Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2012, Article ID 962151, 18 pages
doi:10.1155/2012/962151

Research Article

A Robust and Space-Efficient Stack Management Method for Wireless Sensor Network OS with Scarce Hardware Resources

Seokhwan Kim, Seungku Kim, and Doo-Seop Eom

Department of Electrical Engineering, Korea University, Seoul 136-713, Republic of Korea

Correspondence should be addressed to Doo-Seop Eom, eomds@korea.ac.kr

Received 24 April 2012; Revised 5 September 2012; Accepted 19 September 2012

Academic Editor: Donggang Liu

Due to such requirements as low power consumption and low cost, sensor nodes commonly do not include advanced H/W features. The absence of the features such as the memory management unit enforces several tasks to share a memory address domain on a small data memory space (1~16 KB). It exposes each task to the stack overflow causing the corruption of other memory areas. In this paper, we propose a robust and efficient stack memory management method (RESM) that dynamically assigns and releases a preestimated amount of stack memory to each function call at runtime. RESM maintains the stack memory usage with the similar amount of the stack usage that the system actually requires, and the stack memory area of each task is individually protected from corruption by the stack overflow. RESM can also anticipate a saturated condition in data memory at runtime. When the memory state is unsafe from the saturated condition, it conditionally allows function calls to operate tasks without any memory fault by using the proposed function call deferring algorithm. From the analytical and experiment results, it is proven that the performance of RESM outperforms the other mechanisms and RESM can provide more robust stack operation environment.

1. Introduction

Recently, the development of the semiconductor has enabled cheap, low power, and small-sized sensor nodes with the computing and communication capabilities [1]. Accordingly, wireless sensor networks (WSNs) that consist of a large number of sensor nodes have attracted considerable attention in both academy and industry because of their great potential for realizing ubiquitous environment. Despite the simplicity of the sensor node hardware, there has been an increasing demand for diverse WSN applications, such as military surveillance, habitat monitoring, U-health care, and infrastructure protection [2–6]. Accordingly, the role of the operating system (OS) that bridges the gap between H/W simplicity and application complexity becomes important [7]. OSs for a WSN (sensor OSs) can be designed using two scheduling policy approaches: event-driven task scheduling and multithreaded task scheduling [8–15]. Although the event-driven task scheduling approach, which provides run-to-completion semantics for tasks, is commonly adopted due to its simplicity of implementation and suitability for the sensor nodes with scarce H/W resource, sensor OSs based on multithreaded task scheduling have been steadily developed due to such advantages as user-friendly development environments, real-time support, and robustness which are obtained from the concurrent operation. These advantages enable multithreaded task scheduling to be more attractive for the applications with a time requirement and easy to develop for a large system in a cooperative manner [7, 13].

As many WSN applications require unattended operation for a long lifetime, one of the most important roles of the sensor OS is to ensure the reliable execution of applications. However, it has been a challenging issue to provide reliable services on a sensor node, which typically possesses scarce available H/W resources due to such requirements as low power consumption and low cost. These requirements enforce the sensor node to consist of simple and cheap H/W elements, a microcontroller unit (MCU) and RF module, providing only basic functions. MCUs that are commonly used on sensor nodes, MSP430 series [16] and ATMEGA series [17], do not include advanced H/W features, such as privileged mode and a memory management unit (MMU), to provide a reliable environment for S/W execution [13, 18, 19]. The absence of these features can easily expose

a system to control hazards and data hazards, respectively. The problems caused by this absence can be especially significant for multithreaded task scheduling. The absence causes several tasks to share a memory address space, while each task requires an exclusive stack memory space even on a small data memory space of approximately 1∼16 KB. A shared small memory space between tasks increases the probability of the stack overflow. For example, the overflow of a stack area can cause corruption of the memory areas belonging to other applications and the kernel. The corrupted areas may include dynamic, local and global variables, return addresses, and even critical MCU registers, which can cause a malfunction of the entire system.

As shown in Figure 1, the data memory is vulnerable to data corruption by buggy programs or stack overflow. Whereas logical faults in buggy programs can be detected and debugged by developers before their execution, it would demand enormous time and efforts to detect or anticipate stack overflows because all instructions related to the stack operation are internally generated by the compiler. The usage of stack memory also fluctuates according to the progress of program execution and heavily depends on a developer's coding style. Moreover, poor debugging environments for the WSN application development make it more difficult to debug or anticipate stack operation.

There are some researches for preventing the stack overflow in WSN [13, 18–23]. Without the hardware features, most methods combine compile-time instrumentation and run-time checking, that is, they analyze the program and insert hooks at compile-time, then execute these hooks at run-time to ensure system reliability. In a sensor OS based on multithreaded task scheduling, each task needs its own stack memory space in resource-constrained sensor nodes with small memory footprints. Accordingly, precise management of stack memory is required to prevent the increasing possibility for confliction between stack areas. Most previous researches focus on preventing the overflow by estimating the maximum amount of stack per each function call, each task, or a system. However, they do not detect a saturated condition in data memory at runtime. The saturated condition can occur due to the increase of tasks and indirect or direct recursive calls. In the saturated condition, an additional stack use causes the stack overflow. Accordingly, the runtime detection of the saturated condition should be considered to completely prevent the stack overflow.

In this paper, we propose a robust and space-efficient stack memory management method (RESM) that can individually protect the stack memory area of each task from corruption and can dynamically maintain the allocated stack memory space to the amount actually required by the program at runtime. The basic concept of RESM is simple. The code analyzer scans C and assembly codes to estimate the stack box size for each function. A stack box (SB) is defined as stack space, in which a function can operate its stack without the stack overflow. When three tasks simultaneously call an identical function, *f1*(), in multithreaded OS, three sets of memory blocks are assigned to three SB_{f1}s, respectively. After estimating the SB size for each function, RESM inserts hooks, called the SB manager, at all function entries and exits. At inserted points, the SB manager intercepts the program flow to dynamically assign memory blocks to SBs at function calls and to release the blocks after the functions return. Through the dynamic SB allocation, the total stack memory usage similarly maintains in comparison to the stack size that currently called functions require. Since the SB size includes all factors causing any push operation of the stack pointer (SP), the system can avoid the stack overflow in the stack space of a function call. In WSNs, the memory fluctuation tends to become rapidly high due to bursts of target events. At the occurrence of target events, the corresponding tasks would fully use their stack memory as calling a series of functions, process the event during a few milliseconds, and then release the memory by returning the functions. Note that the duration in which a task fully uses its stack is typically less than a few milliseconds. A saturated condition can easily occur as tasks rapidly increase their stack due to bursts of target events. The memory saturation can cause deadlock: although there is no available memory, all tasks request additional stack blocks at the entries of function calls, and then the system is halt. RESM includes an algorithm, called as the function call deferring algorithm (FDA), to avoid the saturation. RESM divides the memory state into two distinct phases according to the occupancy rate in data memory. The first is the stable state in which function calls can be freely allocated SBs regardless of memory saturation. The other is the metastable state in which function calls are conditionally allowed to avoid the memory saturation. As checking the size of empty memory blocks and maintaining the rate of stack increase (RSI), RESM anticipates the metastable state. In the metastable state, RESM determines whether the function call causes memory saturation at every function call, based on RSI and stack memory occupancy rate. If a new function call is unsafe from the saturation, the function call is denied. If not, the function call is conditionally allowed according to RSI. In Section 3, FDA is explained in detail. Practical implementation of RESM on sensor nodes demands several overheads, that is, additional computing time to search for empty memory blocks for SBs of function calls and an additional memory space to maintain and store the occupancy information of memory blocks. On sensor nodes with resource constraints, the efficient use of resources is one of the most important design factors. Thus, we specifically describe RESM, focusing on how to implement RESM on a real sensor node.

We organize the rest of this paper as follows. In Section 2, we describe the previous works related to memory protection for sensor OSs. In Section 3, we explain the code analyzer, the memory structure, and the SB manager for RESM in detail. Then, in Section 4, we evaluate the performance and the overhead of RESM using numerical results and simulation results. Finally, conclusions are presented in Section 5.

2. Related Work

There are various researches to prevent the stack overflow on sensor nodes. Some researches such as [13, 18, 19] prevent the memory access violation by examining the instructions

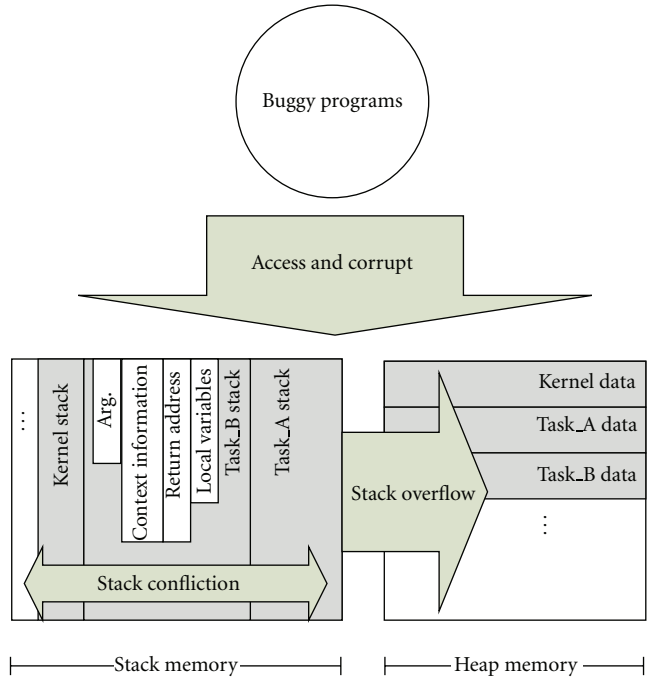

Figure 1: Example of memory corruption.

related to the memory access at runtime. Cha et al. proposed RETOS [13], a multithreaded task scheduling system that provides dual-mode operation to ensure kernel data integrity. Destination fields and source fields for all instructions related to memory access are examined to prevent applications from reading the kernel or other applications' data. RETOS allocates each task with a stack memory space, which is statically estimated using a control flow graph and a depth-first search. However, it overestimates the stack memory and does not consider stack memory for ISRs. Gu and Stankovic presented t-kernel [18] to modify application code at load-time to ensure OS responsiveness. Kumar et al. proposed Harbor [19] for per-domain memory protection in SOS and event-driven task scheduling. Harbor inspects all instructions related to memory access to each domain to verify the validity of the access, using the memory map. Harbor also assigns a part of data memory to a safe stack, in which return addresses for all function calls are stored. Because the memory map contains ownership information (a domain identity) for every block of memory, the memory space for the memory map depends on both the block size and the number of domains. Assuming a 4 KB data memory, a block size of 8 bytes, and 8 domains, 256 B should be allocated to the memory map, 6.25% of total data memory. In [13, 18, 19], the large memory space is required for maintaining the information for all memory sections, and the large computing time is consumed for examining the instructions related to the memory access at runtime.

Some researches such as [20, 21] use the fixed stack memory allocation. They estimate the maximum stack memory usage needed to execute a system, considering all tasks and interrupt service routines, and they statically allocate the estimated stack memory space. Because the stack memory usage varies according to the progress of a program execution, allocating a fixed stack memory space produces a relatively large internal fragment. In addition, the stack estimation using a call graph tends to overestimate the stack memory required. In [20], although authors reduce the stack usage based on analysis of the abstract interpretation, a considerable time is required for the analysis. In sensor nodes with scarce resource, the inefficient memory use and the time overhead degrade such performance as the decrease of task capacity and the response time.

Heo et al. proposed the shared stack management for the cooperative threads. In [22], all the cooperative threads share a single stack. The only the stack of the currently running thread occupies the shared stack at a time. When a thread suspends at the preemption points, it allocates a buffer into a head and it then copies its thread context to the buffer. The thread that will next run copies its stack to the shared stack, and it then resumes its execution. Thus, one context switching requires additional computing time for stack switching and memory compaction. In [22], the authors try to reduce the number of compactions that remove external fragmentation caused by thread switching. The work in [22] determines whether one context switching involves the compaction, based on the overflow probability. The probability is calculated based on the number of push operations at the assembly code level. However, in a task for sensor applications, the depth of a stack depends on the flow control and the characteristics of the event triggered. Accordingly, the overflow probability based on counting push operations at the assembly code level cannot reflect the current stack movement. In addition, although the cooperative threads can

decrease the number of context switching compared to that of the general multithreaded system, the frequency of context switching can greatly increase as intended interrupts or events occur more frequently. In this situation, the response time of a system can be greatly degraded as the larger portion of computing time is wasted for swapping the stack due to the frequent context switching.

Yi et al. proposed the dynamic stack allocation methods, SESAME [23] and OTL [24]. The works of [23, 24] estimate the stack amount for each function call at the assembly code level. At runtime, they dynamically allocate the stack memory block at each function call and release the memory block after the function returns. To allocate and release memory blocks, they use "malloc()" and "free()" APIs in the library. The works of [23, 24] allocate a stack memory block as much as a function call requires. The basic concept for SESAME and OTL is similar to RESM. However, several problems have not been solved. First, they do not consider a saturated condition of data memory. In the resource constraint sensor nodes, a saturated condition can easily occur as the number of tasks rapidly increases due to bursts of targeted events. Although there is no available memory space, tasks in [23, 24] continuously request additional stack memory at the entries of function calls in the saturated condition, and then system is halted. Since [23, 24] do not propose any method to detect and handle the saturated conditions, the only way to resolve deadlock problem would be system reset. Second, they propose only a concept for stack management per a function call. To implement this concept in the sensor node, several consideration points have not been solved. For example, they do not consider an additional stack space caused by context switching although the multithreaded OS is targeted. Typically, the information of some MCU registers is pushed into the current stack to save the current context at a task switching or at a jump to ISR. Accordingly, an estimated stack space in [23, 24] can easily overflow when an interrupt or a task switching occur during the execution of a subroutine. This is a critical problem in the multithreaded OS. Also, they do not suggest how to allocate, release, and manage memory blocks. It seems to use such "malloc()" and "free()" as the library APIs in the compiler or the OS. The use of the library makes the performance, for example, the time to search empty memory blocks, and the space overhead to maintain the blocks, of [23, 24] nondeterministic since it depends on the used library.

3. Robust and Space-Efficient Stack Memory Management Method (RESM)

3.1. Stack Box. We define a stack box (SB) as a stack memory space needed to execute a function or an interrupt service routine (ISR) without any stack overflow at runtime. Figure 2 shows the example of SBs for two functions and an ISR. The actual memory allocation for a SB occurs at a call to either a function or an ISR. The multiple calls of a function result in the residence of the multiple SBs corresponding to the function in data memory. An SB is allocated when a function or an ISR is called and is released when the function or the ISR returns in data memory. An SB does not include additional SBs for other function calls occurred during the execution of a function or an ISR. Therefore, an SB for a function or an ISR can be estimated as the sum of the stack depth caused by local variables, arguments, a return address, and context information. The context information means a set of information that must be saved to store the context at a point and to continue the execution at that point at an arbitrary future time. The content of context information depends on MCU manufacturers. It commonly includes some MCU registers, for example, stack pointer (SP), program counter (PC), and some general-purpose registers. When the program flow deviates from a function due to an interrupt event or task switching, the context information is pushed to save the current context into the SB of the function. On the contrary, it is popped to restore the previous context from the SB when the function retrieves the program flow again. Considering that the deviation of the program flow occurs only once within a function or an ISR at a time, the space for one context information is included in an SB. An SB for a function x, SB_x, can be determined as

$$\mathrm{SB}_x = \sum \mathrm{Local} + \sum \mathrm{Arg} + \mathrm{CI} + \mathrm{RA}. \quad (1)$$

In (1), Local, Arg, CI, and RA denote the size of local variables, arguments, the context information, and a return address for function x, respectively. Noting that an ISR does not include any argument, an SB for an ISR x, SB_x, can be estimated as

$$\mathrm{SB}_x = \sum \mathrm{Local} + \mathrm{CI} + \mathrm{RA}. \quad (2)$$

3.2. Data Memory Structure. In RESM, data memory is divided into equal-sized blocks, and the size of a block follows the form of the nth power of 2 in bytes. A block is the unit of memory allocation for SBs. In RESM, a part of data memory is exclusively assigned to a memory map that contains per-block occupancy information for an entire data memory space. The memory map is formed into a bit string in which each bit represents the availability of a block as 1 or 0; a bit set to 0 means that the corresponding block is empty and able to be allocated for any SB; a bit set to 1 indicates that the corresponding block has been already allocated for another SB. The most significant bit (MSB) of the memory map corresponds a block located at the highest address in data memory. The bits after MSB are sequentially mapped to blocks after the block of the highest address. The value of n is closely related to the memory overhead. Setting n to a larger value can produce more internal fragments in SBs, but the size of the memory map that maintains the occupancy information of blocks decreases. On the contrary, setting n to a small value can decrease internal fragments in SBs, but more memory space is required for storing the memory map. The space for the memory map, S_{mm}, and the internal

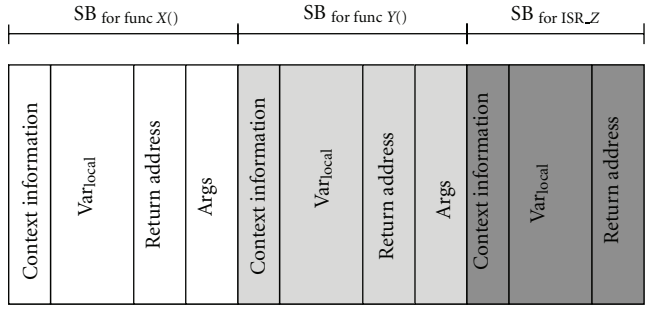

Figure 2: Example of stack boxes.

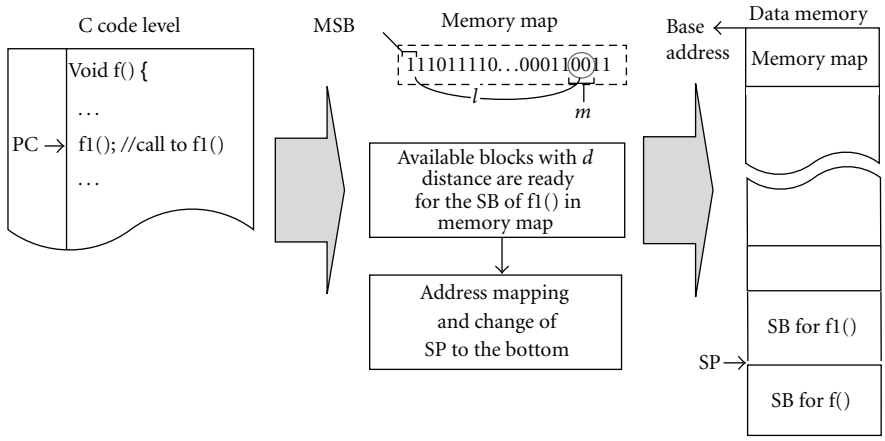

Figure 3: Example of address mapping.

memory fragmentation in SBs of ISRs and functions, S_f, for a given are calculated as

$$S_{\text{mm}} = \frac{\text{data memory size}}{2^{n+3}} \text{ (Byte)},$$
$$S_f = \sum_{\forall x} \left(2^n \times \left\lceil \frac{\text{SB}_x}{2^n} \right\rceil - \text{SB}_x \right) \text{(Byte)}, \tag{3}$$

respectively. Thus, n that minimizes the sum of S_{mm} and S_f is chosen in RESM. Here, x includes active functions and user-defined ISRs. When the size of a block is configured as 2^5 in 4 KB of data memory, 16 B are required for the memory map. It occupies just 0.39% in data memory. Assuming that m of contiguous bits are ready to accommodate an SB at a new function call and the m bits are located l bits apart from the MSB in the memory map, the physical address of the memory space for the SB is calculated by using (4) as shown in Figure 3. Then, the subroutine of the function operates its own stack between the bottom and the top addresses of the SB. Consider

$$\text{Top}(l, n) = \text{Base}_{\text{address}} + l \cdot 2^n,$$
$$\text{Bottom}(l, n, m) = \text{Top}(l, n) + m \cdot 2^n - 1. \tag{4}$$

3.3. Code Analyzer. For RESM, all source codes need to be analyzed and modified before compiling.

We developed a kind of preprocessor, called the code analyzer, which estimates the sizes of SBs for all functions and inserts the SB manager into intended points at C code level. The code analyzer analyzes both C code and assembly code translated by the compiler with the optimization option, respectively. Some user-defined functions in C code can be removed through code optimization during compiling when the compiler decides that the functions are meaningless or require no stack memory. Thus, the optimized assembly code is firstly analyzed to identify active functions, which are not removed from the code optimization and require a stack memory space at runtime. Then, the analysis of C code is conducted to measure the size of the user-made types, for example, structures, and to estimate SBs for all active functions and ISRs. After that, the code analyzer scans C source code to find points, into which the SB manager is inserted. After cross-compiling the source files containing the SB manager and downloading the object file on sensor nodes, the SB manager hooks the program flow at the inserted points and transparently performs the intended tasks such as allocating and releasing SBs for RESM. The SB estimation at C code level can cause overestimating the size of an SB because some local variables and arguments can be replaced with general-purpose registers through compiling with optimization option and, thus, does not consume stack memory at runtime. However, the stack movement can be examined more exactly at C Code level than at assembly

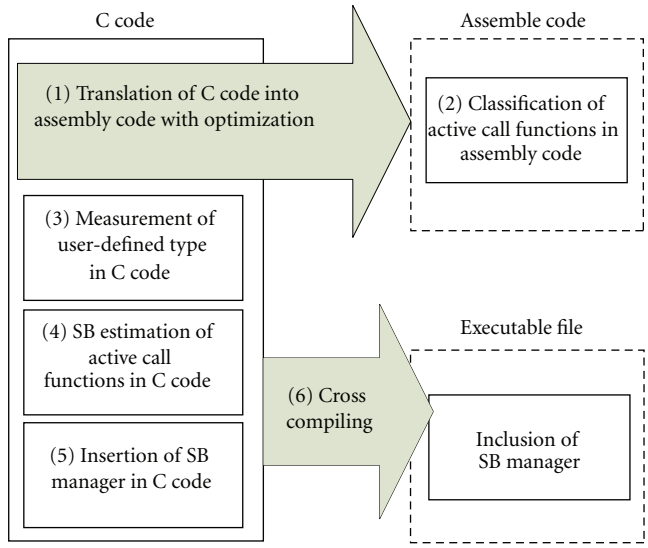

Figure 4: The procedure of code analysis.

code level. All instructions related to stack operation, for example, PUSH, POP, and arithmetic operation for SP, are automatically created by the compiler, and the operands for the instructions can be used in several addressing modes such as the immediate, direct, or indirect addressing mode. Except for in immediate address mode, it is hardly possible to anticipate the movement of SP from the analysis of assembly code. Moreover, the module insertion at C code level can be more portable due to independence on compilers and MCUs. Figure 4 shows the procedure of the code analyzer for an SB manager to be invoked at runtime. The code analyzer also modifies interrupt service routines (ISRs). The code analyzer switches all user-registered ISRs to functions and it redefines the ISRs. In a redefined ISR, code for the SB manager is inserted at the entrance and the exit. Between the entrance and the exit, a code for a call to the original ISR is inserted. Once an interrupt occurs at runtime, PC is diverted to the corresponding ISR, in which the SB manager is executed prior to an actual handler for the interrupt. After the execution of the actual handler is completed, the SB manager is executed again.

3.4. SB Manager. The SB manager performs a series of procedures at inserted points at runtime. It assigns and releases memory blocks to/from an SB for a function or an ISR and maintains the information for the use of the blocks in the memory map. Since SBs of a caller and a callee can be located on separate areas in data memory, the SB manager provides the seamless movement of SP between the SBs.

3.4.1. SB Manager in Functions. Generally, an object file is created through compiling and linking phases, during which some code, referred to as compiler generated code (CGCode), is automatically inserted into several points of source code by the compiler. At runtime, CGCode conducts several tasks to provide a suitable context for the execution of each subroutine. When a function is called, CGCode commonly pushes SP to store argument variables and a return address and diverts PC to the entry of the called function. Then, CGCode pushes SP to secure a stack memory space for local variables of the function. Values of local variables are changed at the points where the local variables are declared with initialization values or are used, for example, int $x = 0$ or $x = 0$, in the function. After the subroutine is completely executed, CGCode is again executed. At the exit of the subroutine, CGCode restores PC to the return address and increases (pop) SP as much as the subroutine decreased (pushed), so as to retrieve the previous context, at which the subroutine was called. In RESM, since SBs allocated to a callee and a caller can be located on separated area in data memory, a call to a callee can cause SP to move out of a caller's SB and to invade another SB or heap memory due to the PUSH operation of CGCode. Similarly, a return from a callee can result in a memory confliction due to the POP operation of CGCode. To prevent any corruption of other memory areas, the code analyzer inserts the SB manager before every function call at C code level. At runtime, the SB manager intercepts the program flow before every function is called. The SB manager searches for available memory blocks for the SB of each callee. If some memory blocks with n and l are successfully assigned to a SB for a call to a callee, it pushes the current value of SP, l, and m onto the stack from Bottom (l, m, n), by using (4), respectively, and it diverts SP to Bottom $(l, m, n) - 2 \cdot l$ and m are the locations of memory blocks allocated to the SB, and they are used to release the blocks at the exit of the function. After that, CGCode is automatically invoked and executes the push operation to secure the stack memory for the callee within the newly allocated SB. Then, the subroutine of the callee can be executed without any corruption of other memory areas. The code analyzer also inserts the SB manager at the next line of every function call at C code level. At runtime, CGCode is executed prior to the SB manager at the exit of a function.

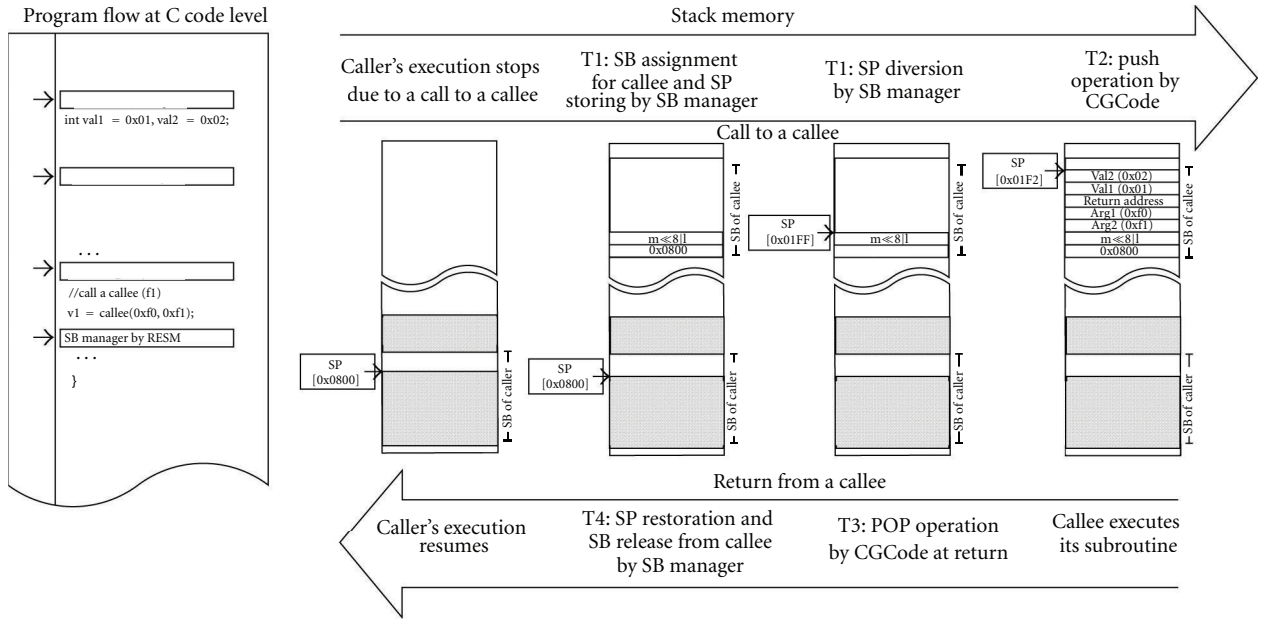

Figure 5: An example of stack operation in RESM.

CGCode automatically restores PC to the return address and increases (pops) SP. In case that the callee returns a valid value, CGCode processes the return value according to its calling convention. Since SP still indicates a point within the callee's SB, the return value can be correctly delivered to the caller independently of the calling convention of the used compiler. After the execution of the CGCode, SP indicates the Bottom $(l, m, n) - 1$, at which the SB manager stored m and l for the SB of the callee. Using m and d popped, the SB manager releases the callee's SB by setting the bits corresponding to the blocks assigned to the callee's SB to 0 in the memory map. Then, it restores SP to the previous point by simply popping the stored value at Bottom (l, m, n) and diverting SP to the stored value. After that, the subroutine of the caller continues from the point where the callee was called. Accordingly, the SB manager can connect the program flow between SBs separated on data memory without any corruption of other memory areas, independently of the used compiler. Figure 5 depletes the movement of stack memory at a call to a callee in RESM.

Prior to allocating memory blocks to an SB at a function call, the SB manager searches for a set of contiguous empty blocks that can accommodate the SB. When the SB requires m blocks, the SB manager creates a bit string consisting of m bits, $bs(m) = \sum_{i=1}^{m} 2^{(i-1)}$. Here, $m = \lceil \mathrm{SB}_x/2^n \rceil$. Then, the SB manager forms another bit string consisting of m bits, map(m, l), which duplicates m bits with a distance of l bits from MSB in the memory map. l ranges from 0 to $l_{\max}$, which indicates the maximum distance from MSB in the memory map as $l_{\max} = (\text{data memory size})/2^n - 1$. Then, the SB manager starts the *and* operation between bs(m) and map$(m, l = l_{\max})$. As decreasing l from $l_{\max}$ to 0, the operation is repeated until a certain condition is satisfied. If a result of the operation shows 0 at $l = i$, the SB manager assigns memory blocks corresponding to map(m, i) to the SB and accepts the call to the function. Then, bits relevant to map(m, i) are set to 1 in the memory map. If a result shows nonzero, some among the blocks corresponding to map(m, l) have been allocated to another SB and, thus, cannot be assigned to the SB. In the case of nonzero, x is subtracted from l, and the *and* operation is repeated. x is the distance between LSB and the highest bit with 1 in a result. This fast bit shifting reduces the time spent for searching empty blocks. The operation continues until a result shows 0 or l reaches 0. If a result of 0 is not obtained until l reaches 0, the call of the function is denied because there is not enough empty memory to accommodate its SB in data memory. The call to the function is deferred until enough memory blocks are released to prevent the stack overflow.

3.4.2. SB Manager in ISRs. Once an interrupt occurs, PC jumps to the corresponding ISR and CGCode is executed to store the current context information in stack memory. In RESM, the context information is stored in the SB of a function or an ISR, which is currently executed. Then, the SB manager is invoked. It conducts a series of tasks involved in assigning memory blocks to the SB of the user-defined ISR. After the user-defined ISR is completely executed, the SB manager again intercepts the program flow to restore the value of SP and to release the SB. After the ISR returns, the context is reinstated at the point at which the interrupt event occurred by CGCode.

3.4.3. Function Call Deferring Algorithm (FDA). Firstly, we introduce some notations used in this section as shown in Table 1. At the entry or exit of a function call, the SB manager calculates MEMo, by increasing or decreasing the number of blocks allocated to the function call. At every context switching, it also calculates the variance of MEMo,

Table 1: Notations description.

Notation	Description
MEMo	Total number of allocated memory blocks
MEMt	Total number of memory blocks
O	Stack occupancy ratio
RSI	Rate of stack increase
π	Stack occupancy threshold
μ	Sample size for RSI
α	Constant ($0 < \alpha < 1$)

by subtracting the current MEMo from the previous one. Then, RSI is calculated using a moving average of the latest μ variances. μ is set to the number of tasks currently executed, that is, the number of tasks in ready state or running state. Thus, RSI indicates an average stack usage of a running task in a scheduling interval during the latest execution time. During an execution time, μ tasks are alternately executed and consequently context switching occurs at μ times. μ ranges from 10 to 40. The negative value of RSI indicates that the stack usage is decreasing. On the other hand, the positive RSI means that the stack usage is increasing. At the entry of every function call, the SB manager executes FDA as shown in Algorithm 1.

At the entry of a function call, FDA examines O and RSI to determine whether the state is stable or metastable. π is a configurable parameter. In our experiment, we set π to 0.7. If the function call requires allocating m blocks which do not cause the memory saturation, the function call is allowed. Otherwise, FDA for the metastable state is executed. In the metastable state, FDA evaluates RSI. If RSI indicates that the stack usage has decreased enough to accommodate the SB, the function call is allowed even in the metastable state (case A). Otherwise, the function call is probabilistically deferred with a probability of $\alpha \cdot$ O to relieve the increase of RSI, since the higher RSI increases the probability that memory saturation occurs (case B). If a function call is denied, the corresponding task requests the task scheduling not to waste the CPU time. α differentiates the execution priority of tasks. In our experiment, α is set to 1. Except for the estimation of the current stack movement, RSI plays an important role in FDA. Solely using case a without case B, the expected delay time that a function is successfully called is approximately $1/(1-\alpha O)\cdot$ *execution cycle*. It can produce considerable delay time per a function call. Since a task does not decrease its stack usage until its job is completely finished, the delay time increases until $\alpha \cdot$ O becomes lower than π. In WSNs, the occurrence of events is occasional and bursty. Since tasks are triggered only when target events are detected, the stack usage follows the pattern of the event occurrence, and thus the fluctuation of the stack usage becomes high. Note that the duration in which a task fully requires its stack is limited to the short term of the burst of a target event. Accordingly, we add case A to react quickly to the fluctuation of stack usage. Figure 6 shows the impact of RSI on the stack usage. Whereas the memory saturation and deadlock occur in [23, 24], RESM avoids both the saturation and the deadlock. In addition, RESM with RSI uses the stack memory more efficiently. This experiment is conducted on a real sensor node, and the experimental environment is equally set to that in Section 4.5.

4. Performance Evaluation

In Section 4, we evaluate the performance of RESM with respect to space efficiency, overhead, and reliability from both numerical results and experimental results.

4.1. Scenario. Since there is no standard application on sensor networks, we define a scenario, in which the performance of RESM is evaluated and compared with that of the conventional fixed stack allocation method. The scenario assumes that a number of tasks run, simultaneously, and each task calls a recursive function, called as a virtual function. A virtual function do not any meaningful job but consumes resources such as CPU time and stack memory like a real function. The resource that a task consumes is adjustable by configuring four parameters; SB, p, t, and max. SB indicates the size of stack space that a virtual function needs for running without any stack overflow, p is the probability that the function recursively calls itself, t is the time consumed for the execution of the subroutine, and max is the maximum call depth by which the function is allowed to recursively call itself. Note that p is not O. The call depth, k, indicates the number that a running virtual function recursively calls itself at that time. Once a virtual function is called, it permanently runs. k of a virtual function can vary between 0 and max. We classify sensor tasks into two types according to the pattern of the stack use. The first is a general type, in which there is no pattern for the stack use. The general type represents non-event-driven tasks which run once or permanently. The second is an event type, in which the stack use of tasks shows a similar pattern. The event task is triggered according to events, for example, tasks for networking, monitoring, or surveillance applications. An event occurrence causes a task to process the event by calling a series of functions, whereas few functions are called when any event does not occur. Accordingly, the stack usage increases rapidly at an event occurrence and decreases sharply after the event processing is completed. Figure 7 shows the pseudocode of a virtual function for the general and the event types. In Figure 7, SB, t, p, and max are set to 30 B, 10 ms, 0.5, and 5, respectively. Then, the stack space that the task requires can vary between 50 B and 300 B according to k. A recursive call is made only if a randomly chosen value is within p. When k reaches zero, a virtual function does not return, though a randomly chosen value is within out of p. When k approaches max, k does not increase anymore, though a randomly chosen value is within p. Adjusting parameters, a task can show the various patterns for the stack memory use.

4.2. Analytical Modeling . Let $d(t)$ be the stochastic process presenting the call depth of a virtual function for a given task. Note that each recursive call requires an identical time for its execution in a task, and discrete and integer time scales are

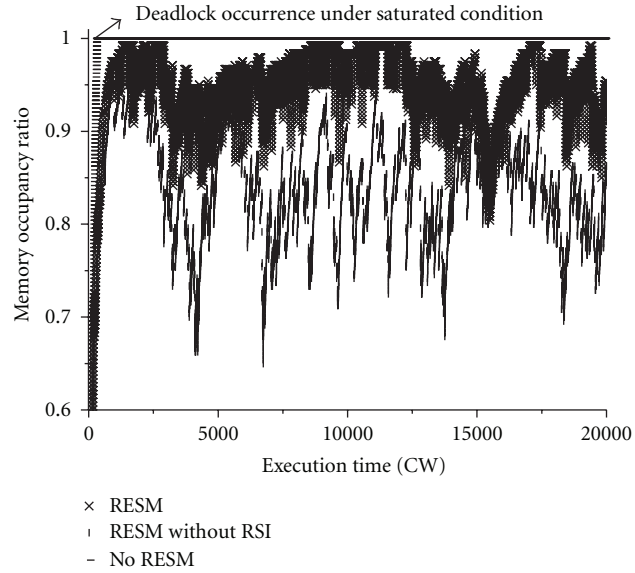

Figure 6: An example of stack operation in RESM.

(i) when a task try a function call which requires m memory blocks for SB
$O = \text{MEM}o/\text{MEM}t$
If $O < \pi$ and $(m + \mu \cdot \text{RSI} + \text{MEM}o) < \text{MEM}t$ **then** (stable state)
the function call is allowed
else (metastable state)
if $(\mu - 1) \cdot \text{RSI} < (-1) \cdot m$ **then** (**case A**)
(i) the function call is allowed
else
if $(m + (\mu - 1) \cdot \text{RSI} + \text{MEM}o) < \text{MEM}t$ **then** (**case B**)
(i) the function call is allowed with a probability of $(1 - \alpha O)$
(ii) the function call is denied with a probability of αO and deferred until the next execution cycle
else (**case C**)
(i) the function call is denied and deferred until the next execution cycle

Algorithm 1: Function call deferring Algorithm.

adopted: t corresponds to the number of calls. That is, $d(t)$ means the call depth when a virtual function is recursively called t times in a task. Note that this discrete time scale does not relate to the system time. Then, it is possible to model the one-dimensional process $d(t)$ with the discrete-time Markov chain depicted in Figure 8. Let $d_k = \lim_{(t \to \infty)} P\{d(t) = k\}$, $k \in (0, \max)$ be the stationary distribution of the chain. Now, it is simple to obtain a closed-form solution for this Markov chain. First, considering the chain regularities, d_k for the general type is

$$d_k = \begin{cases} (1-p) * (d_{0+} d_1), & k = 0, \\ p \cdot d_{k-1} + (1-p) \cdot d_{k+1}, & 1 \le k \le \max - 1, \\ p * (d_{k-1} + d_k), & k = \max. \end{cases} \quad (5)$$

Then, for each $k \in (1, \max)$, d_k is

$$d_k = \frac{p^k}{(1-p)^k} \cdot d_0, \quad 1 \le k \le \max. \quad (6)$$

Using the fact that $\sum_{k=0}^{\max} d_k = 1$, d_0 can be calculated as

$$d_0 = \frac{1}{\sum_{i=1}^{\max} p^i / (1-p)^i}. \quad (7)$$

Similarly, d_k for the event type is expressed as

$$d_k = \begin{cases} (1-p) \cdot \sum_{i=0}^{\max} d_i, & k = 0, \\ d_0 \cdot p^k, & 1 \le k \le \max - 1, \\ d_0 \cdot p^k + d_k \cdot p, & k = \max. \end{cases} \quad (8)$$

Then, d_0 and d_k are calculated as

$$d_0 = (1-p), \qquad d_k = \begin{cases} (1-p) \cdot p^k, & 0 \le k < \max, \\ p^k, & k = \max, \end{cases} \quad (9)$$

```
/*******For General Type********/
/*parameter s*/
#define SB 50
/*parameter t*/
#define T 10
/*parameter p * 100 */
#define P 50
/*parameter max*/
#define MAX 5
/*mcu tics per 1 ms*/
#define DELAY (MCU_CLK/1000) * T
#define RAND() TIMER_TIC

void v_func(void)
{
   volatile uint8 s[SB − 4];
   volatile uint32 i;
   static uint8 k = 0;
   while (1) {
      for (i = 0; i < DELAY; i++)
      {nop;}
      s[0] = RAND() % 100;
      if(s[0] < P1) {
         if(d < MAX) {
         k++;
         v_func(void);
         k − −;}
         else {continue;}
      }
      else if(!k)
      {continue;}
      else {return;}
   }
}
void * task (void * arg)
{
   v_func();
   return 0;
}
```

```
/*******For Event Type********/
/*parameter s*/
#define SB 50
/*parameter t*/
#define T 10
/*parameter p * 100 */
#define P 50
/*parameter max*/
#define MAX 5
/*mcu tics per 1 ms*/
#define DELAY (MCU_CLK/1000) * T
#define RAND() TIMER_TIC

void v_func(void)
{
   volatile uint8 s[SB − 4];
   volatile uint32 i;
   static uint8 k = 0;
   do {
      for (i = 0; i < DELAY; i++)
      {nop;}
      s[0] = RAND() % 100;
      if(s[0] < P) {
         if(k < MAX) {
            k++;
            v_func(void);
            k − −;
         }
      } else if(k) {return;}
   } while(!k ||k==MAX);
}
void * task (void * arg)
{
   v_func();
   return 0;
}
```

Figure 7: Pseudocode of virtual functions with SB = 50, $t = 10$, $p = 0.5$, max = 5 for two types.

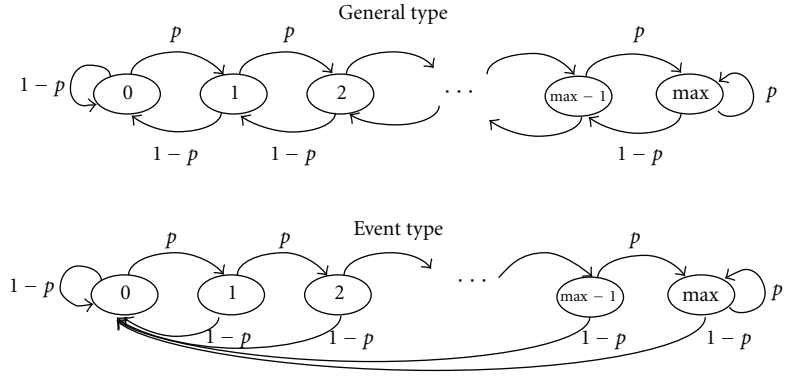

Figure 8: Markov chain model for the call depth.

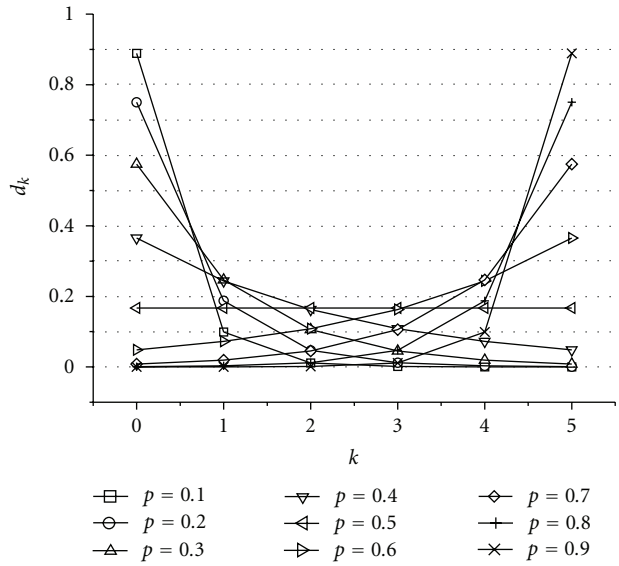

Figure 9: The distribution of d_k for various p in the general type.

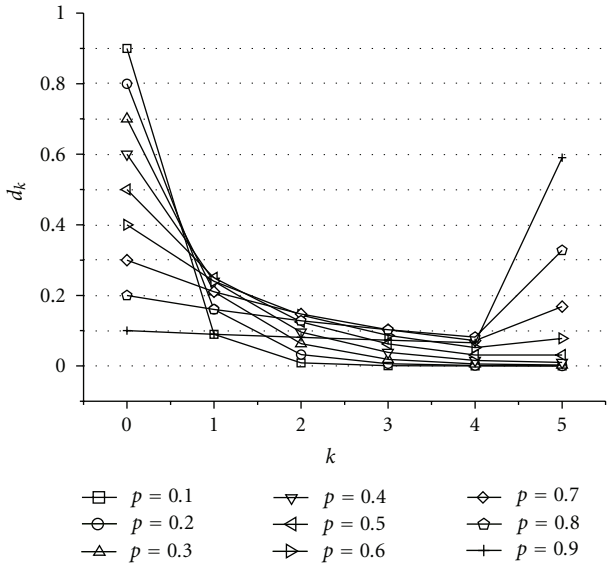

Figure 10: The distribution of d_k for various p in the event type.

respectively. Then, the expected stack memory usage of a task at a given p can be calculated as

$$E = \sum_{k=0}^{\max} d_k \cdot (k+1) \cdot \text{SB}. \tag{10}$$

4.3. Reality versus Analytical Model. Figures 9 and 10 present the impact of varying p on the distribution of d_k for a given task with max = 5 for the general and the event types, respectively. Whereas a smaller value of p results in the higher probability at the lower call depth, a larger value of p leads the higher probability at the higher call depth. From Figures 9 and 10, whereas the distributions of d_k in the general type are not biased, the distributions in the event type are weighed toward the lower call depth. That is, a task of the event type approaches infrequently the maximum call depth, compared to that of general type. Most tasks on sensor nodes are operated in an event-reaction manner. The tasks periodically monitor environments or wait for the occurrence of some events. Once an interesting phenomenon is sensed or an intended event is triggered, a relatively large quantity of work is conducted to process and respond to the event. That is, at the occurrence of an intended event, the task approaches the maximum call depth and experiences the maximum stack memory usage by executing a series of subroutines. The more frequently the intended events are triggered, the higher possibility the task has to reach the maximum call depth. Here, we assume that $d_{\max}$ roughly matches an event ratio, λ, which is the proportion of the number of intended events to the total number of events sensed. To demonstrate our assumption, we operate a real task for autocooling system on a real sensor node equipped with MSP430 5438, CC2420, LEDs, and a step motor connected with a fan. The sensor node periodically senses CPU temperature of a PC, on which various applications such as web browsers and 3D games are executed to change CPU temperature. The task has a temperature category table, which classifies the range of 30∼80 degrees Celsius into 20 categories according to the degree. Each category has a profile for the toggling rate of LEDs and the frequency of the step motor. The maximum call depth that task can reach is 5. The task periodically senses temperatures every 100 ms, and it calculates the average for the latest ten temperatures sensed. After calculating a new average by a newly sensed temperature, it scans the table to find which category includes the new average. If the average is out of the range of the current category and included in new one, it adjusts the toggling rate of LEDs and the frequency of the step motor according to the profile of the new category. When the average enters a new category with

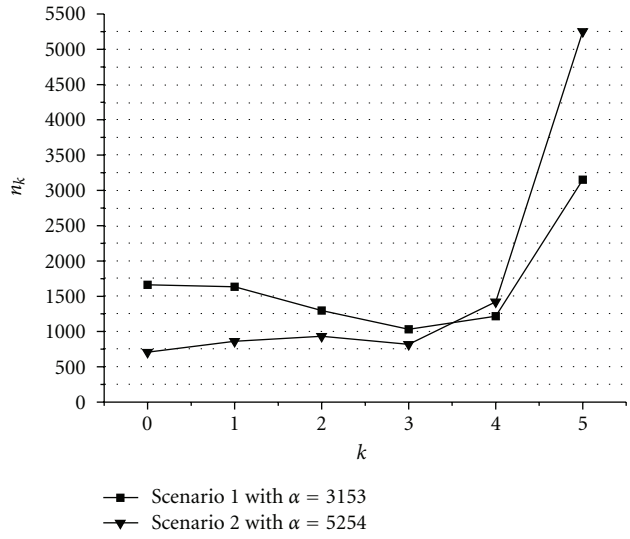

Figure 11: Experimental results for n_k.

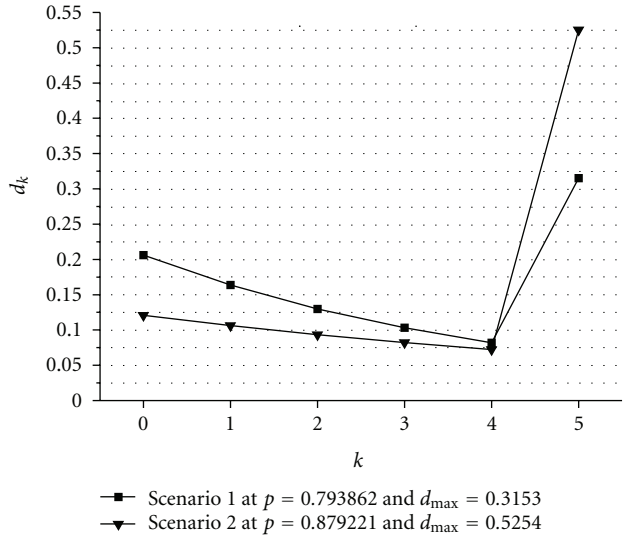

Figure 12: Analytical results for d_k at the given p and $d_{\max} = \lambda$.

high temperatures, the task adjusts the LEDs' toggle and the step motor to run at a higher frequency. When the average enters a new category with lower temperature, the task lowers the frequency. If the average enters the category with the highest temperatures, the task sends a warning message to another sensor node connected to a PC via the wireless interface, and it adjusts the step motor to run at the highest frequency. Here, an intended event is a sensed temperature that moves the calculated average to a new category. Then, the event ratio, λ, changes according to the variation of temperatures sensed. At every temperature sensed, the call depth that the task reaches is logged. Since every temperature sensed does not involve changing the category, each call depth that the task approaches at each temperature sensed differs from each other. We count the number, n_k, that the task approaches each call depth every temperature sensed during $T = 10,000$ seconds under two scenarios. In the first scenario, web browsers are executed varying the number of the application. In the second scenario, a 3D game is played. Figure 11 shows the distribution of n_k in the two scenarios. Assuming that $n_{\max}$ follows Poisson distribution with α, it can be expressed as

$$P[n_{\max} = i] = \frac{(\alpha)^i}{i!} e^{-(\alpha)}. \tag{11}$$

From (11), the event ratio, λ, can be expressed as $\lambda = \alpha/T$. After calculating p at $d_{\max} = \lambda$ by $d_{\max}(\lambda)^{-1}$, the distribution of d_k is obtained at the given p from (9). Figure 12 shows the distribution of d_k in the two scenarios. From Figures 11 and 12, the curve for n_k shows the similar pattern to that of d_k. The results show that, when p is adjusted for $d_{\max} = \lambda$, our model roughly estimates the movement of the call depth for a task of the event type.

4.4. Memory Usage. In this section, we evaluate the expected stack usage and the memory overhead in RESM and OTL [24]. Figure 13 shows the impact of p on the probability, $d_{\max}$, that the call depth reaches max in the general and the event types. We assume that $d_{\max}$ represents the event ratio, λ. Since $d_{\max}$ depends on p, we adjust p to reproduce the stack usage under the various event ratios. The result shows that $d_{\max}$ in the event type increases more slowly than that in the general type according to increasing p. That is, applications for the event type have less probability to use their full stack memory.

Table 2: Experiment environment.

	Parameter	Value
H/W	MCU	MSP430 5438
	Cross complier	MSP IAR
	Data memory	16 KB
	System clock	16 MHz
OS	Task switching time	60.53 us
	Code size	10,476 B
	Data size	334 B
	Start code stack	245 B
	Available data memory	15,360 B [0x1D4E-0x594D]
	Scheduling interval	10 ms
	TCB size	16 B
	Task queue size	$j + 2$ B
RESM	Block size	64 B
	The number of tasks, j	40
	t	10 ms
	SB	60
	max	10

The expected stack memory usage of a task in RESM, E_{RESM}, can be derived from (3) and (10) as

$$E_{\text{RESM}} = \frac{S_{\text{mm}}}{j} + \sum_{i=0}^{\max} d_i \cdot (i+1) \cdot 2^n \cdot \left\lceil \frac{\text{SB}_i}{2^n} \right\rceil. \tag{12}$$

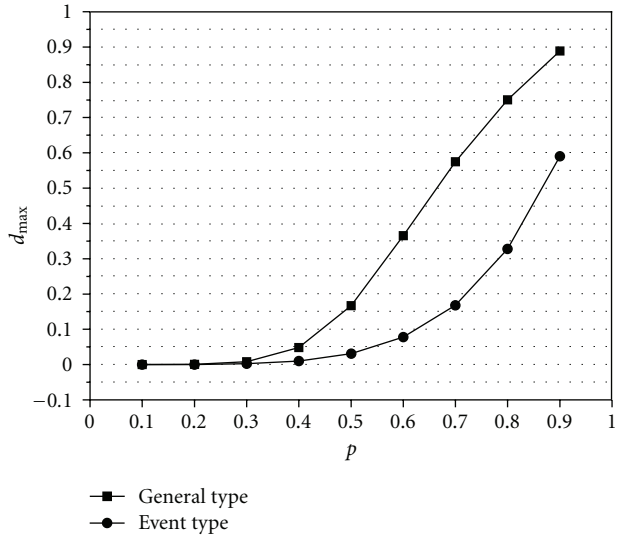

Figure 13: The probability that a task approaches its maximum stack depth.

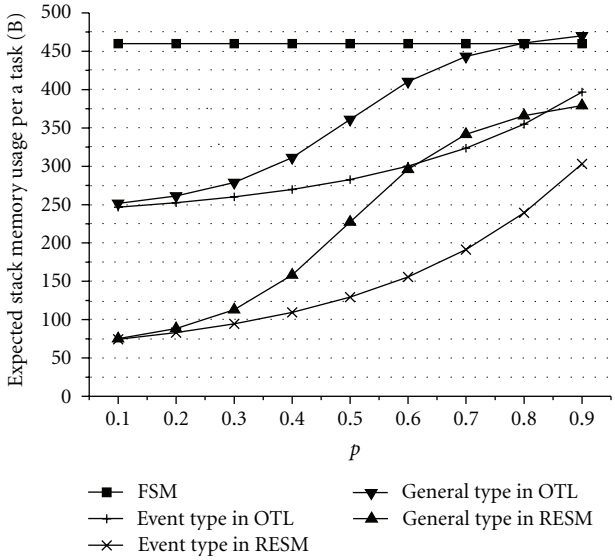

Figure 14: Expected stack memory usage per a task.

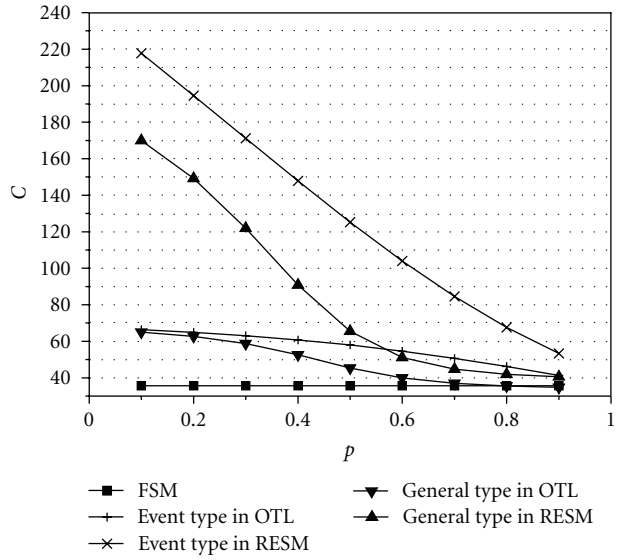

Figure 15: Expected task capacity.

j means the number of tasks. The constant portion for the memory map is offset by increasing tasks in RESM. A task in RESM does not need the stack space for ISR since the stack of ISR is accommodated in other memory blocks. Equation (13) shows the expected stack memory usage of a task in the fixed stack management (FSM). In FSM, the stack space for a task includes the maximum stack memory amount that the task uses and the maximum stack space in which all interrupts occur simultaneously [20, 21]. Consider

$$E_{FSM} = CI + \sum^{All\ isr} SB_{isr} + \sum_{i=0}^{max} S_i. \quad (13)$$

Equation (14) shows the expected stack memory usage of a task in OTL [24]. As mentioned in Section 2, OTL does not consider the space for the context information when estimating the stack space of a function. Similar to FSM, E_{OTL} includes the space for ISRs. E_{OTL} produces 6 B per a memory block for storing the block size, SP. When OTL is evaluated on a real sensor node, the system is frequently reset or halt due to stack overflow as the memory usage increases. In this section, it should be assumed that OTL does not cause overflow. Consider

$$E_{OTL} = \sum^{All\ isr} SB_{isr} + \sum_{i=0}^{max} d_i \cdot (i+1) \cdot (S_i + 6). \quad (14)$$

In this evaluation, j, SB, and max are configured as 10, 60 B, and 5, respectively. Then, n is obtained as 3 by using (3). In MSP430 5438, 20 B is required for $CI \cdot S_i$ is set to

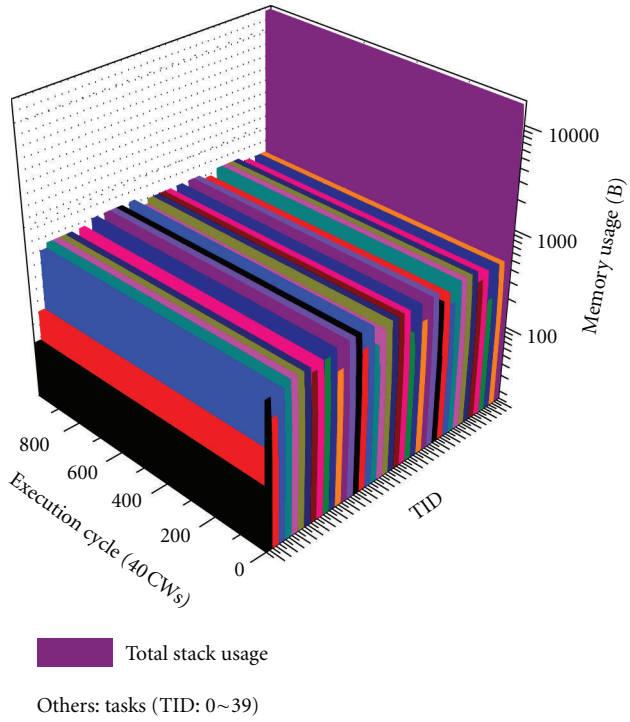

Figure 16: Stack operation under a saturated condition in OTL.

40B by subtracting CI from SB_i. Considering 10 ISRs, at least 200 B are required for stack of ISRs, $\sum^{(\text{All isr})} SB_{isr}$, when all interrupts occur simultaneously. Then, we evaluate the expected usage under the various p. Figure 14 presents the expected stack memory usage for both types in FSM, OTL, and RESM. From Figure 14, the results show that RESM outperforms the others since the stack ISR is operated on other memory blocks. As p increases, the difference between OTL and RESM deceases. This is because RESM produces intrafragments due to per-block allocation. As the event ratio becomes higher, the number of function calls increases, and then more intrafragments are produced.

Assuming that virtual functions in all tasks have the same set of parameters, the capacity, C, in which the data memory can accommodate the maximum number of tasks can be estimated as $C = \text{Memory\_Size}/E$ for OTL and FSM and $C = (\text{Memory}_{\text{Size}} - \sum^{(\text{All isr})} SB_{isr})/E_{\text{RESM}}$ for RESM. Figure 15 shows the capacity in which 16 KB of RAM can accommodate the maximum number of tasks. Similar to Figure 14, RESM outperforms the others. As p increases, three methods show the similar capacity.

4.5. Robustness. In this subsection, we evaluate the robustness of RESM under conditions of stack overflow on a real sensor node. In order to evaluate RESM on the real sensor node, RESM needs to operate on the multithreaded sensor OS such as MANTIS OS. However, most conventional OSs for WSN require the additional memory space and the additional computing time for their various subsidiary functions internally executed. Due to the additional functions, it is hard to measure the memory space or the computing time caused by RESM. To exactly evaluate the RESM, it is necessary to minimize functions in OSs. For the experiment, we developed a simple OS on a real sensor node. The OS consists of the multithreaded task scheduler and device drivers. The scheduler has one ready queue, into which user-defined tasks are sequentially inserted. Then, the scheduler periodically fetches and executes a task from the front of the ready queue at the interval of 10 ms. If a task is being executed at the end of the interval, the execution of the task is suspended and inserted into the rear of the ready queue, and, then, the newly fetched task is executed, that is, context switching. Fortunately, it is very simple to implement the context switching on such MCUs used on sensor nodes. The context switching includes pushing and popping SREG, general-purposed registers, and SP and manipulates SP and PC. Table 2 shows the parameters for the experiments.

To evaluate the robustness of RESM and OTL, we adjust the parameters to reproduce the memory saturation. 40 tasks are concurrently executed on the simple OS, and each task is assigned a unique task ID (TID) from 0 to 39. Each task executes its virtual function for the event type. The same parameters of $p = 0.89$, $s = 60$ B, $t = 10$ ms, and max $= 10$ are applied to all virtual functions for 40 tasks. p is chosen to meet a saturated condition, that is, the total expected stack usage is calculated as 16846.16 B at $p = 0.89$ from (10). Then, the call depth for all tasks is measured every context switching at runtime during 1000 execution cycles. From the measured call depth, the stack memory usage for all tasks is calculated.

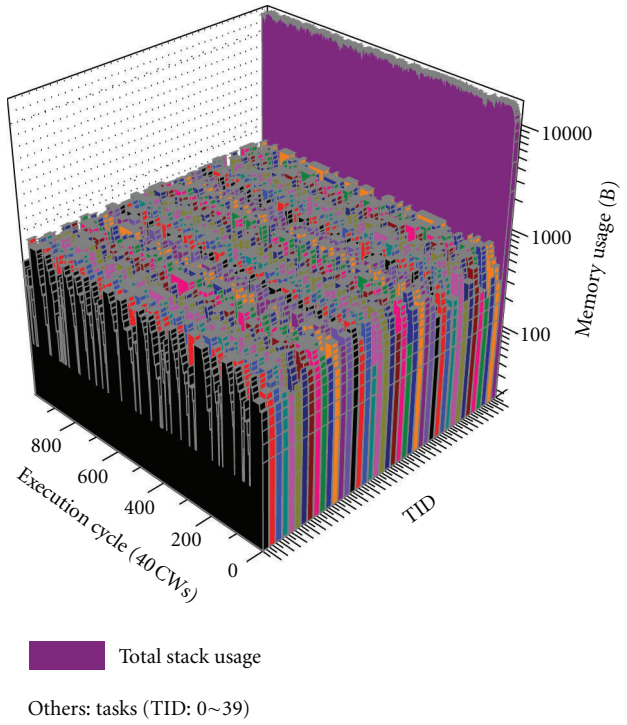

Figure 17: Stack operation under a saturated condition in RESM.

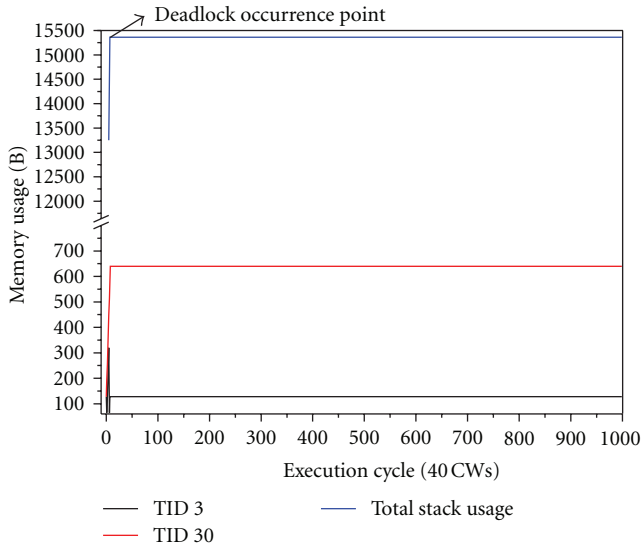

Figure 18: Deadlock problem in OTL.

Figures 16 and 17 show the stack memory usage for all tasks in OTL and RESM during the execution cycles. From Figure 16, it is shown that deadlock occurs and the system halts as soon as the stack memory is saturated in OTL. In contrast, RESM avoids the saturated condition, and all tasks run without deadlock. Since RESM can detect the metastable state and prevent the memory saturation by using FDA at runtime, the total stack memory usage does not excess 16 KB. For visibility, Figures 18 and 19 show the stack usage for the total tasks, task 3 and task 30 from Figures 16 and 17.

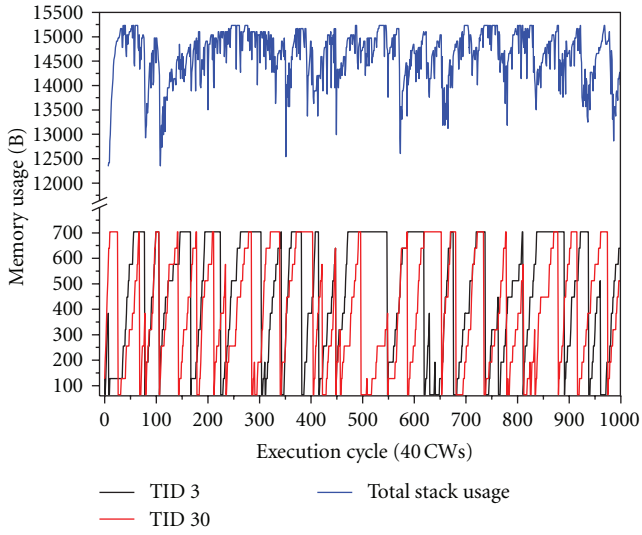

Figure 19: Deadlock avoidance in RESM.

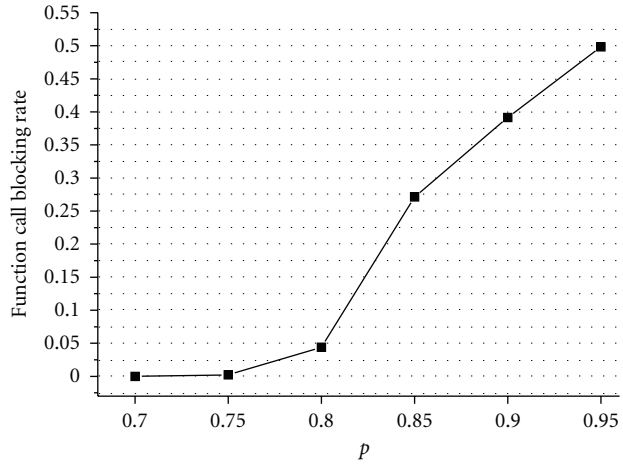

Figure 20: Function call blocking rate versus p.

From Figure 19, the stack usage of tasks increases in a discrete manner since function calls are deferred by FDA. Accordingly, RESM can continue to execute all tasks without any stack memory fault even under the saturated condition at runtime. That is, under the saturated condition, RESM accommodates as many tasks as possible. Figure 20 shows the impact of p on the call deferring probability as the event ratio increases. Except for p, the experiment parameters are identically set to those of Figure 17. As the event ratio increases, tasks process more events, and then the stack usage increases. Accordingly, the call blocking rate increases due to the lack of memory. At $p = 0.95$, the call blocking rate is approximately 0.4984. Now, we show the delay time that a task consumes for executing FDA.

Figure 21 shows the impact of p on the execution time and the delay time for a task. Numbers on bars present the delay ratio calculated as the proportion of the delay time to the execution time. The delay time and the execution time is the average on 40 tasks. The waiting time is measured as

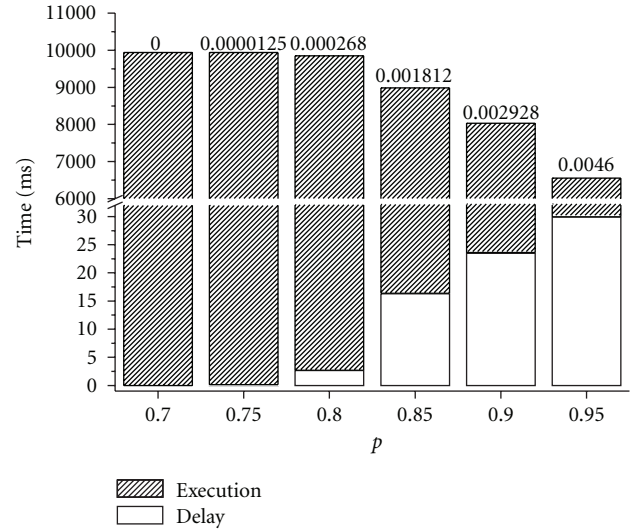

Figure 21: Execution time and delay time versus p.

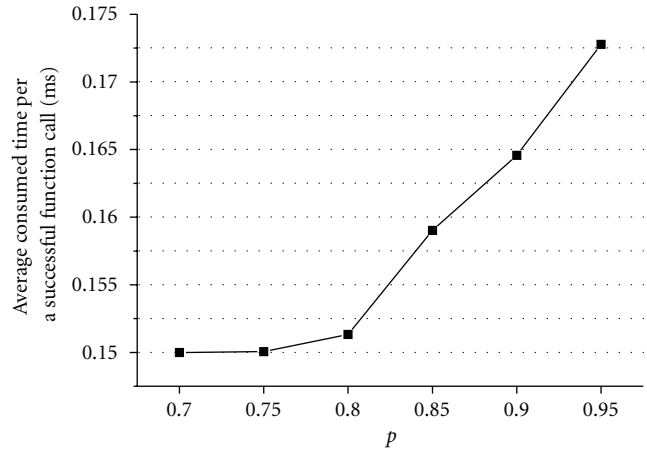

Figure 22: Average time overhead per a successful function call in RESM.

the sum of time that a task spends for the execution of FDA when a new function call is denied at the metastable state. The total execution time is measured as the sum of time during which a task preempts the MCU execution, that is, the time that a task spends in running state. Then, the execution time is obtained by subtracting the delay time from the total execution time. Note that the execution time includes the time that RESM consumes at successful function calls. Here, we are interested in time that a task spends waiting for deferred function calls in the metastable state. At $p = 95$, the delay time is 30 ms and the delay ratio shows 0.46%. Under a higher event ratio, the execution time decreases since FDA requests task scheduling when an attempt to call a new function is failed. The delay time is very low, but the time that all tasks actually conduct their jobs rapidly decreases. As the call blocking rate increases, the time that context switching consumes increases since the tasks fail to call functions and request task switching.

4.6. Time Overhead. In this subsection, we experimentally measure the time overhead caused by RESM. First, the number of attempts on function calls is measured for all tasks. Then, the delay time is measured as the sum of time that RESM consumes. After that, the average consumed time per a successful function call is calculated as the delay time divided by the number of attempts. As shown in Figure 22, the average time consumed by RESM increases under a higher p. It can be shown that the function call deferring rate is zero at $p = 0.70$ from Figure 20. Thus, at $p = 0.70$, the average consumed time per a successful function includes the time for allocating and releasing memory blocks without FDA. As p increases, the call blocking occurs, and then the average time per a successful function call increases due to the call deferring of FDA. Although the call blocking rate approximately becomes 0.5, the average time per a successful function call at $p = 0.95$ increases as much as 30 ms in comparison with that at $p = 0.7$. From these, RESM spends

about 150 μs for searching, allocating, and releasing memory blocks when there is no call blocking. When the average processing time per a function call is 10 ms, the proportion of the time wasted on RESM to the time consumed for executing a subroutine is approximately 1.5%.

5. Conclusion

In this paper, we propose a robust and space-efficient stack management method (RESM) for sensor OSs on sensor nodes with scarce H/W resources. RESM dynamically assigns memory blocks for an SB at each function call, and it releases the blocks at each function return. RESM keeps the total stack memory usage similar to the amounts that a system actually requires at runtime. To do this, we implement the code analyzer to estimate the stack memory amount for all active functions and ISRs and to modify the user code at the C code level, and the connection between stack spaces separated on memory is introduced. In addition, RESM provides a reliable execution environment to tasks. It anticipates the memory saturation by using FDA and conditionally allows function calls when the current memory state is unsafe from the memory saturation. From analytical and experimental results, it is proven that RESM provides the efficient stack usage and the robustness with the tolerable overhead.

Acknowledgment

This research was supported by the Agency for Defense Development (ADD), Republic of Korea, under the Dual Use Technology Program.

References

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," *Computer Networks*, vol. 38, no. 4, pp. 393–422, 2002.

[2] M. Li and Y. Liu, "Underground structure monitoring with wireless sensor networks," in *Proceedings of the 6th International Symposium on Information Processing in Sensor Networks (IPSN '07)*, pp. 69–78, April 2007.

[3] Z. Yang, M. Li, and Y. Liu, "Sea depth measurement with restricted floating sensors," in *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS '07)*, pp. 469–478, 2007.

[4] T. He, S. Krishnamurthy, J. A. Stankovic et al., "Energy-efficient surveillance system using wireless sensor networks," in *Proceedings of the 2nd International Conference on Mobile Systems, Applications and Services (MobiSys '04)*, pp. 270–283, 2004.

[5] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler, "An analysis of a large scale habitat monitoring application," in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys '04)*, pp. 214–226, November 2004.

[6] N. Xu, S. Rangwala, K. K. Chintalapudi et al., "A wireless sensor network for structural monitoring," in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys '04)*, pp. 13–24, November 2004.

[7] W. Dong, C. Chen, X. Liu, and J. Bu, "Providing OS support for wireless sensor networks: challenges and approaches," *IEEE Communications Surveys and Tutorials*, vol. 12, no. 4, pp. 519–530, 2010.

[8] O. S. Tiny, http://www.tinyos.net.

[9] A. Dunkels, B. Grönvall, 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 (LCN '04)*, pp. 455–462, November 2004.

[10] C. C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services, (MobiSys '05)*, pp. 163–176, June 2005.

[11] S. Bhatti, J. Carlson, H. Dai et al., "MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms," *Mobile Networks and Applications*, vol. 10, no. 4, pp. 563–579, 2005.

[12] A. Eswaran, A. Rowe, and R. Rajkumar, "Nano-RK: an energy-aware resource-centric RTOS for sensor networks," in *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS '05)*, 2005.

[13] H. Cha, S. Choi, I. Jung et al., "RETOS: resilient, expandable, and threaded operating system for wireless sensor networks," in *Proceedings of the 6th International Symposium on Information Processing in Sensor Networks (IPSN '07)*, pp. 148–157, April 2007.

[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 International Conference on Information Processing in Sensor Networks (IPSN '08)*, pp. 233–244, April 2008.

[15] 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 (SenSys '06)*, pp. 29–42, November 2006.

[16] "Texas Instruments MSP430F543xA Mixed Signal Microcontroller Datasheet," rev. M, 2010.

[17] Atmel ATmega640/1280/1281/2560/2561 Preliminary Summary, rev. N, 2011.

[18] 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 (SenSys '06)*, pp. 1–14, November 2006.

[19] R. Kumar, E. Kohler, and M. Srivastava, "Harbor: software-based memory protection for sensor nodes," in *Proceedings of the 6th International Symposium on Information Processing in Sensor Networks (IPSN '07)*, pp. 340–349, April 2007.

[20] J. Regehr, A. Reid, and K. Webb, "Eliminating stack overflow by abstract interpretation," *ACM Transactions on Embedded Computing Systems*, vol. 4, no. 4, pp. 6–28, 2005.

[21] W. P. 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 (SenSys '06)*, pp. 167–180, November 2006.

[22] J. Heo, B. Gu, Y. Cho, and J. Hong, "An efficient stack management for sensor operating systems," *Journal of Information Science and Engineering*, vol. 26, no. 3, pp. 1137–1150, 2010.

[23] S. Yi, H. Min, S. Lee, Y. Kim, and I. Jeong, "SESAME: space-efficient stack allocation mechanism for multi-threaded sensor operating systems," in *Proceedings of the ACM Symposium on Applied Computing*, pp. 1201–1202, kor, March 2007.

[24] S. Yi, S. Lee, Y. Cho, and J. Hong, "OTL: on-demand thread stack allocation scheme for real-time sensor operating systems," *Lecture Notes in Computer Science*, vol. 4490, no. 4, pp. 905–912, 2007.