Organization and Design

General Terms

Design, Performance

Keywords

Wireless sensor networks, operating systems, file system, reprogramming, position-independent code, cooperative threads, scheduling, demand paging.

1 Introduction

The right runtime support can make a great difference in the amount of effort required to develop applications for wireless sensor networks (WSN). Many microcontroller units (MCU) with integrated radios (RF-MCU) are commercially available and would be ideal for WSN platforms; unfortunately, most of them are not supported by most WSN operating systems (WSN OS) to date, primarily due to the tight resource constraint. While it is possible to build many applications without such an OS, the code tends to be fragile and requires re-inventing the wheel. Our experience with several real-world applications motivated us to build a new WSN OS.

1.1 Motivating Application

Our work is motivated by several real-life applications, one of which is described here. It is a machine-health monitoring system for a "stocker," or a robot arm that moves glass panels in an LCD factory. The system samples triaxial acceleration at several hundred Hz at several points on the stocker and wirelessly transmits the data to the host PC for analysis. The robot arm imposes a size constraint on the wireless sensor node, which necessitates the use of integrated RF-MCU components. It turns out that the overwhelming majority of such RF-MCUs contain an 8051-compatible core, as shown in Fig. 1. This is not surprising, given that every 1 out of 5 MCUs is an 8051 [13]. Unfortunately, most WSN OSs proposed to date are based on either Atmel or TI MSP430, which add up to less than 1 out of 20, as shown in Table 1, and only one or two options are available for MSP430 with an integrated radio (CC430). Although the TinyOS 8051 Working Group has been trying to port TinyOS to the 8051 for years, the latest available version is 0.1 pre-release from nearly two years ago. On the other hand, several OSs such as μC/OS-II and FreeRTOS currently run on these platforms, but they lack features for WSN support. Therefore, a new WSN OS is sorely needed to take advantage of these RF-MCUs.

One other consideration is that the robot arms and rails are all metal structures that create a harsh operating environment for RF due to reflection. Packet loss is severe, and the problem is exacerbated by the mobility. To address this problem, we log data onto a flash memory card for later transmission or examination. One problem was that a popular open-source FAT file system was unable to log more than 35 samples per second in our experience, slower than what we needed by 1-2 orders of magnitude. Therefore, a specialized, optimized file system that matches the access pattern of the WSN application is needed.

SenSys'10, November 3–5, 2010, Zurich, Switzerland.

1.2 Requirements

Our WSN platform is targeted mainly to MCUs with a limited number of general-purpose I/O (GPIO) ports and a relatively small amount of on-chip code and data memory. Such platforms typically include on-board sensors to collect data and exchange data through RF. The MCU contains common interfaces such as UART, ADC, I^2C, and SPI. External nonvolatile memory such as serial flash and Micro-SD card can be connected to the sensor device through SPI. In the following subsections, we list the requirements of our WSN OS design.

1.2.1 Lightweight and Portability

The OS should be lightweight and portable enough to run on more resource-constrained wireless sensor platforms. Low memory and power consumption must be achieved by utilizing only limited resources in order to increase the life time of wireless sensor devices. A portable interface and the reduction of assembly code implementation enable the OS to be ported to different MCUs.

1.2.2 Programming Model

An appropriate programming model not only facilitates software development but also promotes good programming practices. Event-driven programming model is widely used in WSN but can be unstructured and difficult to use [3]. Consequently, an easy-to-use threading structure that is suitable for event-based WSN applications with low runtime overhead is desired. Context switching and scheduling are the two main sources of run-time overhead of multi-threaded programming and must be efficient.

1.2.3 Virtual Memory

Virtual memory (VM) can overcome the memory shortage problem that a resource-constrained wireless sensor platform usually faces. VM on an MCU without hardware MMU support can be achieved via compiler assistance, where each function is compiled into a code segment that is then stored on the Micro-SD card at a specific virtual address. The segment is loaded on demand while the user program calls this virtual address at run-time. Memory compaction and garbage collection must be implemented to solve the external fragmentation problem and to recycle unused memory for future allocation. Users should be able to write programs without concerns about what the run-time demand segmentation module does, and they should not have to insert additional tags or face restrictions on function prototypes in the user programs.

1.2.4 Remote Reprogramming

To enable run-time firmware updates for deployed wireless sensor devices, wireless reprogramming is required in WSN OS design. Dynamic loading must be enabled for partial update instead of whole image update in order to reduce the time and energy cost due to remote reprogramming.

1.2.5 File System

A lightweight file system supported by the operating system is necessary to assist with the access to data storage and VM. The desired file system is targeted to WSN applications instead of general-purpose computing. The API provided to users must be specifically designed and optimized according to the different access patterns. The file system must be configurable, and therefore only the necessary storage types are chosen in order to conserve code memory.

1.3 Approach

To meet the requirements listed above, we propose Enix, a lightweight and dynamic OS with a specialized file system called EcoFS for tightly constrained WSN platforms. The most important task for our work is to manage the limited resources on WSN platforms and to provide a suitable programming model for WSN application developers.

Our approach to the programming model is to adopt a cooperative threading model to enable multi-threaded programming while minimizing overhead of context switching and reducing code size based on a modified *setjmp/longjmp* system library. A replaceable scheduler is provided to support different scheduling policies that meet the requirements of different WSN applications. The efficiency of the scheduler is improved by fast algorithms instead of linear search for the next running thread; the overhead of context switch can also be reduced by storing the critical registers on the internal stack and avoiding using the slow external memory.

One of our approaches to overcoming resource constraints is compiler-assisted run-time support for features such as virtual memory, remote reprogramming and memory allocation. *Compiler assist* shifts the complexity to the resource-unconstrained part such as the host PC. Virtual memory can be achieved without hardware support via code insertion at compile time. Position-independent code (PIC) is a suitable approach to achieving dynamic loading; the compiler can assist with the generation of PIC on the host PC, thus avoiding the overhead of relocating addresses at run-time. With the host PC's help, the run-time overhead on the wireless sensor nodes can be reduced significantly.

For EcoFS, to achieve efficient file operation and low power consumption, we support four types of data: binary *code data*, *preferences* of sensor devices, *network data* such as routing tables, and *sensed data*. We minimize the active time of the Micro-SD card by I/O scheduling and by limiting the access via the API provided by EcoFS. To reduce code size, we let the controller inside the Micro-SD card handle wear leveling (i.e., evenly distributing writes among pages) and erase-before-write. We also make components in Enix configurable, and therefore only the required components are configured and compiled before programming the sensor devices. The code size of our fully functional implementation of Enix is 8 KB, and the data size is about 512 bytes. By excluding the unused components of Enix, the code and data memory consumption can be further reduced.

The rest of this paper is organized as follows. Section 2 discusses related work on WSN OS design. Section 3 presents an overview of Enix and its design concepts. Section 4 describes the Enix kernel, including the scheduler, compiler-assisted virtual memory, dynamic loading, and run-time reprogramming. Section 5 describes EcoFS and the chosen storage medium. In Section 6, we evaluate all these features of Enix, including a comparison of kernel performance with μC/OS-II, an embedded OS that supports the 8051 ISA and is widely used in industry for many applications. Finally, Section 7 concludes this paper with a sum-

Table 1. Market share of Microcontroller ISAs [13]

Co.	Core	Sh.	Co.	Core	Sh.
Intel	8051	19%	NEC	V850, 78K0, K3/K4	9%
Renesas	740, H8/S, M32R	17%	ST	Proprietary 8-b	6%
Freescale	68XX	15%	Atmel	AVR	3%
PIC	PIC	12%	Infineon	C16X	3%
ARM	ARM	10%	Others	Others	6%

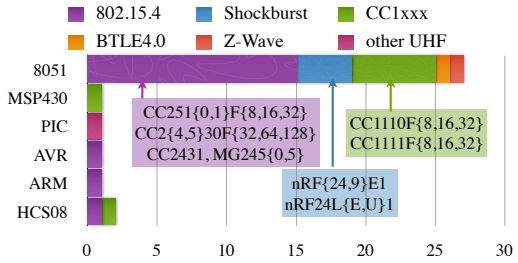


Figure 1. Number of RF+MCU options for popular ISAs.

mary of contributions and discusses directions for future research.

2 Related Work

Wireless sensor networks are composed of sensor nodes and base stations. A sensor node is typically small in physical size, low cost, small in code and data memory, and limited in computing capability and battery energy. Table 1 lists the instruction set architectures (ISA) of the most popular MCUs. Some real-time operating systems (RTOS) that are able to run on such a resource-constrained platform include μC/OS-II [23] and FreeRTOS [31], both of which support preemptive multi-threading with round-robin or priority-based schedulers. These RTOSs are indeed lightweight and well designed, but they are not suited for wireless sensor networks due to the lack of support for features such as runtime code update, power management, and resource control capabilities. In response, researchers have proposed WSN OSs that provide support specifically for WSN applications. Table 2 shows a comparison between existing OSs aimed at WSNs. However, it seems ironic that most WSN OSs to date are for either Atmel (3%) and MSP430 (a fraction of 6% "Others"), leaving the most popular 8051 (19%) and most other ISAs unsupported. When considering integrated RF-MCUs, the number of options for the 8051 dominates all other ISAs combined by 4:1, as shown in Fig. 1.

2.1 Programming Model

TinyOS [25] is a widely used runtime system for WSNs. It uses a special language called nesC [14] to describe the software components that form a sensor system with event-driven semantics. The application code and the runtime library are then compiled into one monolithic executable. Several event-driven runtime-support systems have been developed for wireless sensor networks and applications with similar characteristics, including SOS [17] and Contiki [11]. The processes of these OSs are implemented as event handlers that *run to completion* without preemption. Therefore, event handlers in event-driven models may share the same stack to reserve insufficient memory space. Event-driven programming is based on cooperative multitasking, which may be good for tiny, single-processor embedded devices, but users have to perform stack management manually, and as a result, the code can become difficult to read and maintain. Some WSN OSs provide preemptive multi-threading [3–5, 11, 15, 20]. Multi-threaded programming models can be easier to learn compared to event-driven ones, and the code can be more readable and maintainable. In severely memory- and power-constrained environments, however, a multi-threaded model has several disadvantages. For example, it occupies a large part of the memory resources, spends more CPU time and consumes more battery energy due to context-switching overhead.

Preemptive multi-threading extensions [9] to TinyOS have been proposed to avoid long-running tasks. Various concurrency methods has been considered, including preemptive tasks, fibers and virtual machine threads. Later, TinyOS v2.x incorporated the multi-threading concept in TOSThreads [20]. They consist of kernel threads and application threads. Only application threads can be preempted, while kernel threads are non-preemptible since they are for blocking system calls. Kernel threads are activated only through message passing.

Protothreads [12] allow users to write event-driven programs in a threading style, with the memory overhead of two bytes per Protothread. The concept is similar to C co-routines [21] in that they implement "return and continue" by using C-switch expansion. Protothreads are limited in that auto variables or local states cannot be stored across a blocking wait, and the C-switch implementation may lead to increased code size in the form of a table lookup and a jump, which also incur overhead at runtime. The runtime complexity is proportional to the number of yield points in a Protothread.

Our Enix OS emphasizes cooperative threading, which guarantees that no thread will have to yield control unexpectedly [16], and Enix threads work similarly to co-routines but they are implemented in a mix of C and assembly for smaller code size and better execution efficiency. Swapping between threads in Enix is a real context-switch operation, not just a C-switch. It provides automatic stack management and incurs little runtime overhead compared to preemptive multi-threading. In addition to cooperative threading, Enix also supports lightweight preemptive multi-threading for real-time scheduling. For the power consumed by context switches, it has been shown for MANTIS OS that multi-threading and energy efficiency need not be mutually exclusive when an effective sleeping mechanism is used to reduce context-switching overhead [3, 9].

2.2 Runtime OS support for WSN

In real-world wireless sensor networks, the deployed sensor nodes must have the abilities to manage the tasks and resources at run-time. Run-time reconfiguration and reprogramming also become important issues in WSN OS design. TinyOS, as mentioned before, produces a single image in which the kernel and applications are statically linked. Thus, updating code means whole-system image replacement. To support efficient runtime remote reprogramming for TinyOS, a virtual machine named Maté [24] has been proposed. Using Maté or other virtual machines for WSNs [22, 28], code

Table 2. Comparison between WSNs operating systems.

OS	Enix	TinyOS	SOS	Contiki	MANTIS OS	t-kernel	RETOS	LiteOS	Nano-RK
Platform	Nordic nRF24LE1	ATmega128L &MSP430	ATmega128L	ATmega128L & MSP430	ATmega128L	ATmega128L	ATmega128L & MSP430	ATmega128L	ATmega128L
Programming Model	Thread	Event	Event	Event & Thread	Thread	Thread	Thread	Thread	Thread
Real-time Support	△ [1]	△		◯	◯	◯	◯	◯	◯
Dual Mode Operation[9]	◯		◯	◯	◯	◯	◯	◯	
Remote Update	◯	△	◯	◯	◯		◯	◯	
Dynamic Loading	◯ [2]	△	◯ [2]	◯ [3]			◯ [3]	◯ [3]	
Protection[10]		△				◯			
Virtual Memory	◯ [4]	△				◯ [5]			
File System	◯	△						◯	
Network Abstraction[11]	◯	△					◯		◯
Code Size (Bytes) [6]	8,138	20,924 [8]	20,464	3,874	14,000	28,864	20,394	30,822	10,000
Data Size (Bytes) [7]	512	597	1536	512	512	512	945	1,633	2,000

[1] △ means optional components.
[2] achieved using PIC.
[3] achieved using runtime relocation.
[4] supports code virtual memory only.
[5] supports both code and data paging.
[6] The code size including the basic kernel and the ◯ components, excluded △s.
[7] lists the smallest data memory required to startup OS, at least one thread in multi-threaded model.
[8] TinyOS code size is compiling from v1.1.15, with various sensor drivers and a network module, but excluding storage system and remote programming.
[9] means the kernel code is not writable by user code
[10] threads are protected from each other's access
[11] OS provides layers of API

can be distributed and configured at run time. The drawbacks of running a virtual machine on a sensor node include runtime overhead of the virtual machine interpreter and higher energy consumption.

SOS [17] is another event-driven OS but consists of dynamically loaded modules and a common kernel. The modules are position independent code (PIC) binaries that implement specific tasks or functions. This modularized design is quite flexible, but the interaction between modules may incur high runtime overhead, and the module must be implemented in a fixed format, which increases the overall code size on SOS.

Contiki [11], RETOS [5] and LiteOS [4] provide dynamic loading by runtime relocation, rather than relying on position independence. The application binary to be combined with relocation information must be relocated or linked at runtime [10] before loading into the program memory. Thus, a sizable data buffer is required in order to relocate in space.

Our Enix OS supports dynamic loading using kernel-supported PIC, which is easy to port to other platforms. The dynamic library is pre-linked to the kernel with minor modification. Hence, our runtime overhead and additional buffer are reduced compared to the runtime relocation approach.

2.3 Virtual Memory

To fully utilize the memory of a tightly constrained wireless sensor platform, some researchers propose software virtual memory on MMU-less embedded systems. One approach is code modification by either using a compiler or constructing an additional converter. SNACK-pop [30] provides a framework for compiler-assisted demand code paging with static call graph analysis and optimization. The input Executable and Linking Format (ELF) [19] file is analyzed and translated into an executable image such that every call/return is modified to call the page manager. Choudhuri and Givargis [7] showed that data segments have a greater need to be paged than code segments, and therefore they propose a data paging scheme with an adjustable page size based on an application-level virtual memory library and a virtual memory-aware assembler. The t-kernel [15] is the first WSN OS that provides virtual memory for both code and data segments with additional memory protection. Besides software virtual memory approach, MEMMU [2] proposes a new software-based on-line memory expansion technique [1] that requires no secondary storage. Therefore, it improves performance and minimizes power consumption comparing to the above approach. However, MEMMU introduces about 4 KB of code size overhead and requires at least 512 bytes of data memory. Besides dynamic loading, Enix also supplies software segmented virtual memory by code modification, and uses a Micro-SD card as secondary storage for the convenience of installing virtual code segments using a host PC without requiring a specialized programmer. Enix does not provide virtual memory for data segments because of the high runtime overhead, but it does support a data memory allocation scheme to fully utilize the limited data memory. Furthermore, VM support in Enix consumes only 886 bytes of code memory and 256 bytes of data memory.

2.4 File Systems for WSN

LiteFS, a subsystem of LiteOS [4], provides a hierarchical, Unix-like file system that supports both directories and files. General-purpose file systems such as FAT, Ext2 may be interoperable but may be unsuitable for WSNs due to the relatively large code size and a great deal of data memory space used to store the hierarchical data structures. A more severe problem is the poor performance: in our own experience, a popular FAT file system was able to log at most 35 samples per second on a Telos-class node. Another issue is that the general file format becomes insufficient to store WSN data due to the absence of fast query support.

Some file systems or databases have been customized for WSNs. ELF [8] uses NOR flash to implement a log-structured file system. MicroHash [33], TinyDB [26] and FlashDB [29] use a lightweight index structure or database that works on wireless sensor nodes. Due to supporting high-performance indexing and searching capabilities, overhead of in-memory data structures is inevitable on these systems.

The Coffee file system adopted in ContikiOS is a log-structured, flash-based file system [32]. Capsule [27] cov-

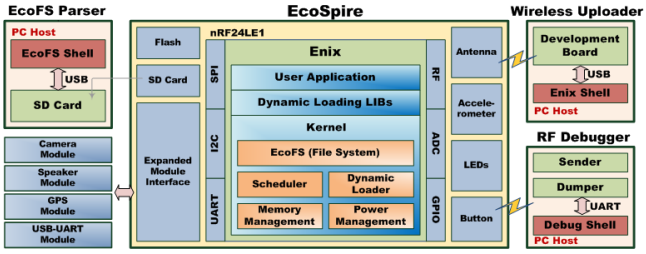


Figure 2. The block diagram of Enix.

ers the abilities mentioned above. It provides the abstraction of typed storage objects to applications, including streams, indexes, stacks, and queues. The composition of different objects may satisfy various requirements of WSNs. However, the high capacity parallel NAND flash used by Capsule makes use of many general-purpose input/output (GPIO) ports, and therefore it does not work on small-sized MCUs with few available I/O ports.

3 Overview of Enix

Fig. 2 shows the block diagram of Enix, which currently runs on a wireless sensor platform called EcoSpire [6]. The Enix OS contains four major components: runtime kernel, file system, dynamic loading library, and utility tools. Although the file system is one of the modules in the runtime kernel, it provides more support for WSN applications. Consequently, we separate the file system as an individual component of Enix.

3.1 Runtime Kernel

The first component of Enix is the *runtime kernel*, which manages hardware resources and supports run-time reconfiguration. With a *cooperative thread scheduler*, developers can write multi-threaded programs instead of being limited to single-threaded programming. A *wireless code image update manager* and a *dynamic loader* are also included to enable remote reprogramming, so that the deployed sensor nodes can be easily updated over RF.

3.2 File System

EcoFS, the file system of Enix, is a configurable and lightweight storage system using a Micro-SD card or an MMC card as the storage medium. Generic file abstraction may not be well suited for WSNs. We realized that only few types of data need to be saved, and we divide EcoFS according to the different usage patterns: *code data*, *preferences*, *network data*, and *sensed data*. The code data block stores the binary code segments that can be loaded efficiently by the runtime loader into the code memory on demand. The preferences are the key-value pairs for storing the node's status and settings. The requirement of preferences is fast searching with modifications enabled. Because of the limited data memory and the wish to provide a simple network abstraction, the routing table may be maintained in EcoFS. Network data such as routing tables or the roles of the adjacent nodes must be enumerable and may be changeable if the network topology is dynamic. Last but not least, the sensing data block is used for logging collected data in harsh environments when real-time transmission is not viable. This append-only storage type must be fast, power-efficient, and reliable to meet the requirements of high-sampling-rate WSN applications.

3.3 Dynamic Loading Library

The third component of Enix is the dynamic loading library called ELIB. ELIB is a special library that is preprocessed by the host PC tools. Most of the commonly used library functions are collected and transformed into *segments* that comprise ELIB. Each segment corresponds to the position-independent code of a function with a unique virtual address, namely where the segment is located in the secondary storage. Due to the constrained memory resource of compact wireless sensor devices and the high energy consumption of the RF transceiver, making use of ELIB enables software virtual memory and reduces the transmission size of runtime reprogramming. A full ELIB library now takes 28K (12KB without standard C library).

3.4 Utility Tools

The last part of Enix is the host PC's *utility tools*, including a wireless reprogramming and debugging shell, a file system parser shell, complier and linker. With the host PC's assistance, heavier computation and compiling tasks can be done on the host PC to reduce the runtime overhead on tightly constrained nodes. Rather than processing delayed linking at run-time as ELF does, the ELIB building tool constructs a position-independent library at link-time. Therefore, the run-time loading burden on the MCU of these compact sensor nodes is reduced significantly, and no additional code is required. Another tool is the file system shell, called the *EcoFS Shell*, which provides an interactive environment for users to manipulate specially formatted data of EcoFS simply and conveniently. This assistance saves the user from wasting time on troublesome tasks such as reading, modifying, and writing secondary storage devices directly.

3.5 Code Size

The non-swappable part of the system takes around 8KB of compiled code, including the thread scheduler, low-level drivers for SD and RF, file system driver, remote programming, the main function, and some basic library needed by above objects. Other utility library code can be saved in ELIB on the SD card and are loaded on-demand. A detail breakdown for Enix compiled code is as follows:

	Enix Scheduler	1023	
	Enix Storage and RF driver	1696	
	EcoFS	2316	
	Remote reprogramming	660	
	Essential C library	1130	
+)	User logical structure	203	
	Total	7028	bytes

4 Runtime Components in Enix

This section describes the kernel components in Enix. They are the cooperative threads with the run-time scheduler, compiler-assisted virtual memory, and dynamic loading and runtime reprogramming support.

4.1 Cooperative Threads and Scheduler

The *cooperative threading* model has the characteristic that a context switch occurs only when the current running thread calls a `yield` or `sleep` function. Thus, the overhead of

context switching and stack usage are lower than that of preemptive multi-threading and is more appropriate for tightly constrained wireless sensor platforms. Here, we describe an efficient way to implement cooperative threads with low context-switching overhead and that consumes little code and data memory. In addition, two popular scheduling policies, namely priority-based and round-robin, are also presented to provide the adaptive abilities of Enix for supporting different WSN applications.

4.1.1 *Multi-points Setjmp/Longjmp*

The system calls `setjmp` and `longjmp` are commonly used for jumping between subroutines. To achieve the inter-subroutines jump, the `setjmp` function stores the program counter and stack pointer into a jump buffer. When `longjmp` function is called from another subroutine, the data in the jump buffer is restored, and then the program returns to the previous `setjmp` point. It is worth mentioning that the used registers are pushed on the stack before the `setjmp` function is called, and therefore the local variables can also be restored after `setjmp`. However, this approach does not support jumping forward or backward between multiple functions as required by coroutines. First, it provides a single-direction jump only from the `longjmp` point to `setjmp` point according to the jump buffer; second, stack overwriting happens while the previous `setjmp` point calls the cascaded function that may modify the stack data of later `setjmp` points.

To achieve cooperative threading, the ideas of `setjmp` and `longjmp` are taken. Each never-returned subroutine represents a cooperative thread, which has its own stack and context buffer for storing critical data. A cooperative thread may yield at any specific point to invoke the scheduler and resume another cooperative thread. Each yielding call automatically pushes the local variables on the stack and records the program counter and stack pointer in the context buffer. The resuming process simply restores the saved data from the context buffer and the stack. As long as the context-switching point is determinable, only necessary data will be stored on the stack. Therefore, the per-thread stack does not require much memory and is adaptive to the number of threads. The maximum number of cooperative threads is seven in the current implementation of Enix. This approach to cooperative threads allows subroutines to suspend and resume execution at specific locations without being concerned with stack and thread states. Moreover, semaphores and functions such as `yield`, `sleep`, `suspend` and `resume` are also provided to enable thread control. These primitives are intended for users familiar with threads programming and their implications. For more advanced applications where the execution time may be data dependent, such as compression algorithms, proper selection of the yield points will be important, or else the system may prevent other threads from execution and decrease system reliability. One standard solution is to use timer watchdogs to regain control from misbehaving threads; another is to enable preemptive multi-threading support.

Fig. 3 shows a sample application written in the cooperative threading model in Enix. This application is to sense triaxial acceleration and then wirelessly transmit the sensed data to a receiver with ID 1234. There are three cooperative threads in this program: `thread0` first initializes the hardware modules and global variables and then blinks the LED periodically; `thread1` senses the data while the sensor is ready and the previous sensed data has already been transmitted; `thread2` transmits the sensed data to the remote sensor node and then clears the global flag to enable the next sensing task in `thread1`. The `main` function adds the threads to the Enix kernel and then calls `enix_kernel_start` to invoke the scheduler, which never returns to `main()`.

4.1.2 *Priority-Based and Round-Robin Schedulers*

Enix provides two scheduling policies: priority-based scheduler and round-robin scheduler to determine the next thread to run based on the thread's priority or registration sequence, respectively. The scheduling policy in Enix is replaceable to provide flexibility for development of WSN applications.

The list of runnable threads is represented as an array of bitmaps to enable quickly finding the next runnable thread. The thread with priority n is runnable only if the n^{th} bit in the bitmap is set; thus, by checking the bitmap, the next running thread can be found. For the priority-based scheduler, the first set bit in the bitmap indicates the next running thread. More powerful ISAs such as ARM support instructions for finding the first set bit in a 32-bit word. Simpler ISAs such as 8051, AVR, or MSP430 do not support such powerful instructions. So, we use a table-lookup implementation. The number of threads is limited by the size of the bitmap.

Algorithm 1 shows the pseudo code for finding the highest-priority runnable thread by table lookup, where *nextPrioTbl* is a byte array with 16 elements, each of which indicates the index of the first set bit in the corresponding element of the array, and *rdylst* is a bitmap list. Each bit represents a cooperative thread with a different priority. By looking up the index of the first set bit, the next running thread can be found as shown in the algorithm.

To implement a round-robin scheduler, we propose another efficient table-lookup algorithm (Algorithm 2). By rotating the *rdylist* to the right for *currentPrio*+1 bits and looking up the table, which is the same as Algorithm 1, the next running thread can be easily found. These two algorithms consume additional 16 bytes of code memory for the immutable table but reduce the total code size and improve the performance significantly, as will be shown in Section 6.

Algorithm 1 Fast algorithm to get the next running thread (Priority-Based).

```
if nextPrioTbl[ rdylst & 0x0F ] ≠ 4 then
    return nextPrioTbl[ rdylst & 0x0F ]
else
    return 4 + nextPrioTbl[ rdylst >> 4 ]
end if
```

4.2 Compiler-Assisted Virtual Memory

Virtual memory is widely used in OSs to support a larger memory space than provided by the physical memory. In this section, we describe the implementation details of compiler-assisted virtual memory in Enix.

4.2.1 *Demand Segmentation*

Virtual memory is achieved in Enix via demand segmentation without any hardware support. The code memory

```
extern xdata char* malloc_ptr_1;
extern unsigned char gb;
//thread 0 control LED and init everything
ENIX_THREAD(thread0) {
   EA = RF = 1;  //init RF
   //init 3-axes
   epl_acc_init(ACC_8G_SCALE,
                ACC_DATA_RATE_100HZ);
   //malloc 3 bytes
   malloc_ptr_1 = (xdata signed char *)
                  eco_kernel_mem_req(3);
   gb = 0;  //init flag
   while(1) {
      LED0 ^= 1;
      LED1 ^= 1;
      enix_kernel_thread_sleep(10);
   }
   ENIX_THREAD_END();
}

//thread 1 sensed data from 3 axes
ENIX_THREAD(thread1) {
   while(1) {
      if (gb || !epl_acc_data_is_ready())
         break;

      malloc_ptr_1[0] = epl_acc_read_X();
      malloc_ptr_1[1] = epl_acc_read_Y();
      malloc_ptr_1[2] = epl_acc_read_Z();
      gb = 1;  //set flag
      enix_kernel_thread_sleep(1);
   }
   ENIX_THREAD_END();
}

//thread 2 transmit sensed data
ENIX_THREAD(thread2) {
   pdata unsigned char *packet;
   packet = enix_kernel_get_tx_buf();
   enix_kernel_rf_start_tx(1234);

   while (1) {
      if (gb) {
         packet[0] = malloc_ptr_1[0];
         packet[1] = malloc_ptr_1[1];
         packet[2] = malloc_ptr_1[2];
         enix_kernel_rf_send();
         gb = 0;  //clear flag
      }
      enix_kernel_thread_sleep(1);
   }
   ENIX_THREAD_END();
}

int main() {
   LED0 = LED1 = OFF;
   //add thread
   enix_kernel_add_thread(0, ENIX_DEFAULT_INIT,
                          thread0, LOW_POWER_ON);
   enix_kernel_add_thread(1, ENIX_DEFAULT_INIT,
                          thread1, LOW_POWER_OFF);
   enix_kernel_add_thread(3, ENIX_DEFAULT_INIT,
                          thread3, LOW_POWER_OFF);
   //run kernel, never return
   enix_kernel_set_timer_period(0);
   enix_kernel_start();
   return 0;
}
```

Figure 3. An example Enix WSN application code: Sense and Transmit.

Algorithm 2 Fast algorithm to get next running thread (Round-Robin).

```
if rdylst = 0 then
    return 7
end if
t ← RIGHTROTATE(rdylst, (currentPrio + 1))
r ← bitmap_lookup( t ) ▷ pass t as rdylist to Algorithm 1
x ← r + currentPrio + 1
if x > 7 then
    return x − 8
else
    return x
end if
```

of the wireless sensor platform is divided into swappable and non-swappable areas. The Enix kernel and the user-defined logical structures such as the `main` function are non-swappable. The swappable area utilizes the rest of code memory managed by the virtual memory manager of Enix.

To reduce the runtime overhead in Enix, a library called ELIB is proposed. ELIB consists of binary code segments that are preprocessed on the host PC and is pre-installed on the Micro-SD card, the secondary storage medium natively supported in Enix. Each segment in ELIB represents the binary code of a function in a position-independent way. That is, a unique virtual address is assigned to each code segment to indicate its location in the Micro-SD card. Calls to ELIB functions from the user program will be translated at compile time into calls to a special run-time loader routine in Enix kernel using a technique called *source code refinement*, to be described in the next section. Therefore, the demanded segments will be loaded into code memory and executed at run-time. The current memory allocation scheme in Enix is first-fit.

It takes three passes to construct ELIB. In the first pass, the common functions are collected into a file named `ELIB.LIB`, which is passed to the library parser to get the binary code size of each function; and then a *virtual-address allocator* is called to allocate a unique virtual address to each function. The second pass is *code modification*: a library function may call another library function that does not exist in code memory, and therefore such code must be modified. The purpose of code modification is to translate ELIB functions to run-time position-independent code. When the code modification is done, the ELIB is compiled and linked to create the file named `ELIB.HEX`, the hex image that consists of the HEX representation of ELIB functions. The final pass splits `ELIB.HEX` into separated HEX files, each of which is called a *code segment*, that is, the HEX representation of a function. Next, the EcoFS installer program installs the code segments onto the Micro-SD card according to their virtual addresses. The procedure above can be done automatically by a shell script. After this procedure, the Micro-SD card is available for loading and executing.

4.2.2 *Memory Compaction and Garbage Collection*

All virtual-memory systems have the common problem of *fragmentation*. *External fragmentation* occurs when the remaining free memory blocks are not consecutive. Since the dynamic loading library ELIB is runtime position-independent, fragmentation can be solved by memory compaction. A garbage collector routine executes periodically to observe the memory usage and compact memory if necessary.

Another problem arises when the garbage collector reclaims those segments that will be executed after the return of the current running segment. This is called the *cascaded call* problem: it occurs when the code memory is out of use, and the caller is garbage-collected while the callee is executing, such that the callee returns to the caller that has been evicted, thereby causing the system to crash. There are some solutions to fix the cascaded call problem. For example, additional checking code can be inserted before callee returns, and thus the absent caller would be reloaded back before it is returned to. Enix solves this problem by restricting the garbage collector to only non-swappable code.

4.3 Dynamic Loading and Run-time Reprogramming

Run-time reprogramming and dynamic loading are important issues in WSN OS design. Deployed sensor nodes need remote reprogrammability for the purpose of bug fixing and firmware updating. Enix supports run-time repro-

gramming and dynamic loading, and user programs can be updated through RF.

4.3.1 Run-time Position-Independent Code

To achieve fast runtime loading, Enix uses Position-Independent Code (PIC). Traditional PIC is machine dependent, and thus not every architecture supports PIC. We propose an efficient way to generate PIC code without hardware support. With the assistance of run-time kernel via code modification, the code becomes position-independent at runtime. This modification is also applied to the function segments in ELIB as mentioned before. Fig. 4 shows the code modification details in Enix. Three types of code are position dependent in general Enix user programs.

First, kernel calls in the user program do not have to be modified, due to the separation between the kernel and user programs. The linker redirects these kernel calls to the appropriate addresses by a linker script as shown in Fig. 4(a).

The second type of position-independent code covers the library calls as shown in Fig. 4(b). We redirect this kind of absolute calls to the special kernel function via code modification. The kernel function finds the address of target function at run-time and then jumps there. If the target does not exist, then the loader is invoked to do the same work as mentioned in the virtual memory section.

The last type is for local absolute jumps, as shown in Fig. 4(c). In most cases, local jumps are relative jumps except for those jumping from the beginning of a large logical structure to the end. We convert a long jump instruction into a relative jump routine by calculating the relative size from the source to the target at compile time, get the current program counter at run-time, and then add the program counter and the relative size to get the target address at run-time.

In addition, Fig. 4(d) also shows two key functions in the Enix kernel to support run-time PIC. The function `enix_getpc` gets the program counter at run-time and stores it into the *DPH1*:*DPL1* register pair (an alternative pair of the Data Pointer High/Low registers in the 8051 ISA). Since the *lcall* instruction will push the return address onto the stack, we can call `enix_getpc` to read the program counter from the stack. `enix_fake` is another kernel function that checks the existence of the target function and redirects the *lcall* instruction to the target function address at runtime. If the target function does not exist in the code memory, then it will be loaded from the Micro-SD card according to the virtual address passed into `enix_fake` function.

4.3.2 Source Code Refinement

In order to hide the details of the run-time loader from the flow of user program development and to reduce the amount of code modifications, a source code refinement technique is applied. Fig. 5(a) shows a simple Enix user application sending a string of data through UART. For the include files of Enix user program, the function definitions are removed, and C macros are applied as shown in the Fig. 5(a). By using macros and *varargs.h* support for the C language, each library function call in the user program is redirected to a special function named `enix_fake`. This vararg-style function allows variable numbers and types of parameters, and therefore every function prototype can work with this refinement with additional type casting. Furthermore, the source code

(a) Kernel Call (unchanged)

```
enix_fs_get_pref("LPOW") ⇒ enix_fs_get_pref("LPOW")
```

(b) Library Call: mapped `enix_fake`

```
epl_uart_putchar('a') ⇒ enix_fake(0x1220,(uint8_t)'a')
```

(c) Local LJMP: converted to relative jump

```
__asm
  ljmp __300107
__endasm;
⇒
__asm
  lcall _enix_kernel_rd_getpc
  mov   DPTR,#(__300107 - .)
  mov   A, _DPL1
  clr   C
  add   A, DPL
  mov   DPL, A
  mov   A, _DPH1
  addc  A, DPH
  mov   DPH, A
  clr   A
  jmp   @A+DPTR
__endasm;
```

(d) Enix Kernel: support routines

```
void enix_getpc() {
  __asm
    mov  A, R0
    mov  R0, SP
    mov  _DPH1, @R0
    dec  R0
    mov  _DPL1, @R0
    mov  R0, A
    ret
  __endasm;
}
```

```
uint16_t enix_fake(
  unsigned int addr,...){
  __asm
    clr   _EA
    push  _ALL
    ; get virtual addr
    mov   R0, _SPX
    dec   R0
    movx  A, @R0
    mov   DPH, A
    dec   R0
    movx  A, @R0
    mov   DPL, A
    mov   _SPX, R0
    lcall _enix_rd_cr
    ; restore XSP
    inc   _SPX
    inc   _SPX
    POP   _ALL
    setb  _EA
    ret
  __endasm;
}
```

Figure 4. (a)-(c): Transformation and (d) support for position independent code.

refinement technique can be used to achieve system configuration. For example, a general library function `SendPacket` can be called by the user program to send a byte buffer through different interfaces, such as RF, UART, I^2C and SPI. It is easy to provide a configuration interface for users to choose the transmission interface of `SendPacket` library function by using the source code refinement technique. Hence, different hardware modules of wireless sensor devices can be easily configured.

5 EcoFS: The File System

A lightweight and efficient storage system is essential to ultra-compact sensor nodes that must log all sensed data for later analysis or asynchronous transmission. Nonvolatile storage is also important for recording node state and time stamps of events, especially since power depletion may occur unpredictably on a deployed sensor node. Although the developer can write drivers to directly control the nonvolatile storage, doing so is tedious, error prone, and unstructured. A more structured approach is to build a simple file system abstraction on top of the raw storage device. We propose a file system named EcoFS as a component in Enix.

5.1 Storage Medium in WSN

In recent years, flash memory has been widely used in embedded systems and handheld devices. The characteristics of flash memory include non-volatility, small size, low cost, low power consumption, and shock resistance. In fact, flash memory can be found on many popular wireless sensor platforms either on-board or as an expansion module. For example, our experimental platform EcoSpire can have an external Micro-SD card module connected via SPI. A Micro-SD card is functionally identical to a regular SD (Secure Digital) card

User Program

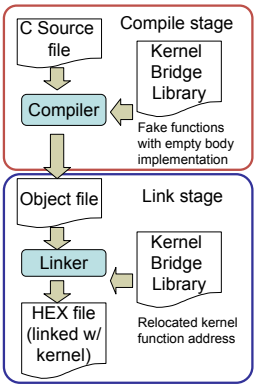
```
extern xdata char* malloc_ptr_1;
int main() {
    LED0 = LED1 = OFF;
    epl_uart_init(UART_BAUD_57K6);
    epl_uart_putstr("test uart\n");
    return 0;
}
```

Source Code Refinement

```
extern xdata char* malloc_ptr_1;
#define _EPL_UART_PUTSINT 11776 //size:145
#define _EPL_UART_PUTSTR  14336 //size:110
#define _EPL_UART_INIT    14848 //size:95

#define epl_uart_init(a) \
        enix_fake(_EPL_UART_INIT, \
        (unsigned char)a)
#define epl_uart_putchar(a) \
        enix_fake(_EPL_UART_PUTCHAR,\
                (unsigned char)a)
#define epl_uart_putstr(a) \
        enix_fake(_EPL_UART_PUTSTR, \
                (unsigned char*)a)
```

(a) Source code refinement.



(b) Bridge library.

Figure 5. Bridge library and Enix user programs.

but in a reduced form factor.

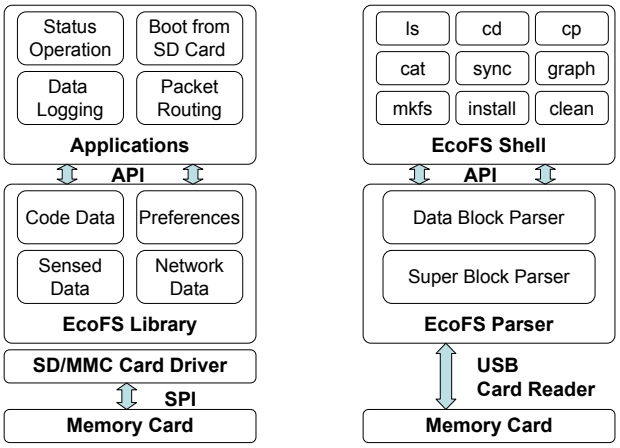
An SD card contains a controller that handles wear leveling, auto-erasing, and error correcting codes (ECC) recovering. SD/MMC cards can be controlled through a serial interface (SPI). Although SD cards consume more energy than raw flash components, their simple interface, small physical size, low cost and high capacity characteristics make them suitable for wireless embedded devices. Furthermore, the removable characteristic enables SD cards to be removed from deployed sensor nodes for later analysis instead of transmitting the logged data back through UART, USB, or RF. Considering the advantages mentioned above, we chose SD/MMC as the storage medium for EcoFS.

5.2 Host Side vs. Host Side EcoFS

The implementation of EcoFS consists of two main parts: node side and host side. The part on the host side is controlled using the *EcoFS Shell*. Due to the resource limitations, the node side focuses on how to efficiently access EcoFS data. On the other hand, the host side provides complete functionalities of EcoFS, including list, read, write, modify, and binary to HEX translation. There is no severe restriction on the host PC. Fig. 6 shows the block diagrams of EcoFS for both the node side and host side.

5.2.1 Node Side

For the wireless sensor node implementation shown in Fig. 6(a), an SD/MMC driver is built according to the Secure Digital Card specification that uses SPI protocol to access the memory card for basic I/O operations. An EcoFS library is provided in order to recognize specialized data types of EcoFS, namely code data, preferences, sensed data, and network data. The EcoFS Library is configurable: only demanded ones are configured and installed on the wireless sensor nodes. Some reference applications commonly used in WSN such as logging sensed data, booting from an SD card, periodically refreshing the node status, and accessing the routing table in multi-hop wireless networks can all be achieved easily by using EcoFS.



(a) EcoFS: Node Side (b) EcoFS: Host Side

Figure 6. The block diagrams of EcoFS.

5.2.2 Host Side

After the data has been logged on the SD Card, it is relatively easy to access the SD card via a USB card reader on a host PC as shown in Fig. 6(b). To extract EcoFS-formatted data, an EcoFS Parser is required. The raw data read from the card are processed by the parser that converts them into the appropriate file format to be accessed by users.

The top layer of EcoFS host-side implementation is EcoFS Shell. The interactive shell environment provides general Unix-like commands such as `ls`, `cd`, `cp`, and `cat` so that users can list the files for each type and see the content of files. There are also special commands, for example, `mkfs` to format an SD card, `clean` to eliminate all files, and `install` to modify and add files to EcoFS. Moreover, the command `graph` can be used to analyze sensed data and plot charts.

To demonstrate the convenient shell environment of EcoFS host side implementation, Fig. 7 shows the snapshots of EcoFS Shell. The users can easily manipulate the EcoFS files by plugging the Micro-SD card into a card reader via the assistance of EcoFS Shell on the host PC. The collected data by the sensors can be analyzed easily as shown in Fig. 7. In addition, all EcoFS-formatted data, including binary code segments, can be accessed with ease through EcoFS Shell.

5.3 Data Types in EcoFS

As mentioned in Section 3, each data item in EcoFS is either fixed length or wrapped by special tags as used in a regular TCP/IP packet format and has a special length field to indicate the size of the item. Therefore, the parsing process can be done without consuming large data memory by using load-partial-then-parse scheme. In the following subsections, we describe the detailed format of EcoFS data types.

5.3.1 Code Data

The purpose of code data is to support virtual memory for code in Enix. By using the EcoFS Shell, the dynamic loading library ELIB can be pre-installed on the Micro-SD card for later loading. Each code segment has a unique virtual address, which is the location of the code segment on the Micro-SD card. The segment format starts from a special BEG(0xAE) tag followed by a two-byte field indicating

the binary segment size; after the size field, the binary data bytes with a pre-defined size follow. When a node wants to retrieve the segment by its virtual address from the Micro-SD card, it first checks for the BEG tag; then it reads two bytes to get the data size and allocate suitable code memory space; finally, the code segment is loaded from the Micro-SD card into the code memory. The current design of the code data block does not allow modification of the code segment by the sensor node itself, though a later version of EcoFS will enable replacement of binary segments at run-time.

5.3.2 Preferences

To provide a fast query data structure of EcoFS, we added the preference data type. Each preference item is represented with 22 bytes, including 1 byte BEG(0xEA) tag, 1 byte TYPE tag that indicates the datatype of the value field such as character, integer, or string, following by the 10-byte key string and the 10-byte value with a type tag.

To prevent high data memory consumption and to support modification of data, each preference item is distributed into a 512-byte SD card sector. The characteristic of preference is to support fast searching and additional modifying ability. As a result, a hashing scheme is applied. When the specific value of a key is required, first the key is passed to a simple hash function to generates an integer. Second, the integer is added to an offset to get the sector number, the location of preference item. Finally, the value is retrieved. For the keys with same hash value, we allocate five slots to store the collision preferences; the next linked preference is located just adjacent to the current located sector. There is also a super block for data of preference type, which is the bitmap used to check whether the target hash value exists or not. Thus, the string-comparison time is reduced for non-existent preference keys.

5.3.3 Sensed Data

The sensed data is appended only when changing the "already sensed data" is unnecessary and the modification of flash memory results in high overhead. Some of the WSN sensing tasks collect sensed data when specific events have happened. For this reason, begin and end tags are added to enclose sensed data to keep events separated. There is also a timestamp field in sensed data format, so that the later analysis can calculate the event-trigger time using the data retrieved from this field. To fill the timestamp field, either Enix system timer or user application granted time can be used. Besides, if the Micro-SD card is full, then the decision to overwrite or stop depends on the preference setting.

5.3.4 Network Data

EcoFS network data are used to store network-related information such as packet routes, neighbor sensor nodes' states, and the WSN topology. The fields can be customized by user applications. Each network data item is represented by fixed 32-byte data, including 2 bytes of network ID and 30 bytes of a user-defined structure such as the type, state, and remaining power of sensor node, depending on the requirements of the WSN applications. A bitmap is maintained in the super block for the purpose of quickly enumerating existing items in the EcoFS network data area. The current version of EcoFS supports at most 65536 IDs, which consumes 8192 bytes. Even though a sensor node reads 8192 bytes of result in about 12 ms, this mechanism consumes only 32 bytes of data memory, which is more beneficial especially for constrained wireless sensor platforms. For example, we provide a sample WSN application called "receive and forward," which receives an RF packet, enumerates all the neighbor nodes' ID, and forwards the packet in sequence.

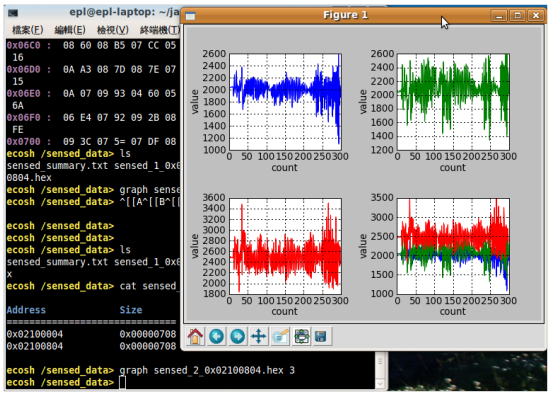


Figure 7. EcoFS host PC shell environment with built-in data analysis and data viewers.

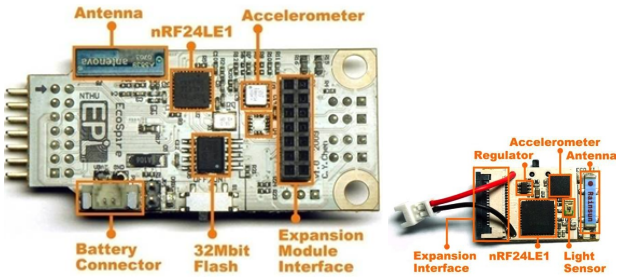


Figure 8. (a) EcoSpire node: 23×50 mm^2; (b) EcoSpire simple node, 13×20 mm^2.

6 Evaluation and Results

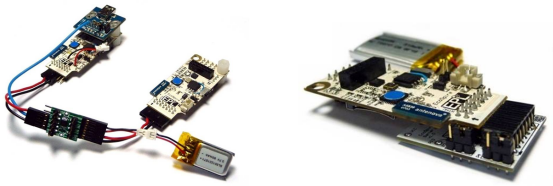
This section shows the efficiency of Enix as a lightweight dynamic WSN OS. We first describe our experimental setup, including our hardware platform and software tools. Second, we compare the context switch overhead and the code and data memory footprints of different multi-threaded scheduler implementations to show that the cooperative threads model of Enix has the lowest overhead. We also compare the power consumption of loading code segments over RF and Micro-SD cards. We evaluate our code updating scheme against other methods to show that we reduce runtime overhead significantly in terms of the size of the uploaded code.

6.1 Experimental Setup

This section describes the hardware platform and software tools for Enix.

6.1.1 Sensor Platform

EcoSpire [6] is our experimental platform. As shown in Fig. 8, EcoSpire refers to the larger version for prototyping and application development, and a compact version called the "simple node" is more suitable for field deployment. This platform consists of an MCU with an integrated RF



(a) Current sensor. (b) Battery fuel gauge.

Figure 9. Power measurement modules.

Table 3. Context switch overhead comparison between algorithms for selecting the next running thread. Unit: μs.

	RR	Priority-Based
Conventional Linear Check	48.925	55.2
Our Fast Table Lookup	16.325	11.9

transceiver, a chip antenna, a triaxial accelerometer, power circuitry, and an expansion interface. The MCU core runs at 16 MHz by default and comes with 16KB program flash and 1KB data RAM. EcoSpire's nRF24LE1 RF-MCU contains an on-chip radio that implements the Enhanced ShockBurst protocol, which is the core of the newly finalized Bluetooth 4.0 Low Energy Technology standard. A 32-Mbit onboard flash memory and Micro-SD expansion capability are included for nonvolatile data storage. This configuration is actually quite representative of many integrated RF-MCUs made by TI (CC2430, CC2510) and Z-Wave ones, all with integrated 8051 cores, and they comprise well over 90% of the RF-MCUs on the market. Most WSN OS cannot fit into this limited platform and are thus not easily comparable.

To measure the energy consumption of Enix, we developed two power measurement modules as shown in Fig. 9. One is the *current sensor module* for measuring the instantaneous current of another EcoSpire in execution. The other module is the *battery fuel gauge*, which allows EcoSpire to measure the power and battery capacity itself at run-time.

6.1.2 Software Tools

The software for EcoSpire includes an IDE and graphical user interface tools (GUI) on the host computer, system software on the sensor node and the base station, and utility tools for image uploading and RF debugging. We built our IDE with Eclipse by creating a plugin for EcoSpire development. With this plugin, we provide a fully GUI-based programming environment. This lowers the burden for programmers to memorize commands and enables them to focus on the software development process, from editing, compiling, and linking to firmware programming.

6.2 Context Switch Overhead of Different Schedulers

To evaluate the context-switch overhead of our cooperative threads in Enix, we implement both round-robin (RR) and priority-based schedulers with different algorithms that may affect the context-switching time.

Table 3 shows the results of using our fast table-lookup algorithms versus straightforward linear search to find the next running thread from the runnable queue. We measure the execution time by taking the average of 60,000 context switches. It is clear to observe that our fast algorithms significantly improve both RR and priority-based schedulers by cutting the execution times down to a quarter of the original linear search implementation.

Table 4. Context switch overhead comparison between different scheduler implementation. Unit: μs.

	RR	Priority-Based
C-Coroutines	16.325	11.9
ASM-Coroutines	8.25	8.475
Preemptive	13.45	23.3

Table 5. Context switch overhead comparison between Enix and μC/OS-II by averaging 60,000 context switches.

OS	Enix		μC/OS-II
policy	Coop.-th.	Preemp.-th.	default
Overhead (μs)	8.47	23.3	250

Table 4 shows a comparison of context-switch overhead between different multi-threaded models for both RR and priority-based scheduler. Preemptive multi-threading has the highest context-switch overhead due to the unpredictable preemption time, and therefore all of the registers must be saved and restored during context switch. We implement C-coroutines using C-switch statements and add the priority-based and RR schedulers to it. C-coroutines and cooperative threads have the feature that context switching occurs only when the running thread calls yield or sleep functions, and thus they have lower context-switch overhead. The implementation of C-coroutines uses C-switch statements, and this means that every context switch results in several comparisons of variables and an absolute jump. Consequently, a context switch of cooperative threads is simply a replacement of the program counter, stack pointer, and some global variables, and thus it has the lowest overhead.

To compare Enix with a real-world OS, we ported μC/OS-II to EcoSpire. It is widely used in industry, whereas no other WSN OSs are known to run on the 8051. Table 5 compares the context-switch overhead of μC/OS-II and our work. The reason why μC/OS-II has high context-switch overhead is that it uses external memory to store both the per-thread stack and registers, thus incurring great overhead from many external memory movements.

Table 6 shows a comparison of the code and data memory usage between different scheduler implementations. The preemptive ones consume the most memory for the same reason mentioned before. Other scheduler implementations require about 1KB of code memory, which is frugal compared to other regular RTOSs such as μC/OS-II and FreeRTOS, as shown in Table 7.

Table 6. Code and data size comparison between different scheduler implementation. Unit: bytes

Model	C-Coroutines		Cooperative		Preemptive	
Scheduler	RR	Prio.	RR	Prio.	RR	Prio.
Code Size	918	897	1140	1051	1688	1559
Data Size	79	79	70	69	97	91

Table 7. Code and data size comparison between Enix, FreeRTOS and μC/OS-II. Unit: bytes

	Enix(Scheduler)	FreeRTOS	μC/OS-II
Code Size	918	7560	10294
Data Size	79	719	488

Table 8. I/O speed comparison between flash chips. Units: Kbytes/s.

	SST25 VF512A	SST25 VF032B	Pm25 LV020	SanDisk MicroSD
Sequential read	170.7	170.7	176.6	170.66
Sequential write	46.5	71.1	54.8	92.75
Random read	46.4	39.4	43	2.18
Random write	20.9	23.1	17.3	0.11

6.3 Virtual Memory and EcoFS

This section shows the speed and power consumption of SD cards and other serial flash memories on EcoSpire. The results confirm the reason that the Micro-SD card was chosen for the main nonvolatile storage, as discussed in Section 5.

Tables 8 and 9 compare the speed and power consumption of a Micro-SD card with three other different on-board serial flash memories. These flash memories and the Micro-SD card are connected to EcoSpire through a common SPI bus. The SD card has fast sequential read and sequential write properties but poor random access speed due to the characteristics of NAND flash memory used by the SD card. Most of the routines in EcoFS use sequential reads and sequential writes such as code data and sensed data. For the other preferences types and network data, their access unit is a sector, and therefore the access time is equivalent to sequential access. Thus, the slow random access speed does not affect EcoFS. For the power consumption, the on-card controller of the SD card causes the highest power consumption among all flash memories. In fact, the data shown in Table 9 is the active power consumption of the SD card, that is, when the chip-select ($\overline{CS}$) signal is asserted. When $\overline{CS}$ is de-asserted, the power consumption of the SD card is low. Accordingly, the appropriate usage of the SD card may reduce the total energy cost of sensor nodes. EcoFS is designed such that once the SD card is selected by $\overline{CS}$, it will finish the I/O operations as soon as possible.

Fig. 10 shows the time and energy cost of the sensor node performing 1MB of sequential read with different block sizes. Due to the requirement of a start command before each sequential read and sequential write operation, the highest performance and lowest energy cost happen while the maximum block size of 512 bytes is applied. EcoFS tries to use the largest possible block size and reduce the number of random-access operations in order to overcome the power and speed bottlenecks of the SD card. Owing to the different approaches by SD card manufacturers, we have tried seven 2GB SD cards made by different manufacturers. We measure the speed and power consumption of sequential read and sequential write operations shown in Table 10. This comparison enables the user to make price/performance trade-offs.

We also evaluated the delay and energy of the virtual memory. They are 6.328ms / KB and 505.2μJ / KB, respectively.

Table 9. The energy consumption comparison between different flash chips. unit: mJ.

	SST25 VF512A	SST25 VF032B	Pm25 LV020	SanDisk MicroSD
Energy Write 1MB	397.866	501.564	481.404	976.794
Energy Read 1MB	146.454	103.656	85.638	517.314

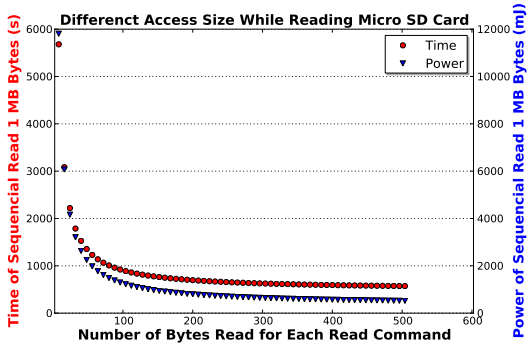


Figure 10. SD card reading using different block lengths.

6.4 Efficiency of Enix Code Update Scheme

The key concept of Enix is the separation of user-defined logical structures and commonly used library functions. By storing the dynamic loading library ELIB on the Micro-SD card, only the user-defined logical structures need to be remotely programmed through the RF. Hence, the number of RF packets for run-time reprogramming is reduced.

We develop five WSN applications compiled with and without Enix. The first three applications are general WSN tasks:

(1) sense and transmit data to the base station through RF,

(2) sense and log data onto the Micro-SD card,

(3) receive and forward packets to other sensor nodes through RF.

(4) EcoNet Transmit, and

(5) EcoNet Receive.

EcoNet is a simple multi-hop network composed of several EcoSpire sensor nodes. The unique ID and the adjacent nodes are all recorded on the Micro-SD card of each sensor node. Every sensor node can invoke the EcoFS API to enumerate its adjacent sensor nodes. The fourth application, EcoNet Transmit, collects the sensor data with a random number and transmits them to the neighboring sensor nodes by enumerating the network data block of EcoFS. The last

Table 10. SD card comparison between manufacturers.

	Speed (KB/s)		Power Consumption (mW)	
	Seq. Read	Seq. Write	Seq. Read	Seq. Write
Toshiba	170.7	51.2	133.9	82.8
Kingston	170.7	37.9	145.5	76.7
SanDisk	170.7	92.8	81.4	86.2
TOPRAM	113.8	51.2	150.1	121
Team	73.1	73.1	105.9	108
Silicon Power	128	53.9	154	118.8
Transcend	93.1	92.8	98.2	117.8

Table 11. Runtime reprogramming code size with/without Enix. Unit: bytes.

application binary size	Sensing & RF Tx	Sensing & Log	RF Tx & RF Rx	EcoNet Tx	EcoNet Rx
no OS	4825	2903	7646	8233	7770
on Enix	474	673	514	1027	442

Table 12. Update code size using VCDIFF delta compression. Units: bytes

Xdelta -9	Sens.& RF Tx	Sens. &Log	RF Tx & Rx	EcoNet Tx	EcoNet Rx
Sensing&RF Tx	X	3148	2834	2663	2810
Sensing&Log	2920	X	1937	1921	1936
RF Tx & Rx	4509	4730	X	3168	3226
EcoNet Tx	4927	5201	3760	X	3847
EcoNet Rx	4523	4753	3246	3291	X

application EcoNet Receive receives the sensed data from neighboring sensor nodes and checks the duplication of random numbers of sequential RF packets, and forwards the valid packets to the base station.

Table 11 compares the uploaded image sizes of the above five WSN applications with and without Enix. The WSN applications without any OS support have binary image sizes larger than 6KB on average. Due to the large code size of the RF library, only the second application has a code size less than 3KB. By comparing the same applications that use Enix as the OS, the sizes of the program images to be transmitted through RF are reduced significantly. These applications produce 500 bytes of binary image on average except for the fourth application, EcoNet Transmit, which generates random numbers without calling any kernel or ELIB functions, and therefore it produces about 1KB of program image.

Most of the run-time reprogramming schemes use the VCDIFF tool to generate the patch between two binary code images, and the patch will be applied by the sensor node. Table 12 shows the results from running VCDIFF for each of the two different WSN applications mentioned above. Although the average binary image size is reduced to 4KB, it is still larger than the applications written on top of Enix.

Table 13 compares the energy cost of 1MB data transfer with an SD card and over RF. As the table shows, reliable RF transmission and reception consumes the highest energy. Thus, for code swapping purposes, it is more energy efficient to swap code from an SD card on demand than to swap over RF. In short, by applying Enix as the operating system, sensor nodes can save energy and time while becoming capable of efficient remote reprogramming due to the reduced binary image size.

7 Conclusions

Enix makes five contributions in the WSN OS area. First, the cooperative threads programming model enhances the performance by decreasing context-switch overhead, making it two times faster than the traditional preemptive multithreaded programming model. This cooperative threading model is easy to learn compared to the event-driven model. Moreover, the local states can be saved and restored automatically during context switches without burdening the programmer with manual state saving in some other programming models. Second, Enix provides code virtual memory to overcome the shortage of on-chip code memory via host-assisted demand segmentation, which most WSN OSs do not support. To achieve virtual memory, the ELIB is built on the host PC composed of PIC segments and is loaded to code memory on-demand by the run-time loader of Enix. Third, remote reprogramming is also available in Enix. Due to the pre-stored ELIB on the Micro-SD card, the size of the binary image of the user program to be wirelessly updated is reduced significantly during the remote reprogramming stage. Fourth, Enix provides a specialized file system called EcoFS that is divided into four configurable parts including code data, preferences, network data, and sensed data, according to the different usage patterns. Besides, a shell for the host PC is also provided to control EcoFS-formatted SD cards, including listing, reading, writing, and modification, thus reducing the difficulty and complexity to access EcoFS formatted data. Finally, the code and data footprints of Enix with full-function including EcoFS are at most 8KB and 512 bytes, respectively, which are the smallest compared to other WSN OSs. Only ten percent of code is machine-dependent, while the rest is written in C language, and thus it is easy to port to other wireless sensor platforms.

Practically speaking, the fact that it runs on a modest-sized 8051 MCU means that Enix is expected to be easily portable to many other integrated RF-MCUs, most of which contain an 8051-compatible core. Thus, Enix is opening up a whole class of cost-effective, miniature, compelling RF-MCUs previously unsupported by WSN OSs. For future work, we are enhancing the shell for interactive execution by adapting a version of EcoExec [18]. We are also further reducing context switch overhead by considering ideas such as Lazy threads. To encourage community involvement in feature enhancements and porting Enix to more platforms, we made Enix open source. It can be freely downloaded from `http://enix.sourceforge.net/`.

Table 13. The energy consumption comparison between SD card and RF (Full Speed). Units: mJ.

	Reliable RF	RF	SD Card
Read/Rx 1MB	1540	1306	517
Write/Tx 1MB	1342	450	977

Acknowledgments

This work was sponsored in part by the National Science Foundation CAREER Grant CNS-0448668, CNS-0721926, the National Science Council (Taiwan) Grant NSC 96-2218-E-007-009, and Ministry of Economy (Taiwan) Grant 96-EC-17-A-04-S1-044. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

8 References

[1] BAI, L. S., YANG, L., AND DICK, R. P. Automated compile-time and run-time techniques to increase usable memory in MMU-less embedded systems. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems* (New York, NY, USA, 2006), ACM, pp. 125–135.

[2] BAI, L. S., YANG, L., AND DICK, R. P. MEMMU: Memory expansion for MMU-less embedded systems. *ACM Trans. Embed. Comput. Syst. 8*, 3 (2009), 1–33.

[3] BHATTI, S., CARLSON, J., DAI, H., DENG, J., ROSE, J., SHETH, A., SHUCKER, B., GRUENWALD, C., TORGERSON, A., AND HAN, R. Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl. 10*, 4 (2005), 563–579.

[4] CAO, Q., ABDELZAHER, T., STANKOVIC, J., AND HE, T. The LiteOS operating system: Towards Unix-Like abstractions for wireless sensor networks. In *IPSN '08* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 233–244.

[5] CHA, H., CHOI, S., JUNG, I., KIM, H., SHIN, H., YOO, J., AND YOON, C. RETOS: resilient, expandable, and threaded operating system for wireless sensor networks. In *IPSN '07* (New York, NY, USA, 2007), ACM, pp. 148–157.

[6] CHEN, C., CHEN, Y., TU, Y., YANG, S., AND CHOU, P. EcoSpire: an application development kit for an Ultra-Compact wireless sensing system. *Embedded Systems Letters, IEEE 1*, 3 (2009), 65–68.

[7] CHOUDHURI, S., AND GIVARGIS, T. Software virtual memory management for MMU-less embedded systems. Tech. rep., Center for Embedded Computer Systems, University of California, Irvine, Nov 2005.

[8] DAI, H., NEUFELD, M., AND HAN, R. ELF: an efficient log-structured flash file system for micro sensor nodes. In *SenSys '04* (New York, NY, USA, 2004), ACM, pp. 176–187.

[9] DUFFY, C., ROEDIG, U., HERBERT, J., AND SREENAN, C. J. Adding preemption to TinyOS. In *EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors* (New York, NY, USA, 2007), ACM, pp. 88–92.

[10] DUNKELS, A., FINNE, N., ERIKSSON, J., AND VOIGT, T. Run-time dynamic linking for reprogramming wireless sensor networks. In *SenSys '06* (New York, NY, USA, 2006), ACM, pp. 15–28.

[11] DUNKELS, A., GRONVALL, B., AND VOIGT, T. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 455–462.

[12] DUNKELS, A., SCHMIDT, O., VOIGT, T., AND ALI, M. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06* (New York, NY, USA, 2006), ACM, pp. 29–42.

[13] EMITT SOLUTIONS. Microcontroller market and technology analysis report – 2008, 2008.

[14] GAY, D., LEVIS, P., VON BEHREN, R., WELSH, M., BREWER, E., AND CULLER, D. The nesC language: A holistic approach to networked embedded systems. *SIGPLAN Not. 38*, 5 (2003), 1–11.

[15] GU, L., AND STANKOVIC, J. A. t-kernel: providing reliable os support to wireless sensor networks. In *SenSys '06* (New York, NY, USA, 2006), ACM, pp. 1–14.

[16] GUSTAFSSON, A. Threads without the pain. *Queue 3*, 9 (2005), 34–41.

[17] HAN, C.-C., KUMAR, R., SHEA, R., KOHLER, E., AND SRIVASTAVA, M. A dynamic operating system for sensor nodes. In *MobiSys '05* (New York, NY, USA, 2005), ACM, pp. 163–176.

[18] HSUEH, C.-H., TU, Y.-H., LI, Y.-C., AND CHOU, P. H. EcoExec: An interactive execution framework for ultra compact wireless sensor nodes. In *SECON 2010* (Boston, MA, USA, June 21-25 2010), pp. 190–198.

[19] INTEL. Portable Formats Specification, Version 1.1.

[20] KLUES, K., LIANG, C.-J. M., PAEK, J., MUSĂLOIU-E, R., LEVIS, P., TERZIS, A., AND GOVINDAN, R. TOSThreads: thread-safe and non-invasive preemption in TinyOS. In *SenSys '09* (New York, NY, USA, 2009), ACM, pp. 127–140.

[21] KNUTH, D. *Fundamental Algorithms, Third Edition.* Addison-Wesley, 1997, ch. Section 1.4.2: Coroutines, pp. 193–200.

[22] KOSHY, J., AND PANDEY, R. VMSTAR: synthesizing scalable runtime environments for sensor networks. In *SenSys '05* (New York, NY, USA, 2005), ACM, pp. 243–254.

[23] LABROSSE, J. J. *MicroC/OS-II, The Real-Time Kernel 2ND EDITION*. CMP Books, 2002.

[24] LEVIS, P., AND CULLER, D. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2002), ACM, pp. 85–95.

[25] LEVIS, P., MADDEN, S., POLASTRE, J., SZEWCZYK, R., WHITEHOUSE, K., WOO, A., GAY, D., HILL, J., WELSH, M., BREWER, E., AND CULLER, D. TinyOS: An operating system for sensor networks. *Ambient Intelligence* (2005), 115–148.

[26] MADDEN, S. R., FRANKLIN, M. J., HELLERSTEIN, J. M., AND HONG, W. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst. 30*, 1 (2005), 122–173.

[27] MATHUR, G., DESNOYERS, P., GANESAN, D., AND SHENOY, P. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In *SenSys '06* (New York, NY, USA, 2006), ACM, pp. 195–208.

[28] MÜLLER, R., ALONSO, G., AND KOSSMANN, D. A virtual machine for sensor networks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), ACM, pp. 145–158.

[29] NATH, S., AND KANSAL, A. FlashDB: dynamic self-tuning database for NAND flash. In *IPSN '07* (New York, NY, USA, 2007), ACM, pp. 410–419.

[30] PARK, C., LIM, J., KWON, K., LEE, J., AND MIN, S. L. Compiler-assisted demand paging for embedded systems with flash memory. In *EMSOFT '04* (New York, NY, USA, 2004), ACM, pp. 114–124.

[31] REAL TIME ENGINEERS, LTD. FreeRTOS: Free, portable, open source, royalty free, mini real time kernel. http://www.freertos.org/, 2010.

[32] TSIFTES, N., DUNKELS, A., HE, Z., AND VOIGT, T. Enabling large-scale storage in sensor networks with the coffee file system. In *IPSN'09* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 349–360.

[33] ZEINALIPOUR-YAZTI, D., LIN, S., KALOGERAKI, V., GUNOPULOS, D., AND NAJJAR, W. A. Microhash: an efficient index structure for fash-based sensor devices. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2005), USENIX Association, pp. 3–3.