2010 International Conference on Distributed Computing Systems

Versatile Stack Management for Multitasking Sensor Networks

Rui Chu [1], Lin Gu [2], Yunhao Liu [2,3], Mo Li [2], Xicheng Lu [1]

[1] National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha, China
[2] Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, China
[3] Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing, China
[1] {rchu, xclu}@nudt.edu.cn [2]{lingu, liu, limo}@cse.ust.hk

***Abstract*—The networked application environment has motivated the development of multitasking operating systems for sensor networks and other low-power electronic devices, but their multitasking capability is severely limited because traditional stack management techniques perform poorly on small-memory systems. In this paper, we show that combining binary translation and a new kernel runtime can lead to efficient OS designs on resource-constrained platforms. We introduce SenSmart, a multitasking OS for sensor networks, and present new OS design techniques for supporting preemptive multi-task scheduling, memory isolation, and versatile stack management. We have implemented SenSmart on MICA2/MICAz motes. Evaluation shows that SenSmart performs efficient binary translation and demonstrates a significantly better capability in managing concurrent tasks than other sensornet operating systems.**

I. Introduction

The growing popularity of low-power and pervasive wireless computing devices naturally leads to an emphasis on networked operations and a seamless interaction with the ambient context. This trend is seen on PDAs, active RFIDs, various intelligent consumer electronic devices, and wireless sensor networks. Such networked operations and contextual interaction make the application software much more complex than that running on traditional embedded devices. Particularly, the sensor network is a representative technology where the relevant design factors – resource constraints, application complexity, and multi-hop networking – are manifested to a great extent. A typical sensor node may only have a simple CPU and a few kilobytes of RAM [1][2], but the software running on it can take tens of thousands lines of code to implement, performing a wide range of tasks related to sensing, topology control, wireless routing, power management, signal processing, and system adminstration [3][4][5].

The complexity of application software and the fact that the software runs on numerous unreliable devices call for strong system software support. One critical need is a preemptive multitasking operating system. Without that, handling important interrupts could be delayed by long computational tasks, communication operations could disrupt the timing of the sensor channel sampling, and unpredictable latencies would make network level activity unreliable and energy-costly.

Consequently, a number of recent operating systems for sensor networks have included multitasking and preemptive scheduling features. However, a careful examination of current multitasking systems shows severe limitations in both functionality and usability. The key problem is stack management – how can an operating system efficiently manage multiple stacks on a small-memory platform?

In a multitasking system, the stacks of concurrent tasks routinely grow and shrink during their execution. The dynamics of the stacks is particularly high for event-driven systems, which is the *de facto* standard programming model for sensornet systems [2][6][7]. The ability to hold multiple stacks in memory and efficiently handle the stack dynamics is a fundamental determinant of the performance of a multitasking OS.

On resource-rich platforms, stack management is often considered as a solved problem with textbook solutions. Three facts have helped the traditional stack management become successful on such systems.

- Virtual memory in modern microprocessors eliminates external fragmentation, and limits stack collisions to only occur within individual address spaces.
- Abundant virtual memory space is usually provided for typical stack usage. Hence, inter-thread stack collisions inside a process can be avoided by allocating "sufficient" memory areas to the threads.
- The size of the physical memory in resource-rich systems keeps growing, making internal fragmentation negligible.

However, none of these facts are true in low-power computing systems, and, not surprisingly, traditional solutions do not perform well on senor nodes that are strictly resource constrained and absent of virtual memory hardware. Some recent multitasking systems have ported existing stack management solutions to sensor networks, but suffered severe limitations. They typically require that programmers provide worst-case estimation of stack usage for various tasks, or use a statically analyzed value for stack size, resulting in significant waste in memory allocation and degradation in the number, types, and combinations of tasks the OS can schedule. Hence, current multitasking sensornet operating systems typically have a weak "stack versatility" – a term

1063-6927/10 $26.00 © 2010 IEEE
DOI 10.1109/ICDCS.2010.42

IEEE computer society

we use in this work to describe the ability to efficiently handle multiple stacks without *a priori* knowledge on their dynamics. The weak stack versatility directly affects these OSes' ability to accommodate concurrent tasks.

Designing versatile stack management on resource constrained platforms is a new challenge. Though the low-power computing technology develops steadily, virtual memory is still very unlikely to be available to the sensor nodes using very-low-power processors. Some recent embedded processors claim to enable a 32-bit architecture with the cost and power consumption of 8-bit systems. The claim is, however, only partially true because downscaling power is often accompanied by removing architectural features. Most low-power microcontrollers (MCUs) do not support hardware memory translation or memory protection. Many low-power systems also do not support instruction privilege, which is prerequisite for traditional multitasking designs. It is also unlikely that very-low-power systems can afford to scale up physical memory size as quickly as the cost of RAM drops. In the past two decades, the typical memory capacity of computer systems has grown dramatically, but many MCUs today still use kilobytes of SRAM for energy efficiency.

Furthermore, simple augmentations to stack handling or memory system are unlikely to work in our design context. A simple copy-on-switch scheme appears to solve the problem by swapping one task's stack out to the external storage (FLASH on motes) and swapping it in when the task is activated again. However, writing the external FLASH takes more than 10 milliseconds on a MICA2 mote. Such long context-switch delays, as well as other limitations (e.g., the erase cycle of FLASH chips), make the copy-on-switch scheme impractical for sensor nodes. Static stack analysis, on the other hand, has intrinsic limitation due to the incomputability of the general problem – how many blocks on the tape a Turing Machine reads or writes [8]. The use of fibers simplifies the scheduler design but does not eliminate the need for stack management [7]. Some other solutions, such as the "protothreads" [9], have their own limitations, as we will cover in Section II.

In contrast to earlier solutions, we take an approach of combining binary translation and a lightweight kernel runtime to provide strong stack versatility and multitasking capability. We have designed and implemented a new operating system prototype, SenSmart, which includes several new designs on base-station-side binary re-writing, logical address translation, and stack relocation. These new designs reduce memory overhead, minimize the external fragmentation, and provide new level of stack versatility on strictly resource constrained sensor nodes. As an example of the effectiveness, SenSmart can handle a multi-task workload even when the total "needed" stack space of all tasks exceeds the total available stack space in the physical memory.

The rest of this paper is organized as follows. Section II discusses the related work. Section III presents an overview of our system architecture and technical approaches. Section IV describes the detailed design of SenSmart. Section V focuses on the system implementation and evaluation. We summarize the work in Section VI.

II. Related Work

Researchers have developed a number of operating systems for sensor networks and low-power devices, such as TinyOS [2],SOS [10], Contiki [11], MANTIS OS [12], Nano-RK [13], LiteOS [14], and the t-kernel [15]. In order to support more reliable, efficient, and sophisticated application systems, recent systems start to provide advanced OS features, such as multitasking, memory protection, and software-based memory swapping. In particular, multitasking has become an important feature as the applications grow to be more network-oriented and function-rich. However, for reasons explained in Section I, existing multitasking systems for sensor networks have to place harsh restrictions on the concurrent tasks, such as pre-determined maximum stack depth.

In TinyOS [2], tasks are executed (called through a function pointer) in serial. Hence, there is no concurrency among them, and the stack management is simple. Without memory isolation, a task in TinyOS, as well as other application logic, can write to any physical memory areas including those used by the OS. To improve the quality of system services, other works attempt to add features such as preemption or memory checking for TinyOS [16][17][8], but the proposed methods still have their own limitations.

MANTIS OS [12], Nano-RK [13], Contiki [11], SOS [10], Harbor [18], and RETOS [19] attempt to design multitasking sensornet operating systems using traditional OS design techniques. For instance, MANTIS OS implements a multithreading kernel with preemption. Each thread has its own stack area, and scheduling is built on clock interrupts. As explained in Section I, traditional solutions usually lead to low stack versatility and harsh restrictions on application tasks. For example, it is very difficult to efficiently allocate stack memory to tasks without introducing extra burden (and dependence) on application programmers. For correctness and simplicity, such systems usually allocate stack memory to a thread based on the worst-case situation. Without virtual memory paging, this pessimism, combined with the aforementioned inflexible allocation, aggravates the waste and drains a fair portion of previous memory resources.

Besides stack versatility, most of the aforementioned systems also have difficulty with preemptive scheduling because the clock-interrupt-based scheduling on sensor nodes is not reliable. Many MCUs used in sensor nodes do not support instruction privilege, and application code can disable interrupts.

Departing from traditional solutions, the t-kernel [15] implements preemptive scheduling, OS protection and vir-

tual memory with binary re-writing on sensor nodes. The tasks in the t-kernel share a common stack space, and the memory protection is asymmetric – only the kernel memory is protected. SenSmart also uses binary re-writing as an important technique to implement preemptive scheduling and memory isolation. Different from the t-kernel and other binary re-writing systems for sensor networks [20], SenSmart conducts complete binary translation on the base station. As we will show later, this approach gives SenSmart unique advantages in reducing system complexity and code inflation ratio.

Maté [21] and MagnetOS [22] represent the virtual machine approach, another software-based method to provide enhanced system abstractions. The disadvantage of this approach is that scarce resources do not allow virtual machines to perform sophisticated optimization on the bytecode. Hence, such virtual machines often resort to slow interpretation based execution.

III. Overview

In this section, we give an overview of SenSmart. We first define the design space, then provide a high-level operational view of SenSmart.

A. Examination of the design space

We use the MICA2/MICAz sensor nodes [23] as representatives of strictly resource constrained networked platforms. As the assumption on hardware, we expect that the hardware platform has at least the same amount of resources as a MICA2 mote, which has an 8-bit ATmega128L MCU, 4KB SRAM-based data memory, and 128KB FLASH-based program memory.

As assumptions on software, we expect the applications to meet the following requirements.

- The application code does not modify itself. Note that this restriction does not apply to the OS code – reprogramming can be performed as an OS service.
- The heap areas and stack areas used by the applications are not overlapping, i.e., the applications do not intentionally use a memory area as both a heap and a stack.
- The application code does not use dynamic memory allocation.

Most of the sensornet platforms and TinyOS/nesC applications meet these requirements. As an explanatory note to the third assumption on software, most of the applications running on MICA2/MICAz motes do not use dynamic memory allocation. For those applications that do, it is not difficult to add a specific allocation module, which claims a chunk of memory and re-allocates parts of it upon requests, to emulate the dynamic memory function. Some versions of TinyOS already contain such a module.

In this strictly resource-constrained design space, we design SenSmart to be a reliable OS with solid multitasking capability. Using software methods, it solves the critical problems of stack versatility and purely software-based preemptive scheduling.

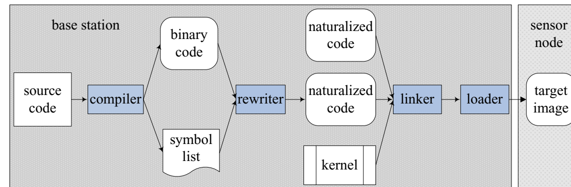

Figure 1. The compiling, re-writing and loading process in SenSmart

B. System overview

A sensornet application is often written in a sensornet programming language, such as nesC [6]. After being compiled into a binary executable, the application program is loaded into the program memory and executes on the sensor node. SenSmart rewrites the binary executable on the base station after the application program has been compiled and before it is loaded. The re-writing logic, called rewriter, analyzes the binary image, and patches the application code to ensure that multiple application tasks run on one node following appropriate multitasking semantics.

Figure 1 shows the process of sensornet application development with SenSmart. After the compiler generates the binary code and the memory usage information contained in the symbol list, the rewriter translates the program code to be a “naturalized program”, which cooperates with the kernel runtime to support multitasking. After compiling and translating multiple programs, SenSmart links them together with the pre-compiled system kernel, which includes the kernel runtime, to form the executable to be loaded to sensor nodes. When the application programs are instantiated, they execute concurrently as application tasks under the control of the kernel. Each application task has its respective time slice and memory region. SenSmart schedules the tasks with preemption, and isolates their memory regions by translating memory addresses into appropriate physical addresses at runtime. Transparent to application tasks, SenSmart automatically adjusts the sizes and locations of the tasks’ stack areas when it is necessary, and avoids stack collisions when it is possible.

IV. SenSmart Design

We present the design details of SenSmart in this section. We first introduce how SenSmart performs binary re-writing on the base station, then briefly cover multi-task scheduling, and present the details of the memory management.

A. Binary re-writing on the base station

Following common C and nesC programming paradigms, sensornet programs are usually developed and executed with a view that they exclusively use the CPU and memory on the sensor node. The code re-writing process, performed on the

base station by the rewriter, virtualizes the CPU and memory so that multiple programs thus developed can be instantiated as application tasks on one sensor node and share the CPU and memory resources.

The rewriter modifies the following types of instructions.

- The instructions which affect the CPU control flow, including the branch instructions and the CPU control instructions (e.g., the SLEEP instruction), are re-written in a way to ensure that the OS frequently takes over CPU to run system services.
- The direct or indirect memory access instructions and stack pointer operations, are re-written in a way that cooperates with the memory management mechanism.
- The instructions that access some OS-reserved resources are also re-written. For example, SenSmart reserves the Timer3 of ATmega128L MCU as a global clock, therefore, the accesses to the I/O registers of Timer3 are intercepted and handled in special ways [1].

Departing from the on-node binary re-writing in the t-kernel, SenSmart rewrites the binary code on the base station, and strikes a balance between reliability and cost. This approach has a number of benefits as follows.

First, the base station can collect the whole-program characteristics such as the heap usage information from the symbol list generated in compiling. Such information is useful for our approach.

Second, having plenty of resources, the base station is able to thoroughly analyze the application program for more efficient re-writing. In contrast, the t-kernel performs code re-writing on resource constrained sensor nodes, and can only work on no more than a "page" at a time. One page containing up to 128 instructions. Such a modest size of re-writing units limits opportunities of optimization, and introduces additional complexity.

Finally, by moving the code re-writing logic to base stations, SenSmart also significantly reduces the kernel size on individual sensor nodes.

Another design in SenSmart is that it maintains an approximate linearity of the original program's instruction addresses and the naturalized program's. After the re-writing, one instruction can be translated into a variable number of instructions, and this usually results in a code inflation non-linear to the instruction addresses in many other systems. SenSmart regularizes the instruction re-writing to mitigate the inflation. When patching one instruction, SenSmart replaces the instruction with one JMP or CALL instruction, which takes the control flow into a code snippet (called "trampoline") corresponding to the re-written logic. All of the trampolines are appended after the application program. Hence, the instruction count of the patched program, excluding the trampoline code, is exactly the same as that of the original, though the byte sizes may still differ because the byte sizes of individual instructions vary. Such an approximate linearity makes it easier to map instruction addresses from the original program to the patched one, particularly when the addresses have to be resolved at run time (e.g., indirect branches). Moreover, since many trampolines are similar, they can be merged to save space (even if they belong to different application programs), and further reduce the size of the naturalized program.

[1] Note that Timer3 is not used for preemption, as SenSmart chooses not to rely on clock interrupts for scheduling.

B. Task scheduling

With no privilege support on many sensor nodes, it is unreliable to design preemptive scheduling based on clock interrupts as traditional operating systems do, since the interrupts could be disabled by application tasks. Instead, SenSmart modifies the branch instructions so that one out of 256 backward branches executed by the MCU jumps to the OS kernel, a technique also used in the t-kernel. However, SenSmart differs from the t-kernel in that it uses time slices to schedule tasks. Counting time slices using Timer3, SenSmart schedules tasks using a round-robin policy, and preempts a task *after* its time slice is used up. Note that the scheduling does not guarantee that the preemption occurs exactly *when* the time slice ends because the software traps are triggered aperiodically. However, the delay of the preemption, usually no more than a couple of microseconds, is small enough to be ignored for most applications. Even with interrupts disabled, SenSmart can still preempt the application task by the software traps, although the moment such preemption occurs may be slightly later than the moment stipulated by the time slice.

C. Stack management

An effective multitasking mechanism shall accommodate arbitrary combinations of tasks within the resource limitation, and handle the dynamics of resource utilization when the tasks execute concurrently. Earlier sensornet operating systems are not able to meet this requirement, and a major obstacle is the difficulty in handling the stack dynamics.

With preemptive multitasking, switching to a different task before the current one terminates forces the system to maintain multiple "active" stacks in the sense that the stacks are still used by tasks, and that the stacks may expand and shrink without predictable patterns. In a small-memory system with multiple active stacks, the expansion of one task's stack can easily touch the border of another active stack, and some application tasks may have to be terminated even if the system still has free memory space. Hence, the ability to manage multiple stack areas is a crucial part of the memory management in a multitasking OS.

To provide stack versatility and adapt to the variety and dynamics of stack usage, SenSmart introduces software-based logical addressing, and automatically adjusts various

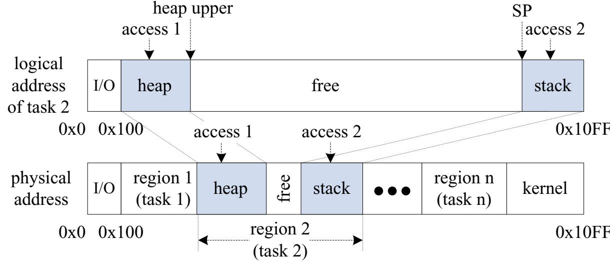

Figure 2. Layout of the data memory and logical addressing

tasks' stacks while guaranteeing the memory access semantics. In this section, we first give an overview of the memory organization in SenSmart, then present the logical address translation and stack management algorithms.

1) Memory Organization: SenSmart divides the data memory space into the I/O area, the application area and the kernel area. The I/O area is mapped to the I/O registers in ATmega MCU, and the kernel area is reserved by SenSmart. The application area is divided into independent memory regions, each region assigned to one task. Without dynamically allocated memory (refer to Section III-A), a memory region comprises a fixed-size heap area, and a variable-size stack area. SenSmart laces the heap area in the lower part of a memory region, and the stack area in the upper part. Figure 2 shows the structure of memory organization.

A task running in SenSmart is analogous to a process instead of a thread because each task has its independent memory region with a heap and a stack. For each task, we use three pointers, p_l, p_u, and p_h, to indicate the lower bound of the task's memory region, the upper bound of the task's memory region, and the upper bound of the task's heap area, respectively. Suppose the size of the physical memory is M. Obviously, we have $p_l < p_h < p_u < M$. After excluding the I/O area, the kernel area and all heap areas, the remaining space is the total available stack space for all tasks.

2) Logical addressing: SenSmart uses a logical addressing mechanism to provide each application task a logical memory space, which is as large as the physical memory, so that the task can "exclusively" use it. The logical addresses are translated into physical addresses at run time, and accesses beyond a task's memory region are intercepted and treated as invalid instructions. Such logical addressing mechanism makes the program-visible memory addresses independent of their locations in the physical memory. It not only makes it very easy to implement memory isolation for multiple tasks. but also allows SenSmart to tune the locations of the memory regions and the stack sizes for various application tasks with different stack dynamics.

To implement logical addressing on strictly resource constrained hardware, the binary rewriter modifies memory access instructions to include logic for run-time address translation. Under the assumptions listed in Section III-A, there are only three types of valid data memory accesses: 1) random access in the current heap area, such as LD/ST instructions in ATmel's AVR instruction set; 2) random access in the stack frame of the current function; 3) LIFO access to the current stack using stack-mutating instructions, such as PUSH/POP/CALL/RET. The memory translation handles all three types of accesses, as illustrated in Figure 2.

Generally, the address translation adds a displacement (p_l for heap and for $p_u - M$ for stack) to the original memory address to form the effective memory address, as well as performs boundary checking. Meanwhile, various forms of translation and adjustments are added for both correctness and performance.

When a task attempts to retrieve its stack pointer, the kernel translates the stack pointer to the logical address which uses the upper bound of the logical memory space as stack bottom. The kernel will also translate it back when an application tries to set the stack pointer. This allows stack memory accesses implicitly using the stack pointers to execute efficiently and correctly.

The overhead of data memory address translation is relatively high because there is no dedicated register or other hardware support, and several extra memory accesses are needed to determine the boundary of current memory region. We have noticed that, in most sensornet applications, 2 or 4 memory access instructions are often performed together using the same indirect address registers to fetch or store word or double-word data. Thus the binary rewriter can identify the instructions as a grouped memory access and only translate the address once. This optimization effectively improves the performance, and is made possible because basic block information can be used by the rewriter to ensure correctness.

Despite the overhead, the benefits of the memory indirection are multi-fold. The most important one is that application tasks can program on logical address spaces which are independent of the real locations in the physical memory. The logical addressing is a key functionality of SenSmart that enables the stack relocation to be discussed in Section IV-C3. In sensor networks, data memory is always a keenly constrained resource. CPU cycles are, however, usually not a bottleneck. Hence, we believe the benefit of logical addressing in system functionality, reliability, and usability by far out-weights the overhead in CPU cycles.

Similar to data memory, program memory address translation is also necessary for the correctness of the naturalized program. At run time, the translation of program memory address occurs when there is a program memory data access or an indirect branch. Other program memory address translations, for example, branches with relative addresses, are directly resolved by the binary rewriter on the base station. Such a separation of program memory address translation effectively reduces the workload on sensor nodes.

Because each instruction in the AVR instruction set is 16 or 32 bits long, we use a sorted array called "shift table"

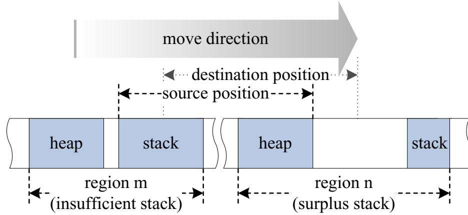

Figure 3. Stack reallocating when an application has insufficient stack space

to record the addresses of re-written instructions that are inflated from a 16-bit instruction to a 32-bits JMP/CALL instruction. Based on the shift table, we can map a program address in the original application task to the corresponding program address in the naturalized program. Note that, on an ISA with fixed-size instructions, the shift table can be eliminated and the time complexity of program memory translation can be reduced to virtually zero.

3) Stack relocation: Most sensornet application programs follow an event-driven model [2], which typically use the stack in a highly dynamic manner [7]. In a sophisticated TinyOS application, 10 levels of nested function calls are common even with compiler inlining optimization, resulting in a sizable stack area. Following a "split-transaction" pattern for event-driven processing, tasks can quickly shrink their stack on a blocking I/O, and leave the unfinished work to another event-driven task to be executed when the I/O completes. There are also tasks which use only a very small stack. Such a workload pattern makes fixed-size stack management cumbersome and inefficient.

One important technique SenSmart uses to enhance stack versatility is allowing stacks to freely relocate in the physical memory, and programs will run on SenSmart without knowing the underlying stack motions. This appears to be a heavyweight solution, but the cost is, in fact, surprisingly low in a small-memory system, and it keeps the stack memory semantics as the compilers know it.

In SenSmart, All of the tasks are given a predefined initial stack size. During their execution, some tasks may need more stack space, and others still have surplus. In order to check the stack space at run time, the binary rewriter rewrites the instructions that change the stack pointer to invoke a stack checking routine. When SenSmart detects that the stack space of a task is to overflow, it increases the stack of the overflowing task by relocating a number of tasks' stacks. With stack relocations, SenSmart adapts to the stack memory usage of the combination of tasks concurrently running in the system.

The relocation logic enumerates the application tasks in the system to look for available memory. The application with most surplus available stack space is selected, and the memory regions are moved as shown in Figure 3. The application task with the most surplus stack space provides half of its available stack space to the one with insufficient stack space. Since the application programs only use logical memory addresses, all accesses to application tasks' memory regions, including heap and stack areas, can be translated to correct physical addresses after the stack relocation.

V. Performance evaluation

We have conducted extensive evaluation on SenSmart prototype implementation. First, the overhead of key operations is measured. Second, we use kernel benchmark programs to evaluate typical code in sensor networks. Finally, we show that SenSmart has a much better capability in accommodating concurrent tasks.

A. Implementation

We have implemented SenSmart on MICA2/MICAz motes. The SenSmart kernel is configurable. In the default setting, the kernel occupies less than 6% of the program memory and about 10% of the data memory. This memory footprint is much smaller than the t-kernel and many other operating systems. The t-kernel uses more data memory because it performs on-node re-writing, and reserves physical data memory to be used as virtual memory frames.

Table I lists the implemented features of typical related systems as a comparison. Although these systems have respective advantages, SenSmart performs better in multitasking related functionalities as listed.

B. Overhead

The task scheduling and memory protection logic in SenSmart ensure the system integrity under multitasking, but they also inevitably introduce overhead into the system. Using the ATmega simulator in AVR Studio, we measure the overhead in CPU cycles, and list the results in Table II.

If not fully optimized, the overhead of memory address translation and checking would dramatically affect system performance since the memory accesses occur frequently in programs. Fortunately, this overhead can often be reduced within basic blocks as discussed in Section IV-C2. Indirect branches have high overhead due to branch destination lookup at run time, but such instructions are rare in current sensornet applications. The overhead of stack reallocation and context switching varies in different cases. The numbers shown in Table II give representative examples. It is worth noting that relocating a stack on an ATmega128L MCU running at 7.32MHz may introduce $100 - 300\mu s$ delay. SenSmart is conservative on memory relocations, hence such delays should be infrequent in stable systems. Moreover, since many common operations, such as sensor I/O and packet transmissions, take multiple milliseconds on a sensor node, we feel confidently that occasional sub-millisecond delays paid for an unprecedentedly versatile multitasking support is a small and welcomed cost.

Table I
COMPARISON OF TYPICAL SYSTEMS

	TinyOS/TinyThread	Maté	MANTIS OS	t-kernel	RETOS	LiteOS	SenSmart
TinyOS Compatible	N/A	No	No	Yes	No	No	Yes
Preemptive Multitasking	Yes	No	Yes	Partial	Yes	Yes	Yes
Concurrent Applications	No	N/A	No	No	No	No	Yes
Interrupt-free Preemption	Yes	N/A	No	Yes	No	No	Yes
Memory Protection	No	Yes	No	Partial	Yes	No	Yes
Logical Memory Address	No	N/A	No	No	No	No	Yes
Physical Mem Management	Automatic	Automatic	Automatic	Automatic	Automatic	Manual	Automatic
Stack Relocation	No	No	No	No	No	No	Yes

Table II
OVERHEAD OF KEY OPERATIONS

Operation			Overhead (cycles)
System initialization			5738
Memory address translation	Direct	I/O area	2
		Others	28
	Indirect	I/O area	54
		Stack frame	69
		Heap	62
		Program memory	376
Stack operation	Get stack pointer		45
	Set stack pointer		94
	Stack relocation		2326
Context switching	Context saving		932
	Context restoring		976
	Full switching		2298

C. Kernel benchmark programs

To assess SenSmart with typical sensornet applications, we test the seven kernel benchmark programs used in the t-kernel for our evaluation [15]. These programs cover typical operations in sensornet applications. Figure 4 analyzes the code inflation of the kernel benchmark programs under SenSmart and the t-kernel, as compared with the native size of the code. The code inflation under SenSmart is within 200%. As a comparison, the t-kernel, which also using the binary translation, makes the code size much larger than SenSmart. The t-kernel's philosophy is to performs swapping for both code and data, and keeps only the working set in memory so that the effect of code inflation is mitigated. SenSmart, in contrast, conducts translation on base station, and can make translated code much more optimized in terms of space efficiency.

In fact, the increased use of program memory does not create a new bottleneck in the system. In general, the program memory is not as scarce a resource as data memory even for large sensornet applications. A very large sensornet application, the VigilNet system, has more than 30,000 lines of C code for the mote-side software. It occupies 90% of data memory without including stack space, but only 30% program memory including both TinyOS and the application code [4].

After measuring the code size of the programs, we compare the execution performance of SenSmart with other software-based solutions. It is not a design goal of us to optimize for execution speed, but SenSmart still has a reasonable execution speed, and only shows a moderate slowdown as compared to the t-kernel, which is optimized for execution speed. Although t-kernel has better performance in most of the seven programs as Figure 5 shows, we believe that the extra cost is fair and reasonable because SenSmart supports concurrent tasks with independent time slice and memory regions, while the task and memory protection in the t-kernel are both simpler as shown in Table I.

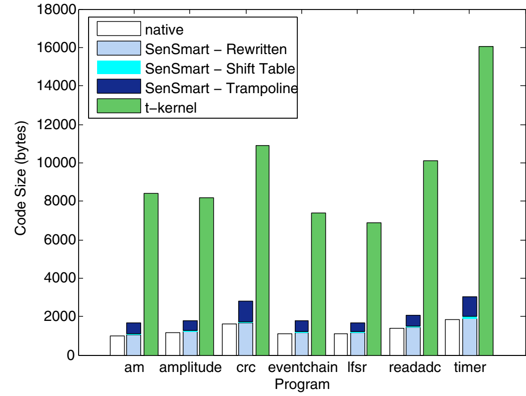

Figure 4. Code inflation of kernel benchmark programs

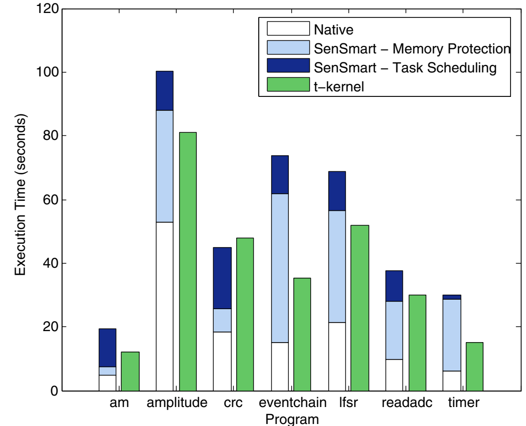

Figure 5. Execution time of kernel benchmark programs

We use a *PeriodicTask* program to emulate the common operating pattern of sensornet applications – periodic events triggering computational tasks. Using the *PeriodicTask* program, we examine SenSmart's performance in more realistic settings, and stress-test it to see when it fails to handle the workload. The computational tasks in *PeriodicTask* can be configured to a desirable computation size (number of instructions) to emulate applications of different complexity. When we configure less computational instructions for each task, it works more like an ordinary event-driven application. When more instructions are added into the tasks, it becomes more and more computation-intensive until the workload is completely CPU bound.

We test the *PeriodicTask* program in SenSmart with different computation sizes. For each test, we record the execution time of 300 tasks on real sensor nodes. Moreover, We use the Avrora [24] to measure the proportion of the active cycles, which can be taken as the average CPU utilization during the execution. As a comparison, the cases for the native-code execution without any operating system overhead are also tested, and we list the results in the work of t-kernel. As shown in Figure 6(a), when the computation size is less than 60,000 instructions, the execution time in SenSmart is very close to the native case. After the threshold of 60,000 instructions, the execution time increases dramatically. Figure 6(b) shows the CPU utilization data. Larger computation size inevitably increases the CPU utilization, and it increases more rapidly in SenSmart due to the overhead of task switching and memory protection operations. When it reaches 60,000 instructions, the CPU utilization in SenSmart is nearly saturated. Beyond that saturation point, the task execution takes longer time because, when the CPU is busy, some timer tasks cannot be handled in time. Hence, SenSmart is suitable for the applications with a CPU utilization lower than 30%, which is the common case in sensornet applications.

It is noteworthy from Figure 6(a) that, for the tasks with less than 60,000 instructions, SenSmart performs better than t-kernel even though the latter has lighter memory protection operations. The reason is that the t-kernel has a warm-up re-writing overhead, which introduces an initialization delay of about one second. This implies that SenSmart is likely to have better performance in both of execution speed and multitasking capability for applications that are not computation intensive.

We have also compared the t-kernel and SenSmart with the software-based virtual machine, Maté, using an equivalent *PeriodicTask* program. The result is shown in Figure 6(c), in which the Y-axis is exponential. The execution time of *PeriodicTask* program in Maté is much slower than in t-kernel and SenSmart. As a fully virtualized environment, the virtual machine can also enhance reliability and ensure memory protection [25]. But interpretation-based execution has a performance penalty, as indicated by the significant difference in execution speed.

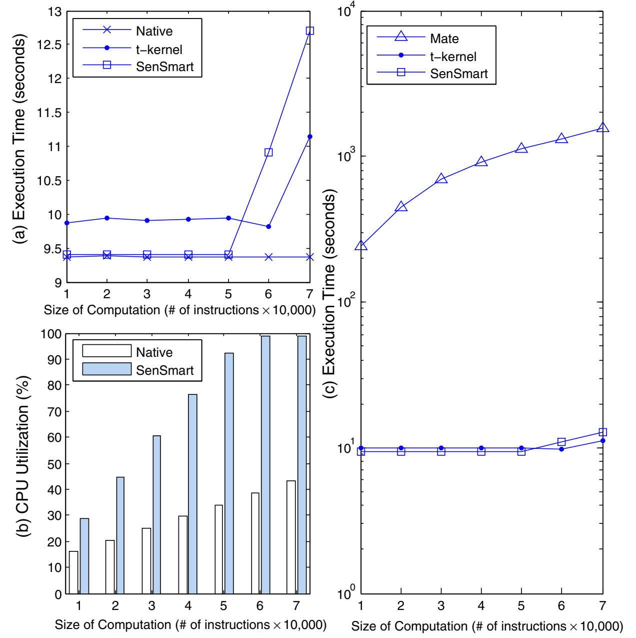

Figure 6. Execution time and CPU utilization of *PeriodicTask* program

D. Stack versatility

In this section, we evaluate SenSmart's versatile stack management and compare it to other multitasking operating systems in this aspect. To make the comparison possible, we choose to use the number of maximal schedulable tasks as the metric, because it is an important indication of the operating system's multitasking capability and can be evaluated on most of the multitasking operating systems.

We use a set of tasks with different stack dynamics. A common workflow in a sensornet application follows a "sense-and-send" paradigm in an event-driven style. Sensor and radio channels feed data to the sensor node. The arrival of data triggers various event-driven handlers to read the data, verified them, and usually, stored them in the heap in a specific data structure. When a certain amount of data is accumulated, a few larger processing tasks may be activated to read data from the heap, analyze them, and, sometimes, send out wireless packets. There are usually multiple processing tasks in a system, e.g., compression, routing, and signal processing, and these tasks are activated upon different conditions. We use one data feeding task and several processing tasks to approximate such a "sense-and-send" workflow. The data feeding task periodically stores randomly generated "incoming data" onto the heap to form six binary trees, and then the processing tasks are activated to recursively search randomly selected binary trees. Both the shapes and heights of the binary trees depend on the sequence of the random data, and, hence, the search tasks have a slight variance in their recursion depths – 12 levels on average and some reaching 15 levels. Each level of recursion

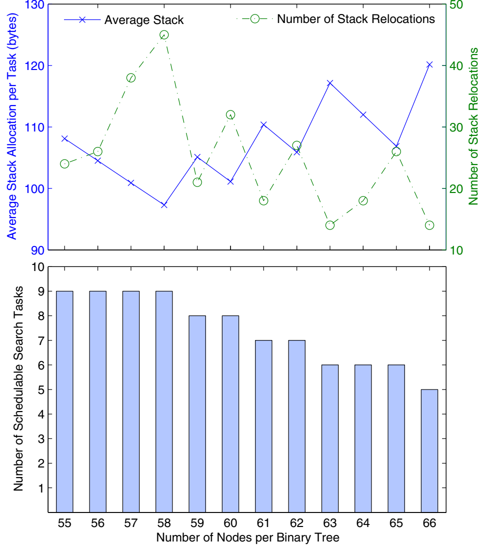

Figure 7. Binary tree search in SenSmart with increasing tree sizes

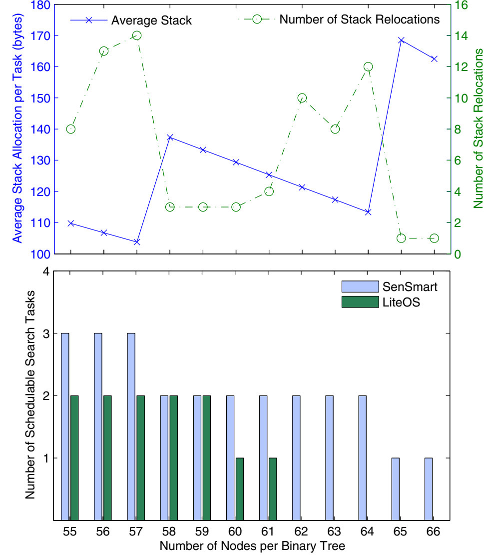

Figure 8. Comparison of SenSmart and LiteOS

adds 15 bytes to the stack, and, hence, the historically largest stack size of the search tasks is around 180 bytes.

Figure 7 shows the number of stack relocation activities, the average stack allocations, and the maximal number of search tasks the system may accommodate, with different binary tree sizes. Obviously, the larger binary trees will increase the heap usage, thus the stack space has to be reduced. Furthermore, the larger binary trees may also increase the recursion depth of the search task, thus the stack usage of each task also increases. Both factors reduce the maximal number of search tasks that can concurrently run in SenSmart. SenSmart terminates one task when it can no longer accommodate all concurrent tasks in the system. As shown in Figure 7, when a search task is terminated, the average stack space for each search task increases because the remaining tasks start to expand their stack space on the released memory of the terminated task. SenSmart performs more stack relocations to tune the sizes of the stacks of concurrent tasks in the system, when the average stack allocation is overly insufficient. Nevertheless, the maximal number of stack relocations is under 50 times, thus the performance penalty of stack relocation is insignificant.

As mentioned before, a search task needs about 180 bytes of stack size on average, while we can also observe from Figure 7 that the average stack allocations in all cases do not exceed 130 bytes. It implies that, on average, a task does not have sufficient stack space for its need, while SenSmart can still accommodate all the tasks by exploiting the dynamics of the stacks. Only when the average stack allocation is significantly smaller than the average required stack size is SenSmart forced to terminate tasks. Such a behavior shows an excellent stack versatility and proves the versatile stack management's effectiveness.

Figure 8 compares SenSmart with LiteOS in their ability to schedule tasks under memory constraints. LiteOS is a well-designed and easy-to-use sensornet operating system with a number of useful features. The advanced design, meanwhile, requires that LiteOS uses more than 2000 bytes of static data in the physical data memory. To perform a fair comparison, we limit the number of binary tree to two, and instruct SenSmart to use the same amount of memory for overall stack space as what LiteOS uses. The result in Figure 8 shows that the versatile stack management enables SenSmart to adapt to stack dynamics and accommodate more concurrent tasks.

VI. Conclusions

Multitasking is a useful system function for complex sensornet applications. However, it is not easy to implement a reliable and flexible multitasking system using traditional approaches on resource constrained sensor nodes. SenSmart implements a multitasking operating system by solving the critical stack management problem with a set of techniques to enhance stack versatility, which significantly improves the multi-task scheduling capability. We have implemented SenSmart. The OS exhibits excellent capability in scheduling concurrent tasks.

Acknowledgements

The work was supported by the National Basic Research Program of China (973) under Grant No.2005CB321801, the National High Technology Research and Development Program of China (863) under Grant No.2009AA01Z142, the National Natural Science Foundation of China under Grant No.60703072, and HKUST research fund SBI08/09.EG03. We would like to also express special thanks to Qing Cao for his technical support on LiteOS.

REFERENCES

[1] P. Dutta, M. Grimmer, A. Arora, S. Bibyk, and D. Culler. Design of a wireless sensor network platform for detecting rare, random, and ephemeral events. In *Proc. of 4th Intl. Conf. on Information Processing in Sensor Networks*, 2005.

[2] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. *Proc. of ASPLOS 2000*, Nov. 2000.

[3] L. Gu, D. Jia, P. Vicaire, T. Yan, L. Luo, T. He, A. Tirumala, Q. Cao, J. A. Stankovic, T. Abdelzaher, and B.H. Krogh. Lightweight detection and classification for wireless sensor networks in realistic environments. In *Proc. of the 3rd ACM Intl. Conf. on Embedded Networked Sensor Systems*, 2005.

[4] T. He, S. Krishnamurthy, L. Luo, T. Yan, and L. Gu et al. VigilNet: An integrated sensor network system for energy-efficient surveillance. *ACM Trans. on Sensor Networks*, 2(1):1–38, 2006.

[5] R. Szewczyk, A. Mainwaring, J. Polastre, and David Culler. An analysis of a large scale habitat monitoring application. In *Proc. of the 2nd ACM Intl. Conf. on Embedded Networked Sensor Systems (SenSys'04)*, Nov. 2004.

[6] D. Gay, P. Levis, R. Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of Programming Language Design and Implementation (PLDI)*, June 2003.

[7] A. Adya, J. Howell, M. Theimer, B. Bolosky, and J. Douceur. Cooperative task management without manual stack management. In *Proc. of the USENIX Annual Technical Conf.*, June 2002.

[8] W. McCartney and N. Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 167–180. ACM, 2006.

[9] A.Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42. ACM, 2006.

[10] C. Han, R. Kumar, R. Shea, E. Kohler, and M. B. Srivastava. A dynamic operating system for sensor nodes. In *Proc. of MobiSys'05*, 2005.

[11] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462. IEEE Computer Society, 2004.

[12] S. Bhatti, J. Carlson, H. Dai, J. Deng, and J. Rose et al. MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms. *ACM/Kluwer Mobile Networks and Applications, Special Issue on Wireless Sensor Networks*, 10(4):563–579, Aug. 2005.

[13] A. Eswaran, A. Rowe, and R. Rajkumar. Nano-RK: An energy-aware resource-centric rtos for sensor networks. In *Proc. of the 26th IEEE International Real-Time Systems Symposium*, pages 256–265. IEEE Computer Society, 2005.

[14] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He. The LiteOS operating system: Towards unix-like abstractions for wireless sensor networks. In *Proceedings of the 7th international conference on Information processing in sensor networks*, pages 233–244. IEEE Computer Society, 2008.

[15] L. Gu and J. A. Stankovic. t-kernel: providing reliable os support to wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 1–14. ACM, 2006.

[16] C. Duffy, U. Roedig, J. Herbert, and C. J. Sreenan. Adding preemption to TinyOS. In *Proceedings of the 4th workshop on Embedded networked sensors*, pages 88–92. ACM, 2007.

[17] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for TinyOS. In *Proceedings of the 5th ACM Conference on Embedded Networked Sensor Systems*, 2007.

[18] R. Kumar, E. Kohler, and M. Srivastava. Harbor: software-based memory protection for sensor nodes. In *Proc. of the 6th Intl. Conf. on Information Processing in Sensor Networks*, pages 340–349. ACM, 2007.

[19] H. Cha, S. Choi, I. Jung, H. Kim, et al. RETOS: resilient, expandable, and threaded operating system for wireless sensor networks. In *Proceedings of the 6th international conference on Information processing in sensor networks*, pages 148–157. ACM, 2007.

[20] J. Yang, M. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 189–203. ACM, 2007.

[21] P. Levis and David Culler. Maté : a virtual machine for tiny networked sensors. In *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, Dec. 2002.

[22] R. Barr, J. Bicket, D. Dantas, B. Du, T. Kim, B. Zhou, and E. Sirer. On the need for system-level support for ad hoc and sensor networks. In *Operating Systems Review, ACM, 36(2):1-5*, Apr. 2002.

[23] Crossbow Technology Inc., http://www.xbow.com/. *MICA2 data sheet*.

[24] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, page 67. IEEE Press, 2005.

[25] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *Proc. of the 2nd USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, May 2005.