Multithreading Optimization Techniques for Sensor Network Operating Systems

Hyoseung Kim and Hojung Cha

Department of Computer Science, Yonsei University
Seodaemun-gu, Shinchon-dong 134, Seoul 120-749, Korea
{hskim, hjcha}@cs.yonsei.ac.kr

Abstract. While a multithreading approach provides a convenient sensor application developing environment with automatic control flow and stack managment, it is considered to have a larger data memory requirement and energy consumption than an event-driven model. Current threaded sensor operating systems unfortunately do not provide appropriate solutions. This paper presents multithreading optimization techniques for sensor network operating systems. Our work focuses on the three major problems of implementing threads on resource-constraint sensor nodes—memory resources, energy consumption, and scheduling policy. Single kernel stack and the thread stack-size analysis techniques reduce the RAM requirement of thread model. The variable timer saves energy consumption and the event-boosting thread scheduling reflects the characteristics of sensor applications and provides fast response time to threads. The experimental results on a common sensor node show that the multithreaded system could be effectively implemented with reasonable overhead.

Keywords: sensor network operating system, multithreading optimization technique.

1 Introduction

Wireless sensor networks have been well-studied in terms of the increasing variety of hardware and the development of diverse applications which now require sophisticated system software. Sensor nodes are normally battery-operated, memory-limited and have low computational power. Sensor network operating systems, therefore, should support high concurrency with minimal memory usage and low energy consumption. Unlike general purpose operating systems, popular sensor operating systems such as TinyOS [1] and SOS [2] adopt an event-driven model to meet these tight constraints. They execute applications with reactive event handlers and cooperatively-operated run-to-completion tasks. Li *et al.* [3] reported that the event-driven TinyOS achieves about a 30-fold improvement in data memory requirement and a 12-fold reduction in power consumption over general purpose multithreaded embedded operating systems.

Although the event-driven sensor operating systems are implemented efficiently in a resource-constraint environment, they do not provide all the functions of general

K. Langendoen and T. Voigt (Eds.): EWSN 2007, LNCS 4373, pp. 293 – 308, 2007.
© Springer-Verlag Berlin Heidelberg 2007

purpose operating systems. Developers typically suffer from the manual configuration when programming applications. With TinyOS and SOS, for example, developers have to split a long-running task, which can unnecessarily delay other tasks, into several phases. Programmers also take responsibility for managing event handlers' states. As tasks are inter-dependent in the execution context, the repeated analysis of system concurrency and system restructuring is inevitable in order to meet the changes in application requirements [4].

In contrast to the event-driven approach, multithreading inherently provides high concurrency with preemption and automatic state management. It also allows programmers to specify control flow. Sensor applications frequently request sensing and radio communication. With an event-driven model, the programmers may split one conceptual function into multiple functions for I/O operations, but multithread systems easily handle it by blocking the I/O interface. Synchronization and deadlock problems experienced with thread [5] can be managed by compiler support and development tools [6]. As most existing development tools are based on threads, multithread systems could provide a general and more efficient development environment. However, threads tend to incur more time and space overhead than events. Stack reservation for each stack is indispensable for multithreading systems. Context switching overhead is caused by preemption and blocking. Kernel services, such as a scheduler and system timer management, are also required. Data memory and energy requirement of multithreading is an obvious obstacle to the resource-constraint sensor nodes. In addition, a scheduling policy optimized for sensor applications should be developed. Although the multithreading approach is attractive in sensor application developments, current thread-based sensor operating systems [7, 8] do not provide appropriate solutions for overhead problems or a scheduling issue. This has motivated us to develop multithreading techniques specifically designed for sensor nodes.

This paper presents optimized techniques for the implementation of multithreaded sensor network operating systems. The thread model for sensor nodes should consider memory resource, energy consumption, and scheduling policy. The proposed techniques contribute to each issue by providing multithreading functions while consuming reasonable overhead compared to existing event-driven sensor operating systems. Our techniques are implemented in the RETOS operating system [9, 10, 11], although they are applicable to other thread-based sensor operating systems. The effectiveness of the proposed techniques is validated by experiments conducted on a commercial mote running the RETOS operating system.

The rest of this paper is organized as follows: Section 2 describes the proposed multithreading optimization techniques; Section 3 validates the effectiveness of the mechanism through real experiments; Section 4 discusses related work; and Section 5 concludes the paper.

2 Thread Optimization for Sensor Applications

This section explains optimization techniques for implementing threads on sensor nodes. Considering the resource constraints of conventional sensor hardware, we propose various techniques in our work: *Single kernel stack* reduces the size of thread

stack requirement, and *Stack-size analysis* automatically assigns an appropriate stack size to each thread. *Variable timer* reduces the overhead of system timer, hence reducing energy consumption. *Event-boosting thread scheduler* satisfies the response time requirement for sensor applications.

2.1 Single Kernel Stack

Multithread systems require stack reservation for each thread. The amount of the required stack of a thread is the sum of the resource required by thread functions, system calls, interrupt handlers and hardware context saving. In general, sensor applications are implemented with system API such as radio packet transmission, and system calls and interrupt handlers use a large portion of the thread stack. Considering these characteristics, we propose single kernel stack management for data memory efficiency. Single kernel stack management separates the thread stack into kernel and user stacks, and maintains a unitary kernel stack for system calls and interrupt handlers to reduce the thread stack bound. Equation 1 explains the total stack size for the multiple kernel stack system. The size for the single kernel stack system is obtained by Equation 2. The sum is computed for the number of application threads. Without any threads, multiple kernel stack systems and single kernel stack systems have the same data memory usage for stack. However, the effect of the single kernel stack becomes more significant when the number of threads increases.

$$\sum\{\max(\mathit{system\,call}) + \max(\mathit{ISR}) + h / w\,\mathit{context}\} \tag{1}$$

$$\max(\mathit{system\,call}) + \max(\mathit{ISR}) + \sum(h / w\,\mathit{context}) \tag{2}$$

In the single kernel stack system, the kernel stack is shared among every thread. A controlled access to the kernel stack is implemented in such a way that the system does not arbitrarily interleave execution flow, including thread preemption, while in the kernel mode. Thread switching could be performed immediately prior to returning to user mode and executing an idle function, such as at the time when all work pushed on the kernel stack is completed. With thread preemption, hardware contexts are saved in each thread's thread control block (TCB) due to kernel stack sharing.

Although the single kernel stack is unable to preempt threads in the kernel mode, it does not inhibit real-time operation of the kernel. With this technique, the execution context and the development environment of the kernel and the user are isolated. Because the kernel is interrupt-driven, the kernel developer, based on underlying system analysis, gives high concurrency to system components such as device drivers or the network stack.

2.2 Stack-Size Analysis

With MMU-less hardware, application developers must estimate accurate thread stack size to optimize the memory usage. A given stack size that is less than the size required by the thread causes stack overflow and easily crashes a system. Assigning a

large stack would cause data memory overhead. The proposed stack-size analysis provides minimal and system-safe stack requirements for each thread, so the kernel automatically allocates an appropriate stack size for threads.

Table 1. MSP430 instructions concerned with stack usage

Instruction	Stack usages	Description
push var	+ 2	Push a value
pop var	- 2	Pop a value
call #label	+ 2	Push return address
sub SP, N	+ N	Directly adjust stack pointer (function prologue)
add SP, N	- N	Directly adjust stack pointer (function epilogue)

The proposed stack analysis produces a control flow graph of an application. Function label, start address and internal stack usage are used as nodes in the graph, and branch instructions are used as edges. The technique then calculates the maximum possible thread stack size with a straightforward depth-first search. The operations are conducted with a binary image, which results from linking the application programmer's code with libraries and compiler-generated codes. Table 1 shows the TI MSP430 instructions, which are related to detecting a function's stack usage. Unlike previous stack bounding techniques [12, 13], which focus on the behavior of the interrupt handler, the proposed technique is based on a system where interrupts are handled by the kernel stack. Thus, this technique determines exact stack usages of functions using the instructions listed in Table 1 only.

The set of start nodes for traversing the flow graph consists of every thread function in an application. Finding out the start nodes depends on the programming language and thread library. On the RETOS operating system, where a user programs a sensor application with standard C and pthread library, we can detect the start node for the main thread with the label "main" and each child thread with the parameter of pthread_create(). The thread start address and stack requirement are stored in the header field of application files, and the kernel looks up the information to create a new thread with optimal stack size.

The proposed technique, however, cannot analyze stack size if the application uses recursive calls or indirectly addressed function calls. Recursive calls create cycles in the flow graph and indirect calls cause a disconnect in the flow graph. In these cases, there is no proper way to know the accurate thread stack size. We have implemented the proposed technique as a tool that notifies users if the analysis fails. In addition, we allow users to determine the default thread size on the RETOS operating system, which is equipped with the application safety mechanism [9]. The mechanism inserts dynamic checking code for stack safety to the application, when the stack-size analysis fails. With the safety mechanism, users do not need to be aware of any restrictions such as explicit prohibition of recursive calls, and the system is safe.

2.3 Variable Timer

The multithreading model of computation generally incurs energy overhead due to context switching, scheduler execution, and system timer management. Context switching and scheduling are known to be the source of major overhead in threaded systems. However, the frequency of scheduling in the threaded system is much lower than that of passing messages between handlers in the event-driven system [6], and the context saving and restoring overhead is only a moderate issue in common sensor nodes [8]. In our work, we propose a variable timer technique to minimize energy consumption of the multithreading system.

The system timer manages timer requests from threads and updates the remaining time quantum of currently running threads. In general-purpose threaded systems, the timer management relies on a periodic timer interrupt. This continuously triggers the interrupt handler whether timer handling requests are present or not, and so increases energy consumption of the sensor node, which stays idle most of the time. Moreover, the periodic timer interrupt restricts the time accuracy within the timer interval. If the interrupt interval is reduced, a significant amount of system power is wasted in order to handle the interrupt. Instead of the periodic timer, the system may use a variable-time tick rate by way of reprogramming the tick rate with an upcoming timeout request. The variable timer can solve these problems. General purpose systems do not use the variable timer because the cost of reprogramming timer requests from hundreds of threads is much higher than for the periodic timer interrupt. Alternately, sensor network applications are typically programmed with a relatively small number of threads and timer requests. Thus, it is reasonable to adopt the variable timer tick rate for threaded sensor systems.

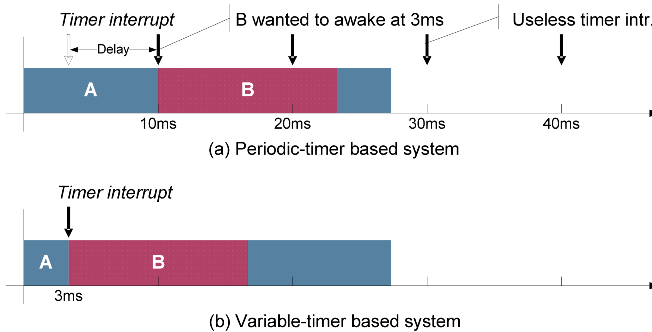


Fig. 1. Variable- and periodic-timer based systems

The variable timer reprograms the timer interrupt interval to the earliest upcoming timeout among the time quantum of currently running thread and the timer requests, such as the sleep() system-call. Figure 1 compares the periodic timer and variable timer systems. General-purpose systems handle the time quantum expiration through

the periodic timer interrupt. In Figure 1(a), thread B wants to wake up after 3ms, but with the 10ms interval it is difficult to meet this request in the system. Unnecessary timer interrupts are generated per 10ms. Figure 1(b) shows the case of the variable timer system; thread B can preempt other threads at 3ms, which is the time when thread B is originally requested, and no more timer interrupts are invoked. The effect of the variable timer system depends on the cost and frequency of timer reprogramming. Section 3 evaluates the correlation of the cost for reprogramming a timer and the frequency on a real sensor node device.

2.4 Event-Boosting Thread Scheduling

The RETOS operating system supports the POSIX 1003.1b real-time scheduling interface [19] to enable both programmers' explicit priority assignment and kernel's dynamic priority management. Threads are scheduled by three policies, SCHED_RR, SCHED_FIFO, and SCHED_OTHER, and the system-calls are provided for programmers to adjust their policy and priority. SCHED_OTHER is the default policy and always has less priority than SCHED_RR or SCHED_FIFO.

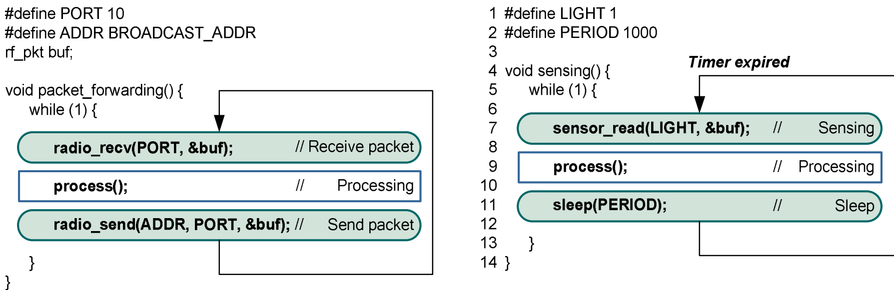


Fig. 2. Typical sensor applications on the multithreading system

We now describe the SCHED_OTHER policy proposed in our work. Although users do not manually give priority assignment to application threads, the operating system should satisfy threads with fast response time. Figure 2 shows typical sensor application codes. The key objectives of common sensor applications are packet forwarding and sensing. Threads usually receive a packet, process data, and forward the result. Threads also collect sensor data, process it, and sleep for a regular period. General operating systems typically classify threads into I/O bound and CPU bound, and they prefer I/O bound threads for high interactivity. From the aspect of sensor node operation, almost every sensor thread is treated as I/O bound, or else the thread property is infinitely switched between I/O bound and CPU bound due to the iteration of I/O and computation in the sensor thread. Therefore, a scheduling policy which specifically concerns sensor network applications should be developed to provide fast response time.

Table 2. Priority adjustment for event-boosting

	Dynamic priority	Description
Init.	4	Thread created
sleep()	+3	Timer request
radio_recv()	+2	Radio event request
sensor_read()	+1	Sensor event request
Consuming CPU time	- 1 per 8ms	Decrease dynamic priority

In our work, we propose an event-boosting thread scheduler to increase the event response time of threads. The scheduler directly boosts the priority of the thread requesting to handle a specific event. Events in the sensor network applications are defined as the expiration of the timer request, the reception of a packet, and the completion of sensing. A thread issues a blocking system-call to handle one of these events, and the kernel enhances the thread's priority according to the type of system-call. When an event occurs, the priority-boosted thread will be able to rapidly preempt other threads. The priority of the thread reduces with the CPU-consumed time. Hence, other threads have chances to be re-scheduled. Table 2 shows the priority adjustment for event-boosting scheduling policy. Threads are created with the initial priority, and obtain higher priority if they call sleep(), radio_recv(), and sensor_read() system-calls. Thread priority is decreased by 1 per 8ms of consumed CPU time. Concerning the priority adjustment, we have not conducted any formal evaluation on the value of adjustment, but rather used a subjective user study on the RETOS operating system. We considered that the explicit timer request is the most critical job and the radio event is more important than the sensor event.

In the real implementation, it is also important to avoid starvation and to provide fairness. Therefore, we compare the remaining thread time quantum if there are equally prioritized SCHED_OTHER threads. When all threads in the run-queue have exhausted their time quanta, the scheduler re-computes the time quantum duration of all threads in the system. The idea for assigning a new quantum is adopted from Linux, which gives half the previously remaining quantum plus a default time quantum to threads.

3 Evaluation

This section presents the experimental results of the proposed multithreading techniques. The experiment evaluates the efficiency of single kernel stack and stack-size analysis, the timer handling overhead of variable timer management, and the concurrency supports of an event-boosting scheduler. Furthermore, the overall effect of optimization techniques are validated by running a real sensor application both on RETOS and TinyOS, the former being a dual mode based multithreaded system and the latter being a single mode and event-driven operating system. RETOS have been implemented for the TI MSP430 F1611 (8Mhz, 10Kb RAM, 48Kb Flash) and CC2420 (IEEE 802.15.4) based Tmote Sky hardware platform [14]. The execution results are based on the average results over 10 sets of 30 runs.

3.1 Effect of Stack Optimization

The single kernel stack and stack-size analyses are to optimize stack usage of the multithreaded system. To adequately evaluate the efficiency of these techniques, we considered entire stack usage on the system. Seven sensor network applications were used for the test. *MPT_mobile* and *MPT_backbone* are decentralized multiple-object tracking programs [15]. When the *MPT_mobile* node moves around, it sends both an ultrasound signal and beacon messages every 300ms to nearby *MPT_backbone* nodes. *MPT_backbone* nodes then report their distance to the mobile node, and *MPT_mobile* computes its location using trilateration. *R_send* and *R_recv* are programs to send and receive radio packets with reliability. *Sensing* samples the data and forwards it to the neighbor node. *Pingpong* makes two nodes blink in turns by means of a counter-exchange. *Surge* is a multihop data collecting application which manages a neighbor table and routes the packet.

Table 3. Kernel stack and thread context block requirements

	Kernel stack size (byte)	Increase of TCB+ H/W context (byte)
Single kernel stack system	76	18
Multiple kernel stack system	76	16

Table 4. Efficiency of a single kernel stack based system

Applications	Num. of Threads	User stack (byte)	Data section (byte)	Kernel stack (byte)	
				Multiple	Single
MPT_backbone	1	68	131	152	76
MPT_mobile	2	78	416	228	76
R_send	3	78	217	304	76
R_recv	3	50	214	304	76
Sensing	2	18	157	228	76
Pingpong	1	8	106	152	76
Surge	4	98	336	380	76

In order to evaluate the single kernel stack, we have implemented two versions of RETOS to measure the effectiveness of stack usage reduction. For the easy stack size comparison, the multiple kernel stack system also stores the hardware context in the TCB. Table 3 shows the size of the required kernel stack and the increase of TCB plus the hardware context for each kernel. The kernel stack requirement is detected by executing all system-calls and interrupt handlers in each system. As the two systems have the same kernel control flow except the kernel stack management scheme, the kernel stack size on the single kernel stack system is identical with the size on the multiple kernel stack system. The increase of TCB plus the amount of saving the hardware context on the single kernel stack system, however, differs from the multiple kernel stack system, since the single kernel stack system requires two more bytes to store the thread return address.

Table 4 shows the results of running sensor applications on two systems. With the multiple kernel stack system, the kernel stack is required for the idle thread and each application thread. Meanwhile, the single kernel stack system uses only 76 bytes of RAM for the kernel stack independent of the number of application threads. The more threads that are created, the more significant the expected stack efficiency on the this system. Our results also show that the stack reservation overhead on the threaded system is trivial. Most of sensor application threads require a little stack size. *Sensing* and *Pingpong* applications, for instance, can be implemented with 18 and 8 bytes of user stack, respectively.

As described in Section 2.2, we have developed a stack-size analysis technique. For the seven sensor applications, the estimated maximum stack size was compared with the worst stack depth via simulation. The sensor application does not use an indirectly addressed function call, so the technique successfully analyzes each program's stack size. The results of the proposed technique were equal to 1 or 2 words more than the results of simulation, and the technique gave the same call graph with the program's control flow, which was determined manually. We also tested this technique on a program which uses indirect function calls. For this program, the technique could not produce a call graph. However, the stack overflow of an application with an immediate stack size was detected in run-time, indicating that the system safety was maintained.

3.2 Effect of Variable Timer

We have implemented two versions of the system using the variable and periodic timer techniques. Since the major difference in the energy consumption between the two systems is the amount of CPU usage, we measured the active CPU time to estimate the energy consumption. Figure 3(a) shows the performance efficiency of the variable timer compared to the periodic timer. One tick in the variable timer is 1ms, and 10ms on the periodic timer. The experimental results include the execution time of a timer interrupt handler and a timer reprogramming routine. The effectiveness of variable timer differs from the execution cycle of applications. *MPT_mobile, R_send, Sensing, Pingpong,* and *Surge* are periodic programs and are executed every 300ms, 100ms, 1000ms, 1500ms, and 2048ms, respectively. *R_send* is the most energy consuming program among the seven benchmark applications. Because *R_send* transmits a radio packet every 100ms and performs ACK and the timeout-based packet retransmission, it creates more frequent timer requests than other applications. *MPT_backbone* and *R_recv* are reactive applications, which are only executed with radio packet reception. *Pingpong* and *Surge* have a relatively slow execution period. Therefore, these applications get significant energy reduction on the variable timer based system.

In order to clearly compare the performance of variable timer and the periodic timer by the timer request interval, we measured the overhead of the timer management routine. The blink application was used for the evaluation by adjusting the period from 20ms to 1000ms. Figure 3(b) shows the results of this evaluation. When the timer request interval is long, variable timer system spends definitively less overhead than the periodic timer system. However, as the timer interval increases, the overhead on the variable timer becomes larger. At the 20ms of interval, two systems

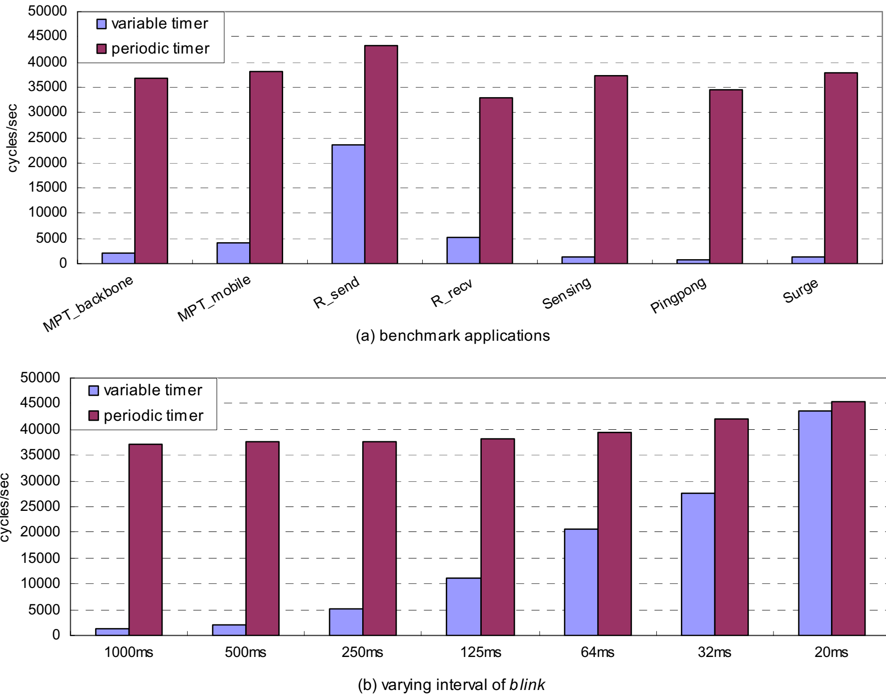


Fig. 3. Timer management overhead

have almost the same overhead because the variable timer takes a longer time per each timer request than the periodic timer, due to the time required to determine the next upcoming timeout event and reprogram the system timer. In this experiment, the periodic timer system has a 10ms tick. If the periodic timer system is implemented to use a 1ms tick as with the variable timer, more overhead would be required to handle timer interrupts.

3.3 Effect of Event-Boosting Scheduling Policy

This section shows that the event-boosting scheduler can effectively satisfy sensor applications' event requests. The test application is a packet round-trip program which continuously sends and receives a packet between two nodes. The first node sends a packet out while the second node receives and returns it to the first node. The first node waits for a reply from the second node and then repeats this process. The round-trip application runs with only two nodes, so that the influence of radio channel and back-off time of the MAC is minimized. The response time for the thread to handle a packet depends on the number of other threads in the system and the scheduling policy. Hence, we can evaluate the functionality of a thread scheduler with radio throughput of the application. The other threads in our experiment are designed to add loads to the scheduler, and perform 10ms of computation each at 100ms intervals.

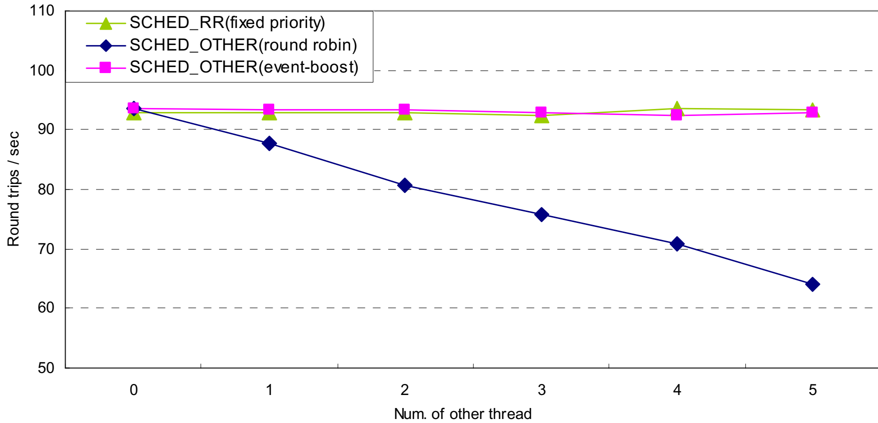


Fig. 4. Scheduling policy comparison

Figure 4 represents the number of round-trips per minute according to the scheduling policy. If users are not concerned with adjusting scheduling policy, SCHED_OTHER is the default policy for threads. In the case of implementing SCHED_OTHER as a simple round-robin, the number of round-trips decreases according to the increment of the other threads. Because preemption is not performed when other threads do not finish their execution or exhaust their time quantum, the radio packet handling is delayed. In the case of explicitly configuration of the round-trip thread as SCHED_RR, the application, which has always higher priority than others, maintains a fixed round-trip performance independent of the number of other threads. The performance of the system, which uses the proposed event-boosting technique for SCHED_OTHER threads, is nearly the same as the case of SCHED_RR. Although users do not manually configure the priority of threads, the dynamic priority adjustment of the event-boosting scheduler minimizes the event handling delay of sensor application threads.

3.4 RETOS vs. TinyOS

This section compares the multithreaded operating system RETOS with the event-driven TinyOS by developing a sensor application. TinyOS is a component-based operating system, and has no distinction between the kernel and the application. Components are programmed with event-driven model and compiled to a single binary image. RETOS provides a rich development environment with preemptive threads. The proposed thread optimization reduces the overhead of traditional multithreading and increases thread response time. We used RETOS v0.96 and TinyOS v1.1.13 for this experiment, and the applications used in the experiment are MPT and a simple packet transmission. MPT is a mobile object tracking program [15] based on ultrasound localization technique. MPT consists of mobile node and backbone node, and the mobile node computes its location using trilateration every 300ms. The trilateration takes approximately 16ms to determine the location. We have considered inserting a simple code which periodically sends and receives a radio

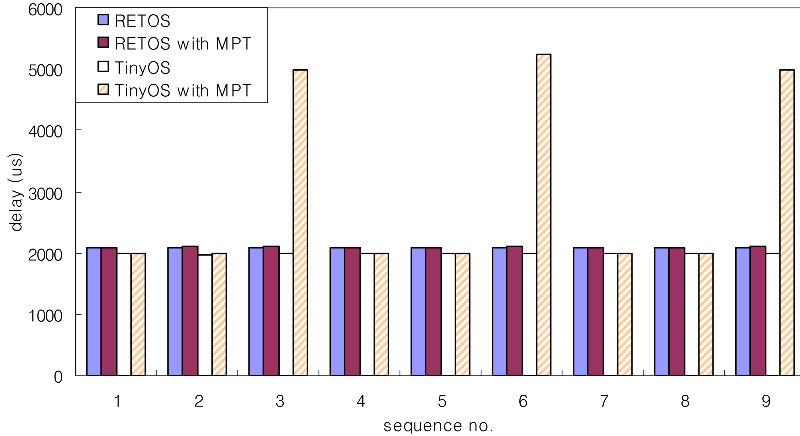


Fig. 5. Packet handling delay

packet to the above application. The sink node transfers a packet every 100ms, and the mobile node receives and counts it. We measured the time from the FIFOP interrupt handling at the CC2420 radio driver to the packet at the thread.

Figure 5 shows the packet-handling latency on RETOS and TinyOS. The purpose of this experiment was to understand the dependency of MPT execution time and packet handling. The results were measured after the two applications' start time was synchronized. With the RETOS system, the packet-handling latency was almost the same whether or not MPT was run, because application threads are preemptive and a packet was received by the radio device driver located in the kernel. In the case of the TinyOS system, the packet-handling latency without MPT was slightly shorter than RETOS. With MPT, the latency was considerably longer than in the other three cases. The extended latency was caused by long computation time of trilateration in the MPT application, which can delay the packet-handler task's execution on the TinyOS.

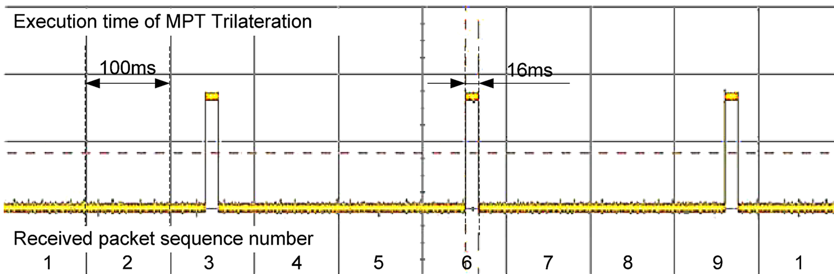


Fig. 6. Execution time of MPT Trilateration observed by oscilloscope

Figure 6 shows the execution pattern of trilateration as observed by oscilloscope. Trilateration was performed every 300ms and took approximately 16ms. We tried to reduce the delay by splitting the trilateration into several phases, but this made the

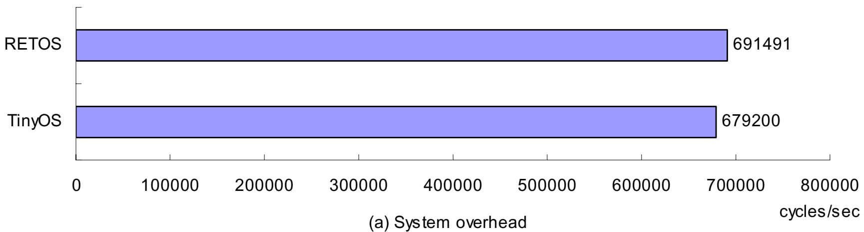
Table 5. Code size for MPT application

	TinyOS (bytes)		RETOS Kernel (bytes)		RETOS Lib. + App. (bytes)		RETOS Total (bytes)	
	ROM	RAM	ROM	RAM	ROM	RAM	ROM	RAM
MPT Backbone	12614	467	18314	748	492	143	18806	891
MPT Mobile	17222	701	18314	748	6848	434	25162	1182

program control flow complex and rendered it difficult to manage the increased number of states. Moreover, measurement of the execution time of each code fragment was necessary to determine whether the split provided reasonable performance.

As RETOS is a multithreaded operating system, it is considered to have more time and space overhead than the event-driven TinyOS. Table 5 shows the code size of MPT on RETOS and TinyOS. The RETOS system uses less than 30Kbytes of flash memory and 2Kbytes of RAM. Although the code size of RETOS is bigger than that of TinyOS, RETOS supports functionality such as application safety mechanism [9], dynamic loadable module [10] and the network stack [11], which are barely supported by the native TinyOS system.

Figure 7(a) compares the computational overhead of MPT with RETOS and TinyOS. *MPT_mobile* spends 2% more overhead with RETOS; it performs thread preemption and scheduling, and also dynamic code checking for system safety. Figure 7(b) shows



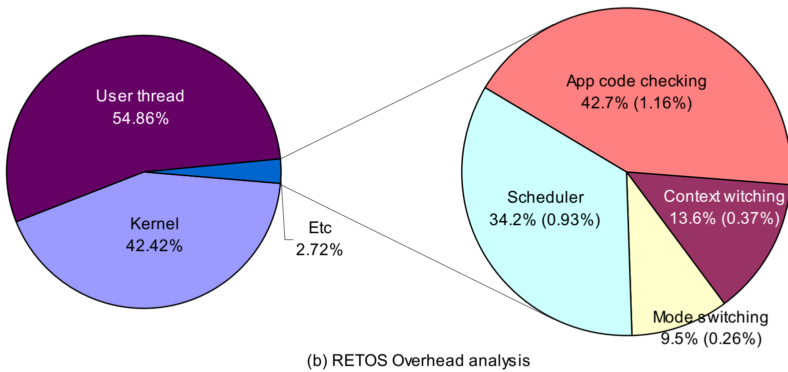


Fig. 7. Computational overhead

the CPU usage distribution of the RETOS system. The user thread occupies 55% of total processing time. The kernel portion is approximately 42% due to the frequent use of radio communication. On the other hand, the amount of calculation time caused by mode switching, scheduler execution and context switching is trivial, compared with the entire processing time. The portion of context switching, mode switching and scheduler execution overhead may be bigger when an application requires little radio communication or computations. Nevertheless, the experiment results show that multithreading could be implemented with reasonable overhead on current sensor node hardware.

4 Related Work

TinyOS [1], the industry defacto sensor network operating system, is based on an event-driven model and provides nesC [16] programming language. TinyOS is considered to provide high concurrency without thread stack reservation, which is essential to multithreading. SOS [2] provides dynamically loadable modules and adopts an event-driven model to avoid context switching overhead for multithreading. However, event-driven models can be inconvenient when developing applications. As event handlers are run to completion, programmers must split long-lived tasks into several phases of codes for concurrency. The tasks of the event-model cannot be blocked, hence a single conceptual function with an I/O operation should be divided into two separate sub functions, one for before and the other for after the I/O operation. The stack frame in the split function is manually maintained by programmers, and it increases the use of global variables. These issues of event-driven model induce poor software structure and render it difficult to debug and develop applications [6, 17].

MANTIS [8] provides a multithreaded programming model, which implement traditional multithreading in sensor nodes. The system shows that programming long-running tasks is much easier than in an event-driven model. With the MANTIS system, programmers heuristically assign a stack size to each thread. If the stack size is too big, the system will suffer from memory insufficiency. If the stack is too small, stack may overflow and system will fail. In MANTIS, fixed priority scheduling based on round-robin is not able to fully utilize the advantage of preemption without the programmer making a manual priority adjustment. As the MANTIS scheduler is executed every 10ms, the overhead for context switching and timer interrupt handling is not trivial. Contiki [7] provides a thread library that works on the event-driven system. With Contiki, programmers empirically choose an appropriate programming model among event-driven, protothread [18], and multithread libraries to develop an application. Hybrid approaches have been studied to integrate the merits of an event model and thread-based model. Adya et al. [17] suggest the combined usage of event and thread model in the same program, but this requires programmers to thoroughly understand the differences between the two models and to appropriately choose the alternatives. Protothread [18] does not require stack reservation; however it cannot maintain local variables and can block only in an explicitly declared area.

5 Conclusion

In this paper, we described multithreading optimization techniques for sensor applications development. Our techniques contribute to possible solutions toward three major problems involved in the implementation of threaded operating systems on resource-constraint sensor nodes—memory resource, energy consumption, and scheduling policy. Single kernel stack and stack-size analysis techniques reduce the memory requirement of a thread model. Variable timer achieves power reduction by improving the timer management scheme. Event-boosting scheduling policy reflects the characteristics of sensor applications and provides fast response time of threads without explicit priority configuration. With the proposed techniques, the overhead of multithreading is reported to be approximately 2% of the total execution time on the TI MSP430 processor, and the system guarantees minimal response delay to sensor applications.

Application libraries or system calls are being implemented, and extensive testing is also conducted on the RETOS sensor operating system. We are presently improving the performance and the energy efficiency of the network stack for radio communication on RETOS, as well as implementing device drivers for diverse sensors and porting them to other processors.

Acknowledgements

This work was supported by the National Research Laboratory (NRL) program of the Korea Science and Engineering Foundation (2005-01352), and the MIC(Ministry of Information and Communication), Korea, under the ITRC(Information Technology Research Center) support program supervised by the IITA(Institute of Information Technology Advancement) (IITA-2006-C1090-0603-0015).

References

1. Hill, J., Szewczyk, R., Woo, A, Hollar, S., Culler, D., and Pister, K.: System architecture directions for network sensors. In Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Cambridge, MA, 2000.
2. Han, C. C., Rengaswamy, R. K., Shea, R., Kohler, E., and Srivastava, M.: SOS: A dynamic operating system for sensor networks. In Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services (Mobisys), Seattle, WA, 2005.
3. Li, S.-F., Sutton, R., and Rabaey, J.: Low Power Operating System for Heterogeneous Wireless Communication Systems. In Proceedings of the Workshop on Compilers and Operating Systems for Low Power, Barcelona, Spain, 2001.
4. Regehr, J., Reid, A., Webb, K., Parker, M., and Lepreau, J.: Evolving real-time systems using hierarchical scheduling and concurrency analysis. In Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS), Cancun, Mexico, 2003.
5. Ousterhout, J. K.: Why threads are a bad idea (for most purposes). Invited Talk at the 1996 USENIX Technical Conference, 1996.

6. Behren, R., Condit, J., and Brewer, E.: Why events are a bad idea (for high-concurrency servers). In Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS), Lihue, Hawaii, 2003.
7. Dunkels, A., Grönvall, B., and Voigt, T.: Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In Proceedings of the 1st IEEE Workshop on Embedded Networked Sensors (EmNetS), Tampa, Florida, 2004.
8. Bhatti, S., Carlson, J., Dai, H., Deng, J., Rose, J., Sheth, A., Shucker, B., Gruenwald, C., Torgerson, A., and Han, R.: MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. ACM/Kluwer Mobile Networks & Applications, Special Issue on Wireless Sensor Networks, vol.10, no.4, 2005.
9. Kim, H., and Cha, H.: Towards a Resilient Operating System for Wireless Sensor Networks. In Proceedings of the 2006 USENIX Annual Technical Conference, Boston, MA, 2006.
10. Shin, H., and Cha, H.: Supporting Application-Oriented Kernel Functionality for Resource Constrained Wireless Sensor Nodes. In Proceedings of the 2nd International Conference on Mobile Ad-hoc and Sensor Networks, Hong Kong, China, 2006.
11. Choi, S., and Cha, H.: Application-Centric Networking Framework for Wireless Sensor Nodes. In Proceedings of the 3rd Annual International Conference on Mobile and Ubiquitous Systems: Networks and Services, San Jose, CA, 2006.
12. Regehr, J., Reid, A., and Webb, K.: Eliminating stack overflow by abstract interpretation. In Proceedings of the 3rd International Conference on Embedded Software, Philadelphia, PA, 2003.
13. Brylow, D., Damgaard, N., and Palsberg, J.: Static checking of interrupt-driven software. In Proceedings of the 23rd International Conference on Software Engineering, Toronto, Canada, 2001.
14. Tmote Sky. http://www.moteiv.com.
15. Yi, S., and Cha, H.: Active Tracking System using IEEE 802.15.4-based Ultrasonic Sensor Devices. In Proceedings of the 2nd International Workshop on RFID and Ubiquitous Sensor Networks, Seoul, Korea, 2006.
16. Gay, D., Levis, P., Behren, R., Welsh, M., Brewer, E., and Culler, D.: The nesC Language: A Holistic Approach to Network Embedded Systems. In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI), San Diego, CA, 2003.
17. Adya, A., Howell, J., Theimer, M., Bolosky, W. J., and Douceur, J. R.: Cooperative Task Management Without Manual Stack Management. In Proceedings of the 2002 USENIX Annual Technical Conference, Monterey, CA, 2002.
18. Dunkels, A., Schmidt, O., and Voigt, T.: Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. In Proceedings of the 4th ACM Conference on Embedded Networked Sensor Systems (Sensys), Boulder, Colorado, 2006.
19. POSIX 1003.1B. http://www.unix.org/version3