An Efficient Stack Management for Sensor Operating Systems*

JUNYOUNG HEO1, BONCHEOL GU2, YOOKUN CHO2 AND JIMAN HONG3+

1Department of Computer Engineering
Hansung University
Seoul, 136-792 Korea

2School of Computer Science and Engineering
Seoul National University
Seoul, 151-742 Korea

3School of Computer Science and Engineering
Soongsil University
Seoul, 156-743 Korea

Operating systems for sensor networks must provide energy and memory-space efficient execution environments for applications because of the resource constraints of the sensor nodes. The shared-stack cooperative threads have been proposed to conserve stack memory-space and to minimize the possibility of stack overflow in the sensor operating systems. However, stack switching brings about external fragmentations in the stack space. Compaction may remove the fragmentation but the compaction overhead could degrade the performance. In this paper, we propose an efficient scheme to determine the compaction time of a shared-stack to reduce the number of compactions. For determining the time of a compaction, we evaluated the expected stack overflow time according to our stack model, which is based on the continuous time Markov chain. Our simulation results show that the number of compactions is greatly reduced and the lifetime of the sensor networks is increased with using our proposed scheme.

Keywords: operating systems, sensor networks, cooperative threads, stack overflow, memory external fragmentation, continuous time Markov chain

1. INTRODUCTION

As the number of wireless sensor networks applications is increasing, a sensor node must perform a wide range of tasks including topology control, routing, aggregation, network management, power management, security, and maintenance in a realistic environment. A sensor node is a small and autonomous computer that is equipped with a microprocessor with limited computation power, a very limited memory-space, a communication system that is usually in the form of radio frequency, and a battery. Inherently, most individual sensor nodes in wireless sensor networks should be able to perform various tasks despite the severe resource constraints such as a low computing power and a very small amount of memory. Therefore, enabling the sensor operating systems to efficiently utilize their system resources has been a very important issue.

Received January 15, 2008; revised May 16 & August 20, 2009; accepted September 26, 2009.
Communicated by Tei-Wei Kuo.
* This research was financially supported by Hansung University in the year of 2009, it was supported by NIPA under the program of Software Engineering Technologies Development and Experts Education, and it was supported by the MKE, Korea, under the ITRC support program supervised by the IITA (IITA-2008-C1090-0803-0006).
+ Corresponding author.
Some sensor networks’ operating systems have been designed to efficiently manage the restricted system resources. Especially, memory management is completely different from that of general computer systems because the sensor nodes have very small memory space. Atmega 128, which is most popularly used processor for sensor nodes, has only 4Kbytes of memory, and it has no MMU (Memory Management Unit) that is required to implement virtual memory in operating systems. In general, stack memory is assigned at a fixed size to each thread. In general, for the systems with the MMU hardware, unused stack memory virtually exists, and it does not waste physical memory through virtual memory management. However, in the sensor nodes without the MMU hardware, physical memory is assigned as stack areas and the unused stack memory tends to waste physical memory [5].

In the embedded systems or sensor nodes without the MMU hardware, estimating stack sizes is required [5]. A very large stack size can result in waste of physical memory. Memory waste is a critical problem in the small memory sensor nodes. Too small a stack size may result in a stack overflow. It is very difficult to detect stack overflows, which can cause the systems to crash. Multithreaded sensor operating systems such as MANTIS OS [2] and Nano-Qplus [3] may demonstrate the above problems.

In this paper, we extend the shared-stack cooperative threads scheme that is presented in [4]. We propose an efficient stack memory compaction scheme to improve the performance of the shared-stack cooperative threads. With using the cooperative threads, it is possible to combine the simplicity of multithreaded programming with the performance and scalability of the event-driven system at the expense of the reasonable memory usage and the context switching overhead.

Traditional threads are assigned separate memory space for their own stacks [2, 3]. Meanwhile, the shared-stack cooperative threads share one single memory space for a call stack, and this is called the shared-stack. A currently running thread occupies the shared-stack, and thus other waiting threads have to store their stack contents into stack buffers that are located in the opposite memory region to the shared-stack. Copying memory between the shared-stack and the stack buffer occurs at each context switch.

The shared-stack cooperative threads bring about external memory fragmentation in the stack buffer because of frequent stack switching. Memory compaction should be performed to remove the fragmentation. We propose an efficient memory compaction scheme to improve the performance of the shared-stack cooperative threads. The key idea is to perform the memory compaction on demand. Our scheme determines whether to perform or delay the memory compaction according to the expected stack overflow time, which is derived by the amount of free memory space in the shared-stack. To compute the expected stack overflow time, we modeled the stack of the shared-stack cooperative threads with using the continuous time Markov chain. Some studies have used the continuous time Markov chain for their power management models [22, 23].

The rest of this paper is organized as follows. Section 2 presents the shared-stack cooperative threads. Section 3 presents the efficient stack memory compaction for eliminating the shortcomings of the shared-stack cooperative threads. Section 4 presents the evaluation of the proposed scheme. Section 5 presents the related works. Finally, section 6 presents our conclusion.
2. BACKGROUND: THE SHARED-STACK COOPERATIVE THREADS

Cooperative threads define a thread as a flow of control, and programming with using cooperative threads is similar to programming with using multithreads [1]. Therefore, the cooperative threads inherit advantages such as apparent control flow, easy exception handling and automatic state management from the multithreads. Unlike fully preemptive multithreaded systems, a running cooperative thread yields its control of execution to other cooperative threads only at well-defined points such as blocking the I/O operations and the explicit yield calls. This behavioral characteristic makes the cooperative thread systems similar to event-driven systems.

However, the cooperative threads also have some serious problems that are similar to those of general multithreaded systems when it comes to the memory overhead. A cooperative thread also needs a stack for executing and storing its context. Generally each cooperative thread occupies a fixed-size stack that is similar to multithreaded systems. This enables programmers to take charge of the difficult trade-off between making the stack too small and risking stack overflows, or the trade-off between making the stack larger than necessary and wasting memory space.

In order to alleviate the memory overhead that is caused by the fixed-size stacks, we used the shared-stack for the cooperative threads. There is a single shared-stack that is used as a runtime stack for all threads. All the cooperative threads share one stack that is called the shared-stack. 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 heap, 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.

Fig. 1 shows an example that describes the memory layout of the shared-stack cooperative threads. When memory space is allocated to a fixed-size stack for each thread, only a maximum of 4 threads can exist in the system. The unused memory space in the stack of each thread is wasted. However, if we use a shared-stack for these threads, then the wasted memory space (the free space in the Fig. 1) can be used for other purposes in the system.

![Diagram of memory layout](image)

Using the shared-stack may cause extensive memory copy overhead whenever context switches occur. Unlike the fully preemptive multithreaded systems, the context switch can only happen in a well-known set of functions (at I/O points or at explicit yield points), and we can deterministically find out the preemption points in cooperative thread
systems. Because most of the wireless sensor applications have a few fixed I/O points that execute longer than the computation [6], the number of context switches is greatly reduced in cooperative threading. The reduced number of context switches and the fully deterministic preemption points can alleviate the copy overhead.

However, the external fragmentation, which is caused by the frequent stack copying among stacks, cannot be avoided. As shown in Fig. 2, during the context switch from a thread $T_4$ to another thread, $T_2$, the size of $T_4$’s stack is too large to fit into the memory space that is occupied by $T_2$. So, $T_4$’s stack contents are stored in another suitable region and $T_2$ moves its stack contents to the shared-stack. As a result, the memory space that $T_2$ had occupied is fragmented.

Memory compaction is sometimes necessary in order to prevent wasting large amount of memory space that is caused by fragmentation. This memory compaction involves moving several stack contents to the end of the memory space so that all of the fragments are combined into a single large free block. We need to perform as few memory compactions as possible while still reserving sufficient memory space for the shared-stack because this is a very expensive operation in terms of time and energy consumption.

3. THE PROPOSED SCHEME: EFFICIENT STACK MEMORY COMPACTION

The external memory fragmentation of the shared-stack cooperative threads can be resolved by compaction at every thread’s switching time. An optimal compaction time can be achieved by performing the compaction of stack memory right before stack overflow. However, to this end, the sensor operating system should check the stack size at every push operation. The space overhead that is increased by this checking process is proportional to the rate of the push operation. The overhead should be limited to being constant in the sensor nodes because these sensor nodes have a limited amount of memory. Therefore, the optimal solution is impractical in the sensor nodes with limited resources. However, we use the result of the optimal solution as a reference for comparing the performance of our proposed scheme with that of other existing schemes.

The proposed scheme determines whether or not compaction is necessary at every thread switching time, and the proposed scheme is based on the expected stack overflow time. If a thread is expected to safely run until the next switching time begins, then no compaction is performed. Otherwise, compaction is performed and the fragmentation is removed.

For evaluating the expected stack overflow time, we simplified the stack model with the following assumptions:
The push operation increases the size of the stack, and the pop operation decreases it.

The push and pop operations are performed at fixed rates in a sub-task:

In general, the applications of the wireless sensor networks repeatedly perform some sub-tasks. Each sub-task can run until it reaches a fixed preemption point such as the I/O operations, or the yield operation. We may use an average rate for a sub-task in a real application.

The push and pop operations are not coincidently performed.

The underflow does not occur.

Based on the above assumptions, we present a continuous time Markov chain in Fig. 3. Each state means the size of the available stack of a thread. \( \lambda \) and \( \mu \) are the increasing rate and the decreasing rate of a stack, respectively. The value of \( \lambda \) and \( \mu \) can be obtained by static analysis of the executable binary or by the accounting push and pop operations at run-time. \( M \) is the size of the available stack memory + 1. That is, the state \( M \) is the overflow state. If the current state is the state \( M \), then the current state never changes, and it is called an absorbing state. State 0 is the initial state.

If we use a discrete time Markov chain for our stack model instead of a continuous time Markov Chain, then we can obtain the number of state transition events from the initial state to the absorbing state. However, we cannot get the expected overflow time because the time between two successive events is not considered in the discrete time Markov Chain. Therefore, we use a continuous time Markov Chain.

The expected stack overflow time is the expected time of transition from state 0 to state \( M \). To obtain the expected time, we first should know \( p_M(t) \), which is the probability that the Markov process must be in state \( M \) at time \( t \). If we know the \( p_M(t) \), then we can obtain the probability of transition to state \( M \) at time \( t \) by differentiating \( p_M(t) \).

\( p_M(t) \) can be obtained from the transition probability vector \( \mathbf{p}(t) = (p_0(t), p_1(t), p_2(t), \ldots, p_M(t)) \) and the \( \mathbf{p}(t) \) can be obtained by the following Kolmogorov differential equation [7, 8].

\[
\frac{d\mathbf{p}(t)}{dt} = \mathbf{p}(t)Q
\]  

(1)

\( Q \) is an infinitesimal generator matrix, and elements \( (i, j) \) of \( Q \) are defined as follows:

- The transition rate from state \( i \) to state \( j (i \neq j) \).
- A negative of the sum of the transition rates from state \( i \) to other states \((i = j)\).
- We can obtain Eq. (2) by applying Eq. (1) to the Markov model in Fig. 3.
In Eq. (2), \( k \) is an integer from 1 to \( M - 2 \) and \( M \) is greater than or equal to 2. The initial probability vector \( p(0) = (1, 0, 0, \ldots, 0) \).

As mentioned above, we can get the probability of the transition to state \( M \) at time \( t \) by differentiating \( p_M(t) \). We can evaluate the expected stack overflow time by using the probability. That is, the expected stack overflow time is

\[
E(T_{ov}) = \int_0^\infty t \frac{dp_M(t)}{dt} dt.
\]

From Eqs. (2) and (3), we can obtain the expected stack overflow time, \( E(T_{ov}) \) in the case of \( M = 2 \). The result is

\[
E(T_{ov}) = \frac{2\lambda + \mu}{\lambda^2} \quad \text{(in case of } M = 2).\]

However, in the case of \( M > 2 \), it is impractical to get \( E(T_{ov}) \) by hand. Therefore we use a numerical method to solve Eq. (3). We choose the MATLAB ODE (the ordinary differential equation) solver for the Runge-Kutta method [7].

Before applying the ODE solver, we should consider the relationship between \( \lambda \) and \( \mu \). In a general application, the increasing rate and the decreasing rate of a stack are similar because a stack that is increased by function calls will decrease by the return of the function calls. Therefore the assumption of \( \lambda = \mu \) is reasonable. In this paper, we assume that \( \mu = \alpha \lambda (0 < \alpha < 1) \) because we want to consider the burst of the push operations. We choose 0.9 as the value of \( \alpha \). The \( \alpha \) that is smaller than 0.9 makes the model very pessimistic. On the contrary, the larger value makes the model stable and only a few stack overflows occur. We investigated several \( \alpha \) values and chose 0.9 because it can reasonably well represent the burst of the stack growing in the sensor applications.

Figs. 4 and 5 show the MATLAB ODE solver results. Fig. 4 shows the result of \( E(T_{ov}) \) when \( \lambda \) is 5 and \( M \) is from 1 to 100. \( F_{app} \) is a graph of function \( 2M - 20 \). \( F_{app} \) is almost equal to the \( E(T_{ov}) \) if \( M > 20 \). That is, the \( E(T_{ov}) \) is proportional to the size of the stack memory, \( M \). Fig. 5 shows the result of \( E(T_{ov}) \) when \( M \) is 64 and \( \lambda \) is from 1 to 50. \( F_{app} \) is a graph of function \( \frac{540}{\lambda} \). \( F_{app} \) is almost equal to the \( E(T_{ov}) \). That is, the \( E(T_{ov}) \) is inversely proportional to \( \lambda \).

It took about 10 minutes to get the result in Fig. 4 with using MATLAB on a Pentium 1.8GHz system. It is impractical to get \( E(T_{ov}) \) by using the above method on small
sensor nodes. Therefore, we need an approximate method that is fast and light enough to run on the sensor nodes.

In summary, $E(T_{ov})$ is proportional to $M$ and inversely proportional to $\lambda$. The $F_{app}$ below can approximate $E(T_{ov})$.

$$F_{app} = \frac{a}{\lambda}(M + b)$$

We can obtain values $a$ and $b$ by applying the results in Figs. 4 and 5. Consequently, the approximate $E(T_{ov})$ is

$$E(T_{ov}) \approx \frac{10}{\lambda}(M - 10) \quad (M > 10). \quad (4)$$

We can obtain the $E(T_{ov})$ by Eq. (4) when we know the size of the residual stack memory ($M$) and the increasing rate ($\lambda$) of a stack (The decreasing rate ($\mu$) is $0.9\lambda$). If $M$ is smaller than 10, then we assume $E(T_{ov})$ is 0.

4. EVALUATION

4.1 Evaluation on a Single Node

We conducted a simulation to evaluate the performance of the proposed scheme. It is difficult to detect the accurate time of the stack overflow of a sensor node in a real implementation because the system failure by stack overflow is similar to a Byzantine failure. The Byzantine fault is an arbitrary fault, and Byzantine faulty node may be continuously working and response to other nodes’ requests [24]. After a stack overflow takes place, a sensor node may continue its operations but these operations may be faulty.

The simulator presents the stack operations of the concurrently running threads. The
stack operations are performed according to the increasing rate($\lambda$) and the decreasing rate($\mu$). The interval between two successive thread switchings is fixed. The unit of time is the number of thread switchings.

Fig. 6 shows the simulation results from 16 threads whose total sizes of stack memory is 1024, and the increasing rate($\lambda$) is 10. We performed the simulation 100 times and we obtained an average value. $T_{ov}$ in the figure is an execution time of the threads that are without failure by a stack overflow. In other words, $T_{ov}$ is the life time of a sensor node. In the case where compaction is always occurring and the case of our scheme, the threads lived longer than did the threads in the case of no-compaction. In the case where compaction always occurs, there is a little longer execution time than in our scheme. However, in the latter case, the compaction scheme required more compactions than did our scheme. That is, our scheme can greatly reduce the number of compactions while it still maintains similar life time when it is compared to the case of always-compaction.

From these results, we can know that the proposed scheme can extend the overflow time. The applications of sensor networks periodically execute something useful. In one execution epoch, when there is no overflow, the applications can safely run until the process is ended. Though some sensor nodes may fail to run, the sensor networks will still be capable of performing well with the rest of the sensor nodes. Therefore, it is important to increase the probability of completing the execution.

Fig. 7 shows the results of the probability of completing the execution and the number of compactions when the execution time is 3000 and other conditions are equal to the first simulation. We repeated the simulation 100 times. When no compactions occurred, the probability of running up to 3000 was 39%, but when with the compactions are present, the probability increased up to 80%. However, our scheme dramatically reduced the number of compactions when compared to the case of always-compaction.

### 4.2 Evaluation on Multiple Nodes

Wireless sensor networks are composed of multiple sensor nodes, and they are operated by the communication that takes place between the sensor nodes. Even though some sensor nodes may be out of order, the networks can still work because the live nodes are
able to isolate the dead nodes. Therefore, simulating multiple sensor nodes that are concurrently operating is worth analyzing. In this simulation, we extended the single node simulation to multiple sensor nodes that communicate with each other. During the simulation, some sensor nodes stop their operation not by depleting energy, but because of the stack overflow. For a more accurate simulation, we used a synthesized workload that is based on traces from the applications that we ran on our testbed sensor nodes.

The area of the sensor networks that is used in the simulation was assumed to be 100m x 100m. The number of the nodes in the network was assumed to be 100 nodes, and one of these nodes is a sink node while the others are designated to be the sensor nodes. The sink node was located at the center of the field, and the sensor nodes were randomly placed in the field.

All sensor nodes periodically send or forward data to the sink node. The lengths of the intervals between the transmissions of each node are identical for all nodes. This interval was 10 time units. That is, all sensor nodes sent their data to the sink node once per 10 time units. We used the EAR [10] as our routing algorithm. The EAR is a routing algorithm that considers energy efficiency. Every node was given an identical amount of initial energy of 0.2J. We used the similar energy model for wireless transmission that is defined in other studies [10, 12, 13]. The energy of the transmissions was assumed to be 20nJ/bit + 1pJ/bit/m³. The energy for the reception was assumed to be 30nJ/bit. The packet length was assumed to be 256bits. In this simulation, the following assumptions about communication were made:

- Every node knows its position and the distance between itself and other nodes.
- Every node has an identical maximum radio range, i.e., 150m.

The energy that is consumed by the stack memory compaction was assumed as 7600nJ. This value was evaluated based on the energy of memory read/write in Atmega 128, which is popularly used for sensor nodes. The energy of the memory-read instructions is 7.32nJ, and the energy of the memory-write instructions is 7.50nJ [14]. We assumed that the number of bytes that was moved by the compaction is 512, which is half
the size of the stack memory. Therefore, the energy for the compaction is $7600\text{nJ} \approx 512 \times 7.32\text{nJ} + 512 \times 7.50\text{nJ}$.

We excluded the switching overhead because the number of the thread switchings is not affected by the compaction method that is used. In addition, we did not include other energy parameters in order to concentrate on the effect of the communication and the stack overflow and to simplify the simulation. The parameters of the expected stack overflow time were equal to those in the single node simulation. For the simulation, we installed the single node simulation into the sensor network simulator that was used in the previous studies [12, 13].

Fig. 8 shows the average of the residual energy of the sensor nodes at every 1000 time units. Until 3000 time units, the residual energy with no compaction is less than that of always-compaction because there were some dead nodes due to the stack overflow in the no-compaction case. Beyond that, however, the residual energy of always-compaction is less than that of no-compaction because the cumulative compaction overhead is too high. There were dead nodes due to the cumulative compaction energy in the case of always-compaction, and other live nodes continued to consume the compactions’ energy. As shown in Fig. 9, the number of live nodes is similar in both cases of the no-compaction and the always-compaction at 5000 time units. From Figs. 7, 8 and 9, we observed that a very small number of compactions occurred in our scheme and that the proposed scheme showed the most energy efficient results.

![Fig. 8. The average of the residual energy of the sensor nodes with using the EAR.](image)

5. RELATED WORK

Some studies have been conducted to defend the stack overflow. Some of them have used static analysis. StackAnalyzer [15] and Stacktool [16] are representative static analysis methods. StackAnalyzer [15] is a commercial tool for automatically determining the worst-case stack usage of threads. It can show annotations in the call graph and the control flow graph, and it can run across several platforms.

Stacktool [16] is proposed to provide stack safety, which is a guarantee that the call-stack does not overflow, and also to automatically reduce the stack memory require
ments. The stacktool performs an analysis based on the abstract interpretation of machine code. The abstract interpretation ensures that the stack depth is accurately bound. However, the analysis based on abstract interpretation is too slow.

In [17], Torgerson proposed a stack analysis tool that is integrated with an MANTIS OS’s build system; it is relatively fast while the analysis result does not overpass the observed stack usage in most cases. But, we still need to achieve the tightly bounded maximum stack usage to reserve the memory space.

A lot of memory space is still wasted because the sizes of the stacks are stationary during the thread execution although the stack analysis tools that were mentioned above can preclude stack overflow by allocating as much memory space as the worst-case stack usage to each stack.

Some studies such as the stackless Python [18], Capriccio [19] and SESAME [20] do not use a stack. A dynamically allocated stack frame is used instead of a stack. The stack frames are linked to each other. These methods require dynamic memory allocation. However, frequent memory allocation can be a cause of memory fragmentation, and this in turn makes for inefficient memory management.

The stackless Python [18] is an alternative implementation of the Python language to solve the problem of the fixed-size stacks. It uses a linked-frame structure for the run time stack, instead of using contiguous memory space. Similar to the stackless Python, it is possible to implement a stackless version of the C language with the compiler’s support.

Capriccio [19] is a user-level thread package that implements stackless C, which is used by high-concurrency servers. Capriccio uses the linked stack management, which minimizes the amount of wasted stack space by providing safe, small and non-contiguous stacks that can grow or shrink at run time. A compiler analysis makes its stack implementation efficient and sound.

SESAME [20] is one of the practical implementations of the stackless C for sensor operating systems. OTL [21] is an extended version of SESAME for real-time systems. It represents a run-time stack as a linked list of stack frames. The SESAME analyzer, which is a sort of a preprocessor, calculates the amount of stack frames that are used for each function at the compiling time. Based on the calculated results, the SESAME stack
dynamically allocates the necessary memory space for the stack frame when each thread calls a function, and it releases the allocated space when the function returns. The SES-AME removes the fixed-size stacks at the expense of frequent memory allocation/deallocation.

6. CONCLUSION

Preventing stack overflow is needed to provide a safe multi-threaded execution environment for the wireless sensor networks. Mechanisms for general operating systems cannot be used in the sensor operating systems because of the limited resources of the sensor nodes. We have proposed shared-stack cooperative threads to save stack space and to reduce the possibility of stack overflow in this environment.

In the thread switching of the shared-stack cooperative threads, a stack of currently running thread is swapped with a stack of the next running thread. The stack switching brings about external fragmentation in the stack space. Compactions may remove the fragmentation, but the compaction overhead could degrade performance. Therefore, it is necessary to reduce the number of compactions. This paper presents a method for determining whether the compaction must be performed or delayed. Therefore, we computed the expected stack overflow time, and we then determined the compaction time based on it. The results of our simulation demonstrate that the proposed scheme greatly reduces the number of compactions without penalties, and it increases the lifetime of network.

REFERENCES

8. K. S. Trivedi, Probability and Statistics with Reliability Queueing and Computer Sci-
Junyoung Heo (許埈栢) received the B.E. and Ph.D. degrees in Computer Engineering from Seoul National University, Seoul, Korea, in 1998 and 2009 respectively. He has been with the Department of Computer Engineering, Hansung University, Seoul, Korea, since 2009, where he is currently an Assistant Professor. His research interests include operating systems, wireless sensor networks, embedded systems, and fault tolerance.

Boncheol Gu (具本喆) received his B.S. degree in Computer Science from Yonsei University, Seoul, Korea, in 2004. He is currently a Ph.D. candidate of School of Computer Science and Engineering, Seoul National University, Seoul, Korea. His research interests include operating systems, embedded systems and software-defined radio.

Yookun Cho (曹裕模) received the B.E. degree from Seoul National University, Seoul, Korea, in 1971 and the Ph.D. degree in Computer Science from the University of Minnesota at Minneapolis in 1978. He has been with the School of Computer Science and Engineering, Seoul National University, since 1979, where he is currently a Professor. His research interests include operating systems, algorithms, system security, and fault-tolerant computing systems. Dr. Cho was president of the Korea Information Science Society during 2001.

Jiman Hong (洪志樹) received the B.S. degree in Computer Science from Korea University, Seoul, Korea, in 1994 and the M.E. and Ph.D. degrees in Computer Engineering from Seoul National University in 1997 and 2003, respectively. He has been with School of Computing, Soongsil University, Seoul, Korea, since 2007, where currently he is an Assistant Professor. From 2004 to 2007, he was a Professor of Kwangwoon University, Seoul. His research interests include embedded operating systems, fault tolerance computing systems, distributed computing systems, and sensor network systems.