SenSmart: Adaptive Stack Management for Multitasking Sensor Networks

Rui Chu, Lin Gu, *Member*, *IEEE*, Yunhao Liu, *Senior Member*, *IEEE*, Mo Li, *Member*, *IEEE*, and Xicheng Lu, *Member*, *IEEE*

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 without virtual memory support. 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 multitask scheduling, memory isolation, and adaptive stack management. Our solution provides memory isolation and automatic stack relocation on usual sensornet platforms. The adaptive stack management frees programmers from the burden of estimating tasks' stack usage, yet it enables SenSmart to schedule and run more tasks than other multitasking OSes for sensor networks. We have implemented SenSmart on MICA2/MICAz motes. Evaluation shows that SenSmart has a significantly better capability in managing concurrent tasks than other sensornet operating systems.

Index Terms—Multitasking, memory management, stack adaptivity, binary translation, kernel

1 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 and application complexity—are manifested to a great extent. A typical sensor node may only have a simple CPU and a few kilobytes of data memory [1], [2], [3], 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 administration [4], [5], [6].

The complexity of application software and the fact that the software runs on numerous unreliable devices call for strong system software support [7], [8]. 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 systems shows severe limitations in both functionality and usability. One of the key problems, as mentioned by a classic research work on the topic of multitasking, is stack management [9]—how can an operating system automatically and efficiently manage multiple stacks. Especially, the problem is even harder 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 of great variation for event-driven systems, which is the *de facto* standard programming model for sensornet systems [10], [11]. The ability to hold multiple stacks in memory and efficiently handle the stack dynamics is a fundamental determinant of a multitasking OS.

On resource-rich platforms, stack management has known solutions. Three facts have helped the traditional stack management become successful on such systems:

- Virtual memory support in modern microprocessors eliminates external fragmentation, and limits stack collisions to only occur within individual address spaces.
- It provides abundant virtual memory space for typical stack usage in modern computer architectures. Hence, interthread stack collisions inside a process can be avoided by allocating sufficient virtual memory areas to the threads.

- *R. Chu and X. Lu are with the National Laboratory for Parallel and Distributed Processing, School of Computer, National University of Defense Technology, PDL Building, ChangSha, HuNan 410073, China. E-mail: {rchu, xclu}@nudt.edu.cn.*
- *L. Gu and Y. Liu are with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. E-mail: {lingu, liu}@cse.ust.hk.*
- *M. Li is with School of Computer Engineering, Nanyang Technological University, N4-02C-108, 50 Nanyang Avenue, Singapore 639798. E-mail: limo@ntu.edu.sg.*

Manuscript received 27 July 2010; revised 23 Sept. 2011; accepted 2 Oct. 2011; published online 14 Dec. 2011.
Recommended for acceptance by K. Roemer.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2010-07-0421.
Digital Object Identifier no. 10.1109/TC.2011.238.

- 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 sensor 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 not only extra burden on application programmers, but also 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 adaptivity*—a term we use in this work to describe the ability to efficiently handle multiple stacks without a priori knowledge on their dynamics. The weak stack adaptivity directly affects these OSes' ability to accommodate concurrent tasks.

Designing adaptive stack management on resource constrained platforms is a new challenge, one important question is whether we could avoid this problem by upgrading hardware to qualify traditional solutions. 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, and many low-power systems 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 FLASH-based external storage, while the writing overhead, as well as the limited erase cycle of FLASH chips, make the 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 [12]. The use of fibers simplifies the scheduler design but does not eliminate the need for stack management [9]. Some other solutions, such as the protothreads [13], have their own limitations, as we will cover in Section 2.

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

The major contributions of this work are as follows:

- We present a solution to adaptive stack management and provide important OS features, such as memory isolation, on small-memory systems without virtual memory hardware support.
- A base-station-side binary rewriting approach is proposed. It minimizes the kernel resident in sensor nodes and significantly reduces the per-node resource consumption as compared with earlier works.
- We have implemented SenSmart on MICA2/MICAz motes. Detailed evaluation is performed on this implementation.

The rest of this paper is organized as follows: Section 2 discusses the related work. Section 3 presents an overview of our system approaches and architecture. Section 4 describes the detailed design of SenSmart. Section 5 focuses on the system implementation and evaluation. We summarize the work in Section 6.

2 Related Work

Researchers have developed a number of operating systems for sensor networks and low-power devices, such as TinyOS [3], SOS [14], Contiki [15], MANTIS OS [16], Nano-RK [17], SESAME/SESAME-P [18] [19], LiteOS [20], and the t-kernel [21]. In order to support more reliable, efficient, and sophisticated applications, multitasking has become an important feature in such systems. However, for reasons explained in Section 1, existing multitasking systems for sensor networks have to place harsh restrictions on the concurrent tasks. Table 1 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.

In TinyOS [3], tasks are executed in serial. Hence, there is no concurrency among them, and the stack management is rather simple. Moreover, the memory isolation is absent so that the program code can write to any physical memory areas. To improve the quality of system services, many works attempt to add diverse features for TinyOS [12], [22], [23], [24]. As a representative focus, several preemptive multitasking models are proposed, respectively. For example, TOSThread [24] introduces user threads along with existing TinyOS tasks. Each thread is allocated an independent but fixed-size stack for local variables and execution context. Substantially, such multitasking models are tailored from the traditional design techniques, while they often work inefficiently in resource constrained systems as described in Section 1.

TABLE 1
Comparison of Typical Systems

	Maté	Contiki	MANTIS	t-kernel	RETOS	LiteOS	TOSThreads	SenSmart
TinyOS Compatible	No	No	No	Yes	No	No	Yes	Yes
Preemptive Multitasking	No	Yes	Yes	Partial	Yes	Yes	Yes	Yes
Concurrent Applications	N/A	No	No	No	No	No	No	Yes
Interrupt-free Preemption	N/A	No	No	Yes	No	No	No	Yes
Memory Protection	Yes	No	No	Partial	Yes	No	No	Yes
Logical Memory Address	N/A	No	No	No	No	No	No	Yes
Memory Arrangement	Automatic	Automatic	Automatic	Automatic	Automatic	Manual	Automatic	Automatic
Stack Relocation	No	No	No	No	No	No	No	Yes

Some other sensornet operating systems, such as MANTIS OS [16], SOS [14], [25], RETOS [26], and LiteOS [20], also attempt to adopt traditional OS mechanisms as TOSThread does. Those traditional solutions usually lead to 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 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. The fundamental reason is the weak stack adaptivity, and consequently, limited application flexibilities.

Researchers have noticed that the traditional monolithic stack allocation can reduce the overall efficiency. To improve the flexibility, SESAME/SESAME-P [18], [19] propose novel solutions to convert the call stack into the heap area, and perform bookkeeping to manage the discrete stack blocks. The runtime overhead is mitigated by a flexible dynamic stack allocation mechanism. Capriccio [27] also explores noncontinuous stack configurations for general purpose systems. In contrast, SenSmart still maintains the continuity of the per-task stack space, and adjusts the overall stack utilization by logical addressing and stack relocation. For applications with complex stack usage, such as the pointer arithmetic in the stack area or optimized PASCAL-like programs, the continuous-stack structure unifies the abstraction at the system level with that at the programming language level, and would avoid special handling in the memory management logic.

Lightweight thread models can avoid the stack management problem by dramatically simplifying the semantics of concurrent tasks. For instance, the stackless protothreads in Contiki minimizes memory usage [13], [15], but they also incur severe functional limitations, e.g., no retention of state between context switches. Such limitations are likely to make programming harder.

Maté [28] and MagnetOS [29] 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.

As our previous work, the t-kernel [21] implements preemptive scheduling, OS protection and virtual memory with binary rewriting 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 rewriting as an important technique to implement preemptive scheduling and memory isolation. Different from the t-kernel, SenSmart conducts complete binary translation on the base station. As we will show later, this approach brings unique advantages in reducing system complexity and code inflation ratio.

3 Overview

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

3.1 Examination of the Design Space

We use the MICA2/MICAz sensor nodes [30] 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, 4 KB SRAM-based data memory, and 128 KB FLASH-based program memory.

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

- The application code does not modify any existing instruction. 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 application does 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 reallocates parts of it upon requests, to emulate the dynamic memory function.

In this strictly resource constrained design space, we design SenSmart to be an OS with solid multitasking capability. Using software methods, it solves the critical

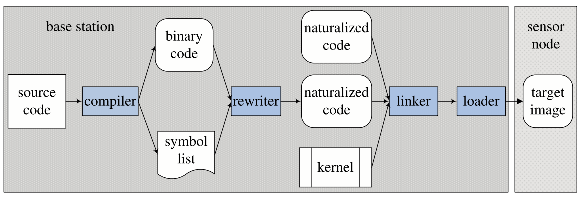


Fig. 1. The compiling, rewriting, and loading process in SenSmart.

problems of stack adaptivity and software-based preemptive scheduling. It is worth explicitly listing the challenges in order to clarify the design space and help understand our design choices:

- Strictly constrained energy budget makes it very difficult to dramatically increase the data memory size.
- Absence of virtual address translation hardware makes it impossible to accommodate the growth and shrinkage of multiple stacks without fragmentation or stack collisions.
- Absence of instruction privilege support makes it impossible to use traditional OS design techniques.

3.2 System Overview

A sensornet application is often written in a specific programming language, such as nesC [10]. After being compiled into a binary executable, the application code 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 rewriting logic, called rewriter, analyzes the binary image, and modifies the application code to ensure that multiple application tasks run on one node following appropriate multitasking semantics.

Fig. 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 precompiled system kernel, which includes the kernel runtime, to form the executable image 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 task has its respective time slice and memory region. Without relying on clock interrupts or virtual address translation, 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.

It is worth noting that the program rewriting and dissemination do not significantly increase the energy consumption in typical sensornet deployments. The program analysis and rewriting are performed on the base station, which is usually provisioned with sufficient energy supply. The final executable is either directly uploaded or disseminated as part of the wireless reprogramming, which is already needed for programming a large number of nodes [31], [32], [33].

4 SENSMART DESIGN

In this section, the design details of SenSmart will be presented. We first introduce how SenSmart performs binary rewriting on the base station, then briefly cover multitask scheduling, and present the details of the memory management.

4.1 Binary Rewriting on the Base Station

Following common nesC and C 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 rewriting 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 rewritten 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 rewritten in a way that cooperates with the memory management mechanism.
- The instructions that access some OS-reserved resources are also rewritten. 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.

Departing from the on-node binary rewriting 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, by performing rewriting after compiling and before program distribution, the base station can collect the whole-program characteristics. One example is the position and length of the program memory data, which is embedded in the program code as constants. Obviously, such program data should not be identified as instructions and thus be modified. SenSmart can easily skip such cases when rewriting, while the t-kernel has to introduce complex and exhausting schemes for that. Another example is the heap usage information from the symbol list generated in compiling. As mentioned later, such information is useful for logical addressing in our approach.

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

Third, by moving the code rewriting logic to the base station, SenSmart also significantly reduces the kernel size on individual sensor nodes. By keeping the kernel small, we allow more application code to reside in the program memory.

Finally, SenSmart maintains an approximate linearity of the instruction addresses between the original and the naturalized program. After the rewriting, one instruction can be translated into a variable number of instructions, and this usually results in a code inflation nonlinear to the instruction addresses. SenSmart regularizes the instruction rewriting to mitigate the inflation. When modifying 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 rewritten logic. All of the trampolines are appended after the application program. Hence, the instruction count of the modified 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 modified 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.

Meanwhile, rewriting on the base station has its limitations—it cannot utilize runtime information. For example, the target address of an indirect branch or an indirect memory access cannot be known by the rewriter on the base station. These issues need to be resolved at runtime by the kernel.

4.2 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 backward branch instructions when rewriting the application code, to construct software traps so that the branches will jump to the OS kernel before their target addresses. The kernel routine maintains the time slice for each task using the reserved Timer3 clock counter,[1] and a task will be preempted *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 depend on the dynamic instruction flow and the backward branches, which may not occur exactly at the absolute end of a slice. However, the delay of the preemption, usually no more than a few milliseconds, is small enough to be ignored for most applications. It is also noteworthy that, to make the scheduling fair for all tasks, SenSmart will dynamically adjust the time slice to compensate the preemption delay and avoid error accumulating. Moreover, the 16-bit Timer3 is large enough for the scheduling frequency, thus the counter overflow can also be handled easily without an interrupt routine. Therefore, even with interrupts disabled, SenSmart can still preempt the tasks by the software traps and schedule them properly.

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

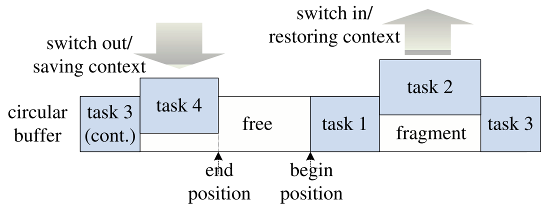


Fig. 2. Circular buffer for context saving and restoring.

As a matter of fact, we have also noticed that such interrupt-free preemption might be a mixed blessing. It provides fairness for scheduling, while some sensory or communication component, which intends to disable the interrupts to ensure atomicity, may also be preempted by the kernel unexpectedly. Although the drawback is mostly not significant because its consequence, such as occasional packet loss, can be handled properly by usual application logic, SenSmart also provides APIs to denote a *critical section*, which will not be preempted by the kernel. By such means the issue is mitigated effectively in our practice. Moreover, as a future work, we intend to migrate the control logic of the peripheral resources, such as timer, radio, and external FLASH storage, to the OS kernel services, and letting application tasks access the virtualized hardware using system calls. Such a mechanism could solve the problem of broken atomicity since it prevents the application code from manually controlling the physical peripherals using atomic operations.

When SenSmart schedules a task to run, the kernel must save the execution context of the current running task. The execution context, about 50 bytes including CPU registers, CPU flags, and some I/O registers relevant to program status, is a sizable structure pressuring the scarce physical memory on low-power sensor nodes as the number of tasks grows. SenSmart employs run-length compression on task contexts to save memory space. Since there are often unused or copied registers, sequences of identical values are not uncommon, giving opportunities for compression.

To avoid the possible risk of stack overflow in a small-memory system, SenSmart uses a shared circular buffer, instead of the task's stack, to save contexts, as illustrated in Fig. 2. Upon a context switch, the kernel compresses the context of the switched-out task, saves it at the current end position of the circular buffer, and then restores the context of the task to be switched in.

The size of context of each application task varies after compression, and imposes a challenge on the context management algorithm. When there are fewer than two application tasks or the scheduling is strictly round robin, the context of the task to be switched in is always at the beginning position of the circular buffer. This makes the context management very efficient even when the sizes of task contexts vary. However, we can also easily extend the scheduling policy to a nonround-robin one. In such cases, external fragment may appear and memory space may be wasted, as illustrated in Fig. 2. A defrag routine is designed in SenSmart, to detect and eliminate the fragment by rearranging the context.

In our practice, the features of context compression and fragment elimination are optional, as they introduce extra overheads, and, more importantly, there is a risk of circular buffer overflow, since the compression ratio cannot be

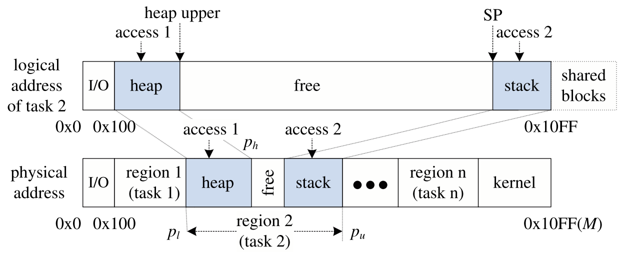


Fig. 3. Layout of data memory and logical addressing.

predicted and an optimistically estimated buffer size may be exhausted. The programmer can switch off the compression when the memory budget is not so strict.

4.3 Memory 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 stack. Without stack adaptivity, such stack collisions cannot be resolved, 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 in a multitasking OS.

In order to adapt to the dynamics of stack usage, SenSmart introduces software-based logical addressing, and automatically adjusts various 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 mechanisms.

4.3.1 Memory Organization

SenSmart divides the physical 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 3.1), 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. The lower half of Fig. 3 shows the structure of physical 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. Except for explicitly using the memory sharing APIs provided by SenSmart, a task cannot touch others' memory regions. 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 $p_l < p_h < p_u < M$. The data memory layout and the pointers for all tasks are initially determined by the rewriter, and the SenSmart kernel maintains these pointers at runtime. After excluding the I/O area, the kernel area and all heap areas, the remaining space is partitioned into stacks for all tasks.

4.3.2 Logical Addressing

SenSmart uses a logical addressing mechanism to provide each application task a logical memory space, and the task can exclusively use it. The logical addresses are translated into physical addresses at runtime, 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 and sizes of the memory regions 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 runtime address translation. Under the assumptions listed in Section 3.1, there are only three types of valid data memory accesses: 1) random access in the current heap area; 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 Fig. 3 and described as follows:

Generally, the address translation adds a displacement (p_l for heap and $p_u - M$ 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 M as stack bottom. The kernel will also translate the logical address 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 logical memory spaces for different tasks are isolated. For the tasks that intend to share data, SenSmart also provides an API, which works like the POSIX *mmap* system call, to explicitly map multiple logical address blocks to the same physical address block. Those shared blocks reside in the upper address region of the logical memory space as shown in Fig. 3.

Such software-based data memory address translation incurs overhead for extra instructions and memory accesses. We have noticed that, in most sensornet applications, two or four 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.

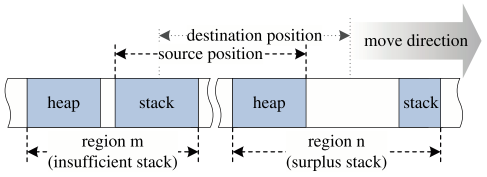


Fig. 4. Stack relocating when an application has insufficient stack space.

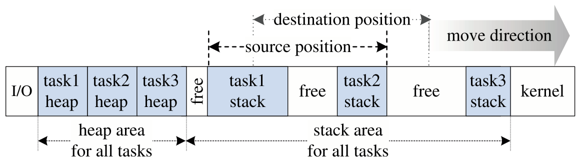


Fig. 5. Data memory arrangement to aggregate heaps and stacks together.

Despite the overhead, the benefits of the memory indirection are multifold. 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 4.3.3. For many sensornet applications, data memory is always a keenly constrained resource, while CPU cycles are not lacked. Hence, for such light load applications, we believe the benefit of logical addressing in system functionality, reliability, and usability by far outweighs the overhead in CPU cycles.

Similar to data memory, the program memory address translation is also necessary for the correctness of the naturalized program. At runtime, the translation of program memory address occurs when there is an indirect program memory data access or 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, and it is modified into a 32-bit JMP/CALL instruction when necessary, we use a sorted array called *shift table* to record the addresses of rewritten instructions that are inflated from a 16-bit instruction to a 32-bit 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. For an application program with I instructions, if $m\%$ of the instructions are 16-bit instructions that need to be modified, the space consumption of the shift table, L_S, is given by $L_S = 2 \times I \times m\%$, and the time complexity of program address translation is $O(log(L_S))$ for binary searching in the shift table.

4.3.3 Stack Relocation

Most sensornet application programs typically use the stack in a highly dynamic manner. In sophisticated TinyOS applications, 10 levels of nested function calls are common even with compiler inline optimization. Such programs often result in a sizable stack area.

Following a split-phase pattern for event-driven processing, tasks can quickly shrink their stack on a blocking I/O request, 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 the fixed-size stack management cumbersome and inefficient.

One important technique SenSmart uses to enhance stack adaptivity 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 an evenly partitioned 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 runtime, the binary rewriter modifies 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 Fig. 4. 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. With logical addressing, the application tasks' memory access semantics is maintained.

As we will evaluate in Section 5, the stack relocation is rather time consuming since it will move a trunk of memory. Actually, since the physical addresses and logical addresses are uncoupled by the address translation, we can rearrange the data memory areas to aggregate the heaps and stacks for different tasks together, as shown in Fig. 5. Such an arrangement may reduce the overhead of the stack relocation since less bytes will be moved. However, it increases the logical addressing overhead because we should maintain four boundaries for a task's memory region, and extra memory access overhead will be thus introduced. As the logical addressing occurs more frequently than the stack relocation, such an arrangement is, in practice, less efficient than our current design.

5 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 and TinyOS applications, to evaluate typical code in sensor networks. Third, we assess the multitask scheduling performance when multiple concurrent tasks execute. Finally, we show that SenSmart has a feasible stack adaptivity in accommodating concurrent tasks.

TABLE 2
Overhead of Key Operations

Operation			Cycles	Time(μs)
System initialization			5738	783.88
Memory address translation	Direct	I/O area	2	0.27
		Others	28	3.83
	Indirect	I/O area	54	7.38
		Stack frame	69	9.43
		Heap	62	8.47
		Program mem.	376	51.37
Stack operation	Get stack pointer		45	6.15
	Set stack pointer		94	12.84
	Stack relocation		2326	317.76
Context switching	Context saving		932	127.32
	Context restoring		976	133.33
	Full switching		2298	313.93
	Context defrag		2319	316.80

5.1 Overhead

We have implemented SenSmart on MICA2/MICAz motes, and made the source code available [34]. With only very basic hardware assumptions, SenSmart should be able to be ported to target platforms with ATmega128L MCUs without much difficulty. The SenSmart kernel is configurable. In the default setting, it occupies less than 6 percent of the program memory and about 10 percent of the data memory. This memory footprint is much smaller than our previous work, the t-kernel, which uses more data memory to perform on-node rewriting.

The task scheduling and memory management 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 the corresponding execution time on the 7.32 MHz ATmega128L MCU is also calculated. The results are listed in Table 2.

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 amortized within basic blocks as discussed in Section 4.3.2. Indirect branches have high overhead due to branch destination lookup at runtime, but such instructions are rare in current sensornet applications. The overheads of stack relocation and context switching vary in different cases. The numbers shown in Table 2 give representative examples. It is worth noting that relocating a stack on an ATmega128L MCU may introduce $100 - 400$ μs 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 confident that occasional submillisecond delay paid for an unprecedentedly adaptive multitasking support is a small and welcomed cost.

TABLE 3
Kernel Benchmark Programs

Name	Function	Application
am	Active messaging	Network protocol
amplitude	Signal processing	Sensing
crc	CRC calculation	Network protocol
eventchain	Event dispatch	All TinyOS apps.
lsfr	Random number	Various apps.
readadc	Read analog sensor	Sensing apps.
timer	Timer event dispatch	Periodic tasks

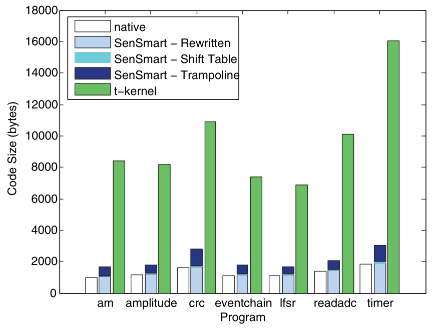


Fig. 6. Code inflation of kernel benchmark programs.

5.2 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 [21]. As listed in Table 3, these programs cover typical operations in sensornet applications. Fig. 6 analyzes the code inflation of the kernel benchmark programs under SenSmart and the t-kernel, as compared with the native code size. The code inflation under SenSmart is within 200 percent. As a comparison, the t-kernel, which also uses the binary translation, makes the code size much larger. The reason is that SenSmart conducts translation on base station, and can make translated code much more optimized in terms of space efficiency.

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. Instead, SenSmart aims at providing stack adaptivity, memory protection, and flexible multitasking capability. 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 the t-kernel has better performance in most of the seven programs as Fig. 7 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 1.

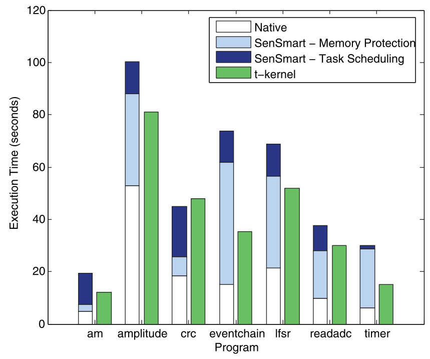


Fig. 7. Execution time of kernel benchmark programs.

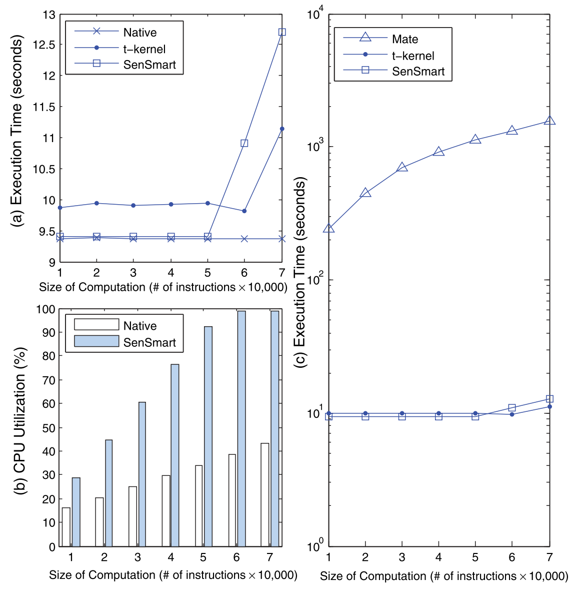


Fig. 8. Execution time and CPU utilization of *PeriodicTask* program.

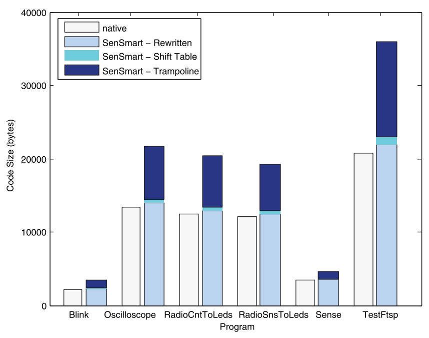


Fig. 9. Code inflation of TinyOS applications.

Although the performance disadvantage of SenSmart emerges in the seven kernel benchmark programs, which perform intensive computation during the whole working process, in realistic energy constrained sensor nodes, most applications have a periodic events triggering pattern [35], [36]. For those applications, the extra CPU cycles spent in SenSmart only modestly reduce the CPU idle time, and do not affect the overall application performance.

We use a *PeriodicTask* program to emulate such common operating pattern of sensornet applications. Using the 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 program. 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 on real sensor nodes. Moreover, we use the *avrora* [37] simulator to measure the proportion of the active CPU 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 system kernel overhead, and results in the work of t-kernel, are also listed. As shown in Fig. 8a, 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. Fig. 8b 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 logical addressing 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 percent, which is the common case in current sensornet applications. For the computation-intensive applications, there is a tradeoff between stack adaptivity, multitasking capability, and power consumption. SenSmart may not be a suitable solution for those applications.

It is noteworthy from Fig. 8a 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 rewriting overhead, which introduces a considerable initialization delay. Even without that, the t-kernel performs almost the same than SenSmart for applications that are not computation intensive. Fortunately, many current sensornet applications have light CPU utilizations. The detailed results will be shown in Section 5.3.

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 Fig. 8c, in which the Y-axis is exponential. The execution time of *PeriodicTask* program in Maté is much higher than in t-kernel and SenSmart. As a fully virtualized environment, the virtual machine can also enhance reliability and ensure memory protection [38]. But the interpretation-based execution has a significant performance penalty, as indicated by the difference in execution speed. Overall, SenSmart is an efficient design among software-based solutions for general application programming.

5.3 TinyOS Applications

Different from the kernel benchmark programs, which simply emulate the common operations, and execute with a run-to-complete style, most sensornet applications developed for TinyOS are event-driven—they contain routines to be invoked periodically, and execute without an explicit exit. We use six sample applications provided in TinyOS 2.1 to assess their overheads running in SenSmart. The results are shown in Figs. 9 and 10. Note that instead of the execution time, the CPU utilization simulated in *avrora* over an execution periods of 3 minutes depicts the active execution cycles for such applications in Fig. 10. Obviously, the code size of real applications, which indicates the program complexity to a certain extent, is much larger than the kernel benchmark programs. However, the code inflation ratio remains roughly unchanged, and the inflated size still can be accommodated by MICAz nodes with 128 KB program memory. The CPU utilizations of those applications are also low enough, which indicates that SenSmart fits for such applications with acceptable overhead.

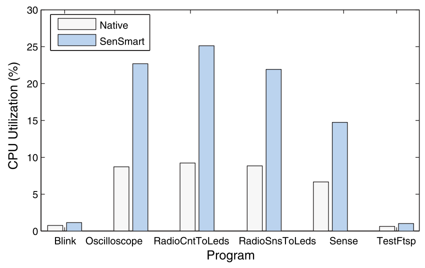


Fig. 10. CPU utilization of TinyOS applications.

It is noteworthy that the *TestFtsp* application has the largest code size but lowest CPU utilization in Fig. 10, because its primary workload is driven by beacon messages from another node, which cannot be simulated accurately by *avrora*. To reflect the real case, we deploy sensor nodes to study the execution of the *TestFtsp* and *Oscilloscope* applications on SenSmart. We use totally eight MICAz nodes for this experiment, one of them acts as the beacon node for *TestFtsp*. Two of them connected to PC act as base stations for receiving the messages from *TestFtsp* and *Oscilloscope*, respectively. The rest five nodes are tested under different configurations where *TestFtsp* and *Oscilloscope* execute with or without SenSmart. We also examine the case that these two applications concurrently executing under the scheduling of SenSmart. Each experiment is repeated for four rounds and the results are shown in Fig. 11. The time synchronization processes of *TestFtsp* under different configurations, as depicted by Figs. 11a, 11b, and 11c, are rather random even within the same configuration (plotted in one subfigure), since it depends on several factors such as the initial clock offset and skew. But in general, the results in Figs. 11a, 11b, and 11c show similar application-level performance in terms of converging time and the synchronization error. Fig. 11d plots the number of messages from *TestFtsp* and *Oscilloscope* to base stations under different configurations, and shows that the message counts are consistent with a fluctuation within 8 percent for *TestFtsp* and roughly unchanged for *Oscilloscope*. Therefore, although execution on SenSmart inevitably introduces overheads, such cost is acceptable and does not noticeably affect the application-level behaviors.

5.4 Multitasking Performance

To examine the concurrent execution of multiple tasks in SenSmart, we run the *PeriodicTask* together with each kernel benchmark programs, and plot the completion time of the tasks under increasing computation sizes in *PeriodicTask*. Fig. 12 shows the results.

The dotted line and dashed line in each plotting denotes the individual execution times of all the tasks if they are executed in a single-task environment. The sum of individual execution times, denoted with the dash-dotted line, depicts the execution time for the tasks to execute serially on the sensor node without any context switching overhead. Intuitively, the actual execution time of concurrent tasks, with the context switching overhead, should be longer than the sum of individual execution times. However, in some cases, both concurrent tasks are completed earlier than the sum value. The reason is that *PeriodicTask* looks for the task queue after each computational task completes. If the queue is empty, it yields the CPU before its time slice is used up.

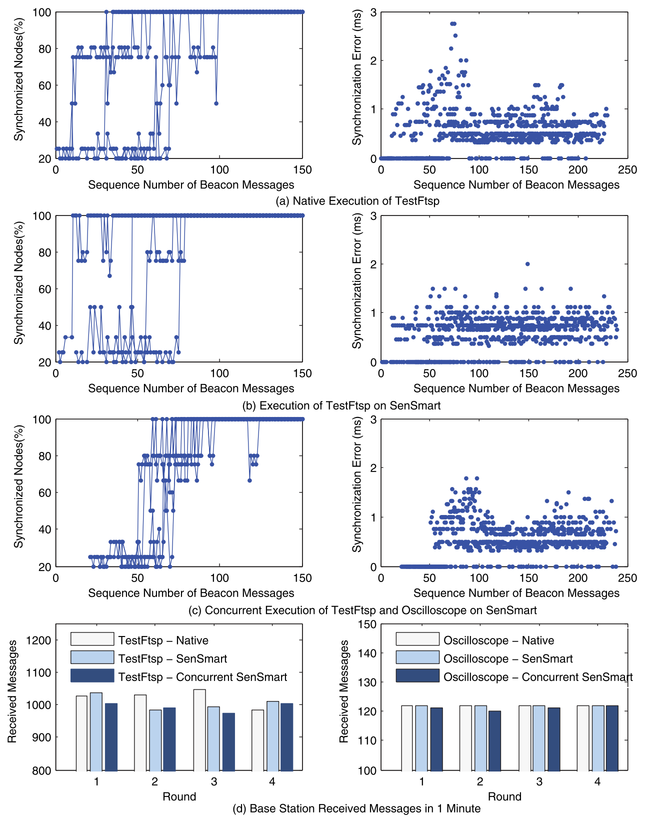


Fig. 11. Execution of *TestFtsp* and *Oscilloscope*.

When *PeriodicTask* is running with lower workload, the kernel benchmark program has more opportunities to be executed, as illustrated in Fig. 13a, where P denotes the execution of *PeriodicTask* program, and K denotes the other concurrent kernel benchmark program. In that case, the time slice is utilized more effectively, and this reduces the completion time.

Another interesting observation from Fig. 12 is that the complete time of *PeriodicTask* drops dramatically and reaches the minimal value when the computation size is 50,000 instructions. The reason can also be explained by Fig. 13. When the execution time of each task is long enough, the timer-driven *PeriodicTask* generates the next task soon enough for that task to be executed right after the current one, resulting in a higher utilization of time slices. As illustrated in Fig. 13b, *PeriodicTask* does not yield CPU voluntarily, and it continuously executes until being preempted. Therefore, more computational tasks can be executed in a fixed time span. When the computation size is larger than 60,000 instructions, however, fewer tasks can be completed within one time slice, so that the total execution time also increases.

Clearly, the length of the time slice is an important factor for the preemptive scheduling. In the above experiment, the time slice is configured to 1 second. We now study the effect of shorter time slices by varying it to $1/2, 1/4, \ldots, 1/256$ second, respectively, and execute the *PeriodicTask* in four typical computation sizes with two kernel benchmark programs. To make it roughly random, we choose the first and last programs in alphabetical order, *am* and *timer*. Fig. 14 shows the execution completion times of the three programs.

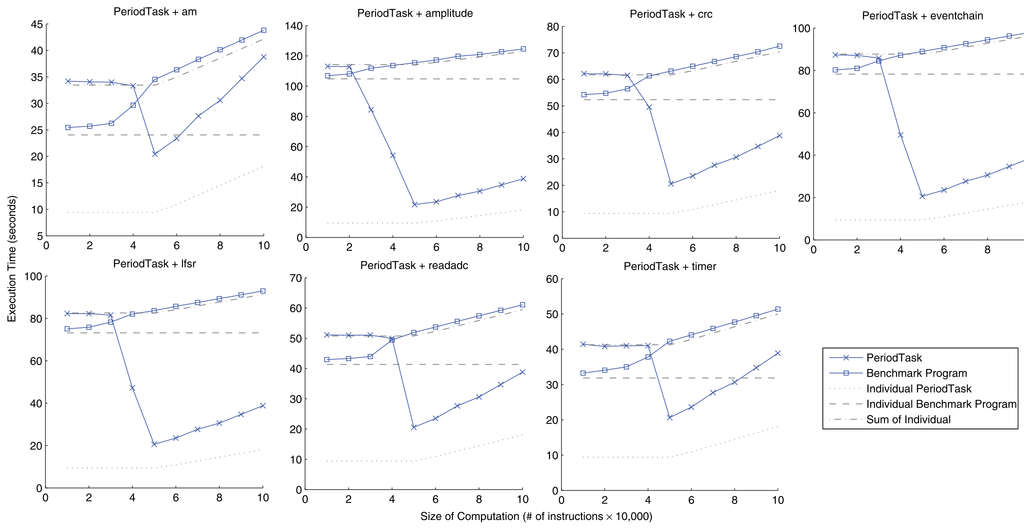


Fig. 12. Execution completion time of *PeriodicTask* with different kernel benchmark programs.

Considering the overhead of context switching, we would imagine that the performance should suffer with smaller time slices. Surprisingly, all of them execute a little faster with the time slice of 1/4 or 1/8 second than that of 1 second or 1/2 second. As listed in Table 2, a full context switching will introduce a considerable overhead. Such overhead impacts the system performance slightly when the time slice is longer than 1/8 second. Generally, shorter time slices provide more precise scheduling control. After the key point of 1/64 second, the completion sequence of *am* and *timer* exchanges their position in all of the four subplots. It can be explained in Fig. 7, where we can find that *am* has much higher proportion of task scheduling overhead than *timer*. Hence, the former will be more seriously impacted when the context switching overhead increases.

As described in Section 4.2, the executing tasks are preempted by SenSmart kernel using the modified branches. By such means, a small delay is introduced on each time slice, and causes the scheduling error. We collect the statistic delay time for the seven kernel benchmark programs, and compare them with a moderate time slice, 1/16 second. The results are listed in Table 4. Although such delay depends on the program control flow and can hardly be predicted, it has the same order of magnitude with the context switching overhead, which consumes more than 300 μs. Obviously, relative to the time slice, the scheduling error is small enough and ignorable in most cases.

5.5 Stack Adaptivity

As studied in Section 5.1, each stack relocation costs about $100 - 400$ μs. In this section, we will evaluate the effects and impacts of adaptive stack management.

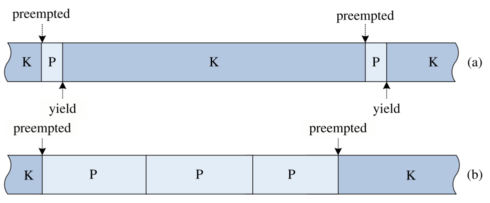


Fig. 13. *PeriodicTask* program with different workloads.

A common workflow in a sensornet application follows a *sense-and-send* paradigm in an event-driven style [39]. After sensor and radio channels feed data to the sensor node, various event-driven handlers are triggered to read the data, verify them, and usually, store 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, such as compression, routing, signal processing, and these tasks are activated upon different conditions.

We use a set of tasks with different stack dynamics, including one data feeding task and several processing tasks, to approximate such a sense-and-send workflow. The data feeding task periodically stores randomly generated data onto the heap to form six binary trees, and then, using the synchronization and memory sharing APIs provided by SenSmart, the processing tasks are activated to map the binary trees from the data feeding task into the logical spaces of themselves, and recursively search randomly selected 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 adds 15 bytes to the stack; hence, the historically largest stack size of the search tasks is around 180 bytes. Based on our observations on VigilNet and other sensornet applications, such a moderate stack size is typical for processing tasks.

Fig. 15 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 available stack space has to be reduced. Meanwhile, 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.

As mentioned before, a search task needs about 180 bytes of stack on average, while we can also observe from Fig. 15

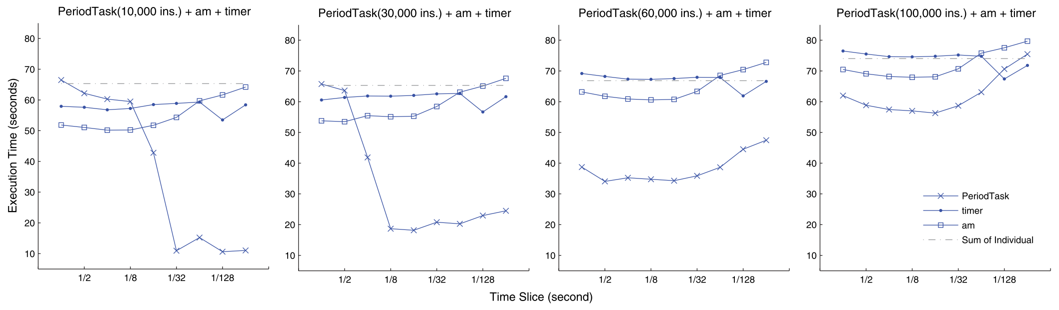


Fig. 14. Execution completion time of three tasks with different time slice length.

that the average stack allocation in all cases does not exceed 130 bytes. It implies that 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. Particularly, when the size of the binary tree is 58 bytes with nine concurrent search tasks, the size of a task's stack, on average, is only about 97 bytes, 46 percent lower than the 180 bytes required, on average by each task. SenSmart can still accommodate all of the nine search tasks and one data feed task, with 45 times of stack relocations. Such a behavior shows the stack adaptivity and proves the stack management's effectiveness.

When the average stack allocation is significantly smaller than the required stack size, SenSmart terminates one task since the stack relocation no longer works and not all concurrent tasks can be accommodated in the system. After a task is terminated, the average stack space for each remaining task increases because their stack space allocations can grow to use the released memory of the terminated task. Currently, SenSmart simply terminates the recent task that caused the stack relocation failure. It is technically feasible to extend this work to prioritize tasks and terminate the lowest priority task first under insufficient stack allocation space.

SenSmart performs more stack relocations to tune the sizes of the stacks of concurrent tasks, when the initial stack allocation is severely inadequate. Nevertheless, the maximal number of stack relocations is under 50 times in our experiments; thus, the performance penalty of stack relocation is acceptable. Fig. 16 depicts the average execution time of the binary tree searching tasks in the experiment above. Although the tasks complete slightly later with heavy stack relocations, we believe such a cost is worthwhile in order to improve the system resilience. Moreover, it is noteworthy that the search tasks are still computation intensive. For the event-driven sensornet applications with light workload, such performance impact will be hidden by the idle time as shown in Section 5.2. In the case that the programmer or compiler can approximately estimate the stack usage for each task, and predefine a more or less appropriate initial stack size, the number of stack relocations, as well as the performance overhead, will be further reduced.

The overhead of stack relocation inevitably increases the execution time of a segment of code and the jitter. If significant, may disrupt time-sensitive routines, such as the radio communication. In order to examine whether this would introduce degradation in system performance, we conduct an experiment using a transmitter program, which periodically sends packets, and a corresponding receiver program, which runs concurrently with stack-consuming binary tree searching tasks. In this experiment, the search tasks are adjusted to execute continuously, so that the receiver is always affected by the impact of stack relocation.

TABLE 4
Scheduling Error of Kernel Benchmark Programs

Program	Average delay(ms)	Relative error	Program	Average delay(ms)	Relative error
am	0.55	0.88%	*lfsr*	1.05	1.68%
amplitude	1.36	2.17%	*readadc*	1.03	1.65%
crc	0.59	0.95%	*timer*	0.64	1.02%
eventchain	0.51	0.81%	Relative to 1/16 sec. time slice		

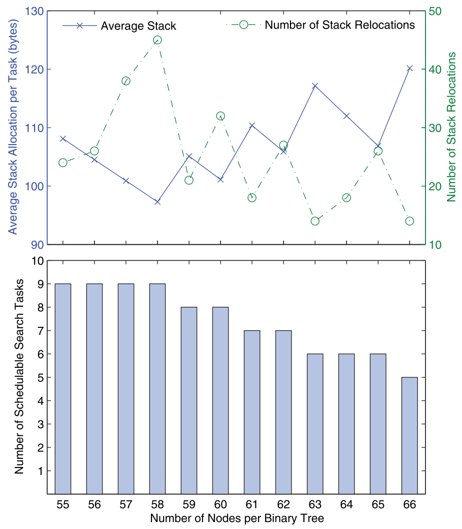


Fig. 15. Binary tree search in SenSmart with increasing tree sizes.

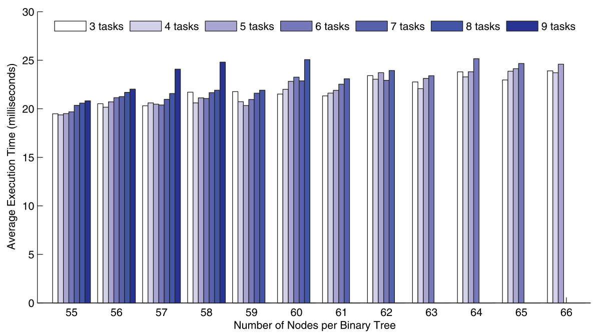


Fig. 16. Execution time of binary tree searching tasks.

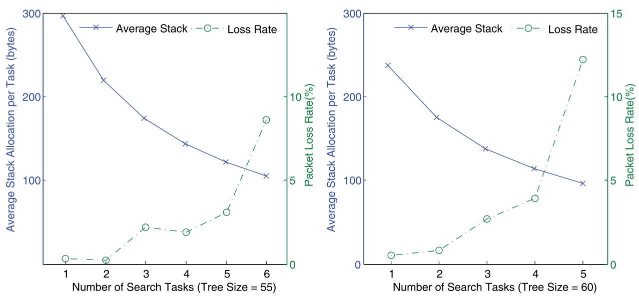


Fig. 17. Radio communication with binary tree searching tasks.

As shown in Fig. 16, the experimental results indicates that the packet loss rate remains lower than 5 percent until the memory is extremely stressed (e.g., six search tasks with a tree size of 55 bytes). The results in Figs. 16 and 17 also show that, the performance impact would become nonlinear when the memory is severely overcommitted, and the risk of task termination as a result of memory overflow would also increase. To avoid such extreme overcommitment, users should certainly exercise caution when deciding the number of concurrent tasks running on a single sensor node. Meanwhile, with SenSmart, the cumbersome (and intrinsically uncomputable) work of estimating individual tasks' stack usages is no longer necessary.

6 Conclusions

Multitasking is a useful system function for complex sensornet applications. It is not easy to implement flexible multitasking using traditional approaches on sensor nodes. SenSmart is a multitasking operating system which solves the critical stack management problem, and improves the preemptive scheduling capability with a set of techniques. We have implemented and evaluated SenSmart. The OS exhibits the ability of CPU and memory multiplexing. Such features will efficiently support concurrent high-quality services in very low power systems.

Acknowledgments

The work was supported by National Natural Science Foundation of China under Grant No. 61003076, National Basic Research Program of China (973) under Grant No. 2011CB302601, National High Technology Research and Development Program of China (863) under grant No. 2011AA010100, the HKUST AEF and SBI (SBI08/09.EG03) grants, COE SUG M58020025, NAP SUG M58020033 from Nanyang Technological University, and the Singapore NRF/EWI grant IDD 90301/1/22.

References

[1] S. Lin et al., "Efficient Indexing Data Structures for Flash-Based Sensor Devices," *ACM Trans. Storage*, vol. 2, no. 4, pp. 468-503, 2006.
[2] 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," *Proc. Fourth Int'l Conf. Information Processing in Sensor Networks,* 2005.
[3] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System Architecture Directions for Network Sensors," *Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* 2000.
[4] M. Eltoweissy, D. Gracanin, S. Olariu, and M. Younis, "Agile Sensor Network Systems," *Ad Hoc and Sensor Wireless Networks,* vol. 4, no. 1, pp. 97-124, 2007.
[5] T. He et al., "VigilNet: An Integrated Sensor Network System for Energy-Efficient Surveillance," *ACM Trans. Sensor Networks,* vol. 2, no. 1, pp. 1-38, 2006.
[6] R. Szewczyk, A. Mainwaring, J. Polastre, and D. Culler, "An Analysis of a Large Scale Habitat Monitoring Application," *Proc. Second Int'l Conf. Embedded Networked Sensor Systems,* 2004.
[7] K. Römer and J. Ma, "PDA: Passive Distributed Assertions for Sensor Networks," *Proc. Eight Int'l Conf. Information Processing in Sensor Networks,* pp. 337-348, 2009.
[8] M. Khan et al., "Diagnostic Powertracing for Sensor Node Failure Analysis," *Proc. Ninth Int'l Conf. Information Processing in Sensor Networks,* pp. 117-128, 2010.
[9] A. Adya, J. Howell, M. Theimer, B. Bolosky, and J. Douceur, "Cooperative Task Management without Manual Stack Management," *Proc. USENIX Ann. Technical Conf.,* 2002.
[10] D. Gay, P. Levis, R. Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC Language: A Holistic Approach to Networked Embedded Systems," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* 2003.
[11] O. Kasten and K. Römer, "Beyond Event Handlers: Programming Wireless Sensors with Attributed State Machines," *Proc. Fourth Int'l Conf. Information Processing in Sensor Networks,* 2005.
[12] W. McCartney and N. Sridhar, "Abstractions for Safe Concurrent Programming in Networked Embedded Systems," *Proc. Fourth Int'l Conf. Embedded Networked Sensor Systems,* pp. 167-180, 2006.
[13] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems," *Proc. Fourth Int'l Conf. Embedded Networked Sensor Systems,* pp. 29-42, 2006.
[14] C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A Dynamic Operating System for Sensor Nodes," *Proc. Third Int'l Conf. Mobile Systems, Applications, and Services,* pp. 163-176, 2005.
[15] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki—A Lightweight and Flexible Operating System for Tiny Networked Sensors," *Proc. 29th Ann. IEEE Int'l Conf. Local Computer Networks,* pp. 455-462, 2004.
[16] S. Bhatti et al., "MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms," *ACM/Kluwer Mobile Networks and Applications,* vol. 10, no. 4, pp. 563-579, 2005.
[17] A. Eswaran, A. Rowe, and R. Rajkumar, "Nano-RK: An Energy-Aware Resource-Centric rtos for Sensor Networks," *Proc. 26th IEEE Int'l Real-Time Systems Symp.,* pp. 256-265, 2005.
[18] S. Yi, H. Min, S. Lee, Y. Kim, and I. Jeong, "SESAME: Space Efficient Stack Allocation Mechanism for Multithreaded Sensor Operating Systems," *Proc. 22nd Symp. Applied Computing,* pp. 1201-1202, 2007.
[19] S. Yi, S. Lee, Y. Cho, and J. Hong, "SESAME-P: Memory Pool-Based Dynamic Stack Management for Sensor Operating Systems," *Proc. Third Int'l Conf. Distributed Computing in Sensor Systems,* pp. 544-549, 2008.
[20] Q. Cao et al., "The LiteOS Operating System: Towards Unix-like Abstractions for Wireless Sensor Networks," *Proc. Int'l Conf. Information Processing in Sensor Networks,* pp. 233-244, 2008.
[21] L. Gu and J. Stankovic, "t-kernel: Providing Reliable os Support to Wireless Sensor Networks," *Proc. Fourth Int'l Conf. Embedded Networked Sensor Systems,* 2006.
[22] C. Duffy, U. Roedig, J. Herbert, and C.J. Sreenan, "Adding Preemption to TinyOS," *Proc. Fourth Workshop Embedded Networked Sensors,* pp. 88-92, 2007.
[23] N. Cooprider et al., "Efficient Memory Safety for TinyOS," *Proc. Fifth Int'l Conf. Embedded Networked Sensor Systems,* 2007.
[24] K. Klues et al., "TOSThreads: Thread-Safe and Non-Invasive Preemption in Tinyos," *Proc. Seventh Int'l Conf. Embedded Networked Sensor Systems,* pp. 127-140, 2009.
[25] R. Kumar, E. Kohler, and M. Srivastava, "Harbor: Software-Based Memory Protection for Sensor Nodes," *Proc. Sixth Int'l Conf. Information Processing in Sensor Networks,* pp. 340-349, 2007.
[26] H. Cha et al., "RETOS: Resilient, Expandable, and Threaded Operating System for Wireless Sensor Networks," *Proc. Sixth Int'l Conf. Information Processing in Sensor Networks,* pp. 148-157, 2007.
[27] R. Behren, J. Condit, F. Zhou, G. Necula, and E. Brewer, "Capriccio: Scalable Threads for Internet Services," *SIGOPS Operating Systems Rev.,* vol. 37, no. 5, pp. 268-281, 2003.

[28] P. Levis and D. Culler, "Maté: A Virtual Machine for Tiny Networked Sensors," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* 2002.

[29] R. Barr, J. Bicket, D. Dantas, B. Du, and T. Kim et al., "On the Need for System-Level Support for Ad Hoc and Sensor Networks," *Operating Systems Rev.,* vol. 36, no. 2, pp. 1-5, 2002.

[30] Crossbow Technology, Inc., MICA2 Data Sheet.

[31] J.W. Hui and D. Culler, "The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale," *Proc. Second Int'l Conf. Embedded Networked Sensor Systems,* pp. 81-94, 2004.

[32] P. Levis et al., "Trickle: A Self-Regulating Algorithm for Code Propogation and Maintenance in Wireless Sensor Network," *Proc. USENIX/ACM Symp. Networked Systems Design and Implementation,* 2004.

[33] S. González-Valenzuela, M. Chen, H. Cao, and V.C.M. Leung, "Programmable Re-Tasking of Wireless Sensor Networks Using WISEMAN," *Proc. Int'l Conf. Ad Hoc Networks,* 2010.

[34] SenSmart Project, http://www.sensmart.org, 2012.

[35] Y. Wu, S. Fahmy, and N.B. Shroff, "Optimal Sleep/wake Scheduling for Time-Synchronized Sensor Networks with QoS Guarantees," *IEEE/ACM Trans. Networking,* vol. 17, no. 5, pp. 1508-1521, Oct. 2009.

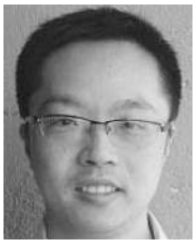
[36] S. Nikoletseas, "On the Energy Balance Problem in Distributed Sensor Networks," *Computer Science Rev.,* vol. 4, no. 2, pp. 65-79, 2010.

[37] B.L. Titzer, D.K. Lee, and J. Palsberg, "Avrora: Scalable Sensor Network Simulation with Precise Timing," *Proc. Fourth Int'l Symp. Information Processing in Sensor Networks,* p. 67, 2005.

[38] P. Levis, D. Gay, and D. Culler, "Active Sensor Networks," *Proc. Second USENIX/ACM Symp. Network Systems Design and Implementation,* 2005.

[39] S. Nath, P.B. Gibbons, S. Seshan, and Z. Anderson, "Synopsis Diffusion for Robust Aggregation in Sensor Networks," *ACM Trans. Sensor Networks,* vol. 4, no. 2, pp. 1-40, 2008.

Rui Chu received the BS and PhD degrees in computer science from the National University of Defense Technology, China, in 2001 and 2008, respectively. He was a visiting student at the Hong Kong University of Science and Technology in 2007. He is currently an assistant professor in the National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, China. His research interests include cloud computing, wireless sensor networks, virtualization and operating systems.



Lin Gu received the BS degree from Fudan University, China, in 1996, the MS degree from Peking University, China, in 2001, and the PhD degree from the University of Virginia in 2006. He is currently working as an assistant professor at the Hong Kong University of Science and Technology. His current research interests include large-scale distributed systems and cloud computing, operating systems, wireless sensor networks, and energy-efficient computing. He is a member of the IEEE.

Yunhao Liu received the BS degree from Tsinghua University, Beijing, China, in 1995, and the MS and PhD degrees from Michigan State University, in 2003 and 2004, respectively. Being a member of Tsinghua National Lab for Information Science and Technology, he holds Tsinghua EMC chair professorship. He is the director of Key Laboratory for Information System Security, Ministry of Education, and professor at School of Software, Tsinghua University. He is also a faculty member at the Department of Computer Science and Engineering, Hong Kong University of Science and Technology. His research interests include sensor networks, peer-to-peer computing, and pervasive computing. He is a senior member of the IEEE.

Mo Li received the BS degree from the Department of Computer Science and Technology, Tsinghua University, China, in 2004, and the PhD degree from the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, in 2010. He is currently an assistant professor in the School of Computer Engineering, Nanyang Technological University. His research interests include wireless sensor networking, pervasive computing and peer-to-peer computing. He is a member of the IEEE.

Xicheng Lu received the BS degree in computer science from Harbin Military Engineering Institute, China, in 1970. He was a visiting scholar at the University of Massachusetts between 1982 and 1984. He is currently a professor in the National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, China. His research interests include distributed computing, computer networks, and parallel computing. He is an academician of the Chinese Academy of Engineering. He is a member of the IEEE.