PAVNET OS: A Compact Hard Real-Time Operating System for Precise Sampling in Wireless Sensor Networks

Shunsuke SARUWATARI *, Makoto SUZUKI *, and Hiroyuki MORIKAWA *

Abstract: The paper shows a compact hard real-time operating system for wireless sensor nodes called PAVNET OS. PAVNET OS provides hybrid multithreading: preemptive multithreading and cooperative multithreading. Both of the multithreading are optimized for two kinds of tasks on wireless sensor networks, and those are real-time tasks and best-effort ones. PAVNET OS can efficiently perform hard real-time tasks that cannot be performed by TinyOS. The paper demonstrates the hybrid multithreading realizes compactness and low overheads, which are comparable to those of TinyOS, through quantitative evaluation. The evaluation results show PAVNET OS performs 100 Hz sensor sampling with 0.01% jitter while performing wireless communication tasks, whereas optimized TinyOS has 0.62% jitter. In addition, PAVNET OS has a small footprint and low overheads (minimum RAM size: 29 bytes, minimum ROM size: 490 bytes, minimum task switch time: 23 cycles).

Key Words: wireless sensor networks, operating system, hard real-time, sampling.

1. Introduction

Wireless sensor networks (WSNs) have a number of potential fields of applications, including habitat monitoring, military applications [1], wildland fire monitoring [2], volcano monitoring [3], and structural monitoring [4]–[6]. Each application has different requirements for communication protocols and sensing tasks, and sensor nodes have very limited physical resources because of their design requirements, namely, low power, low cost, and small size.

In [7], Saruwatari et al. have developed a hardware and software framework for wireless sensor networks, and applied it to the various sensor network services. In a part of the applications, we found that sensor nodes must perform hardware-tightened tasks such as radio management, which must be completed by highly constrained deadline. While Ref. [7] focused on a hardware and software development environment, the present paper focuses on an operating system that supports hard real-time tasks.

TinyOS [8] is a standard operating system for wireless sensor nodes. TinyOS takes up only 47 bytes of RAM and 473 bytes of ROM and switches tasks in several dozens of cycles. This excellent compactness is provided by the event-based architecture of TinyOS. In the event-based architecture, only one main loop executes event handlers according to received events, and the handlers never preempt each other.

However, the event-based architecture causes difficulties in performing hard real-time tasks and high programming complexity. In a real-time system, a higher priority task must preempt other tasks, but the event architecture forbids preemption for its compactness and small overheads.

In the present paper, we show a compact hard real-time operating system called PAVNET OS. To realize the hard real-time feature, PAVNET OS is designed with a thread model and enabling preemption. The enabling preemption causes two problems. First, the preemption induces huge overheads for checking task priorities and saving CPU context. Second, the preemption induces a conflict management problem among tasks.

To reduce the preemption overheads, PAVNET OS uses a characteristic of wireless sensor nodes: tasks can be categorized as real-time tasks or best-effort tasks. PAVNET OS provides two kinds of multithreading, which are preemptive multithreading and cooperative one. The preemptive multithreading is optimized for the real-time tasks with a CPU specific design, and the cooperative multithreading is optimized for the best-effort tasks. To mitigate the conflict management problem, PAVNET OS uses another characteristic of wireless sensor nodes: most conflicts occur between communication layers. PAVNET OS provides a wireless communication stack for hiding the exclusive controls to users.

The hard real-time feature can perform 100 Hz sensor sampling while performing radio management tasks with 0.01% jitter, whereas optimized TinyOS has 0.62% jitter. Additionally, PAVNET OS realizes compactness and low overheads that are comparable to those of TinyOS. For example, PAVNET OS can implement Blink, which is a sample program in TinyOS [9], on 63 bytes of RAM and 1,183 bytes of ROM, whereas TinyOS implements Blink on 44 bytes of RAM and 1,428 bytes of ROM. PAVNET OS also can switch tasks in 23 cycles minimally.

The present study is not intended to show that PAVNET OS is the best operating system. In fact, in contrast to highly portable TinyOS, PAVNET OS sacrifices portability because PAVNET OS has a design specific to Microchip PIC18. Lack of portability is a significant problem. However, the results of the present study imply that a better CPU design and operating system design may exist for future wireless sensor networks.

The remainder of the present paper is organized as follows. In the following section, we present the motivation for this re-
search and discuss the difference between the event model and the thread model. In Section 3, we provide the implementation details for the PAVENET OS in three parts: a hard real-time task scheduler, a best-effort task scheduler, and a wireless communication stack. Section 4 presents an evaluation of the performance of PAVENET OS. Section 5 discusses the relationship between priority levels and task switch overheads. Section 6 reviews related research, and conclusions are presented in Section 7.

2. Requirements

Some applications in wireless sensor networks need to obtain data of sufficient quality to have real scientific value, and the applications include earthquake monitoring [10], volcano monitoring [3], and structural health monitoring [4]–[6]. The applications require high fidelity sampling. For example, earthquake monitoring requires precise time-synchronized 100 Hz sampling, and tasks are periodically executed with strict deadlines [4], [10]. In addition, based on the success of TinyOS which is an event driven operating system, we know that compactness is an important factor when covering wide-area applications for wireless sensor networks because compactness is strongly related to power consumption over the entire sensor network.

The advantages of using either the event model or the thread model have been discussed thoroughly [8], [11]–[14]. It is difficult to strictly categorize all operating systems as event models or thread models, and there are many variations in programming style among models. To simplify the discussion herein, we define an event model in the manner of TinyOS [8], [15] and a thread model as traditional time-sliced multithreading, such as the POSIX thread. The event model has only one execution stream and forbids preemption among tasks: an event loop waits for events, an event invokes a handler, and the event handler is executed in run-to-completion. The thread model has multiple independent execution streams, shared states, preemptive scheduling, and synchronization schemes such as locks and conditions.

2.1 Event Model

In wireless sensor network research, a number of operating systems have been implemented with the event model, including TinyOS [8], SOS [16], Contiki [14], and protothreads [17]. The event-based architecture has two advantages. First, the user need not be concerned with conflict management because all event handlers execute in a run-to-completion manner and do not preempt each other. This feature also reduces context switch overheads because all task switches are realized by function call. Second, event models can be implemented using limited resources because of their simple structure, which consists of a memory stack, an event loop, and event handlers. This simplicity also allows portability of the system.

However, this simplicity causes two problems. First, the event model cannot perform hard real-time tasks. To support hard real-time tasks, the system must allow preemption. However, the event model does not allow preemption because the advantages are strongly related to the absence of preemption. For example, earthquake monitoring requires radio physical layer tasks and exact 100 Hz sensor sampling [4], [10]. The radio physical layer task has a 26 μs deadline and cycle, and a 12.5 μs computation time. The precise 100 Hz sensor-sampling task has a 10-ms cycle, a 2.2 μs computation time, and a 3.2 μs deadline. While TinyOS is performing a radio physical layer task, the sampling task cannot be executed until the radio physical layer task is finished. In fact, Kim et al. [6] struggled with temporal jitter caused by logging interferences in sampling. They succeeded to reduce the jitter with MicroTimer and turning off all unnecessary components on TinyOS. We note that the MicroTimer breaks the simplicity of the event model because the MicroTimer is implemented inside an interrupt handler. The implementation causes resource conflict problems.

Second, the event model has high programming complexity because the event model has to divide a sequence of tasks into multiple event handlers. With the words of Dunkels et al. [17]: “an event-driven model does not support a blocking wait abstraction. Therefore, programmers of such systems frequently need to use state machines to implement control flow for high-level logic that cannot be expressed as a single event handler.”

To reduce the programming complexity, Dunkels et al. [17] proposed a programming abstraction for the event model called protothreads. Protothreads makes it possible to write an event model in a thread-like style. However, protothreads still does not support hard real-time tasks.

2.2 Thread Model

The thread model can support hard real-time tasks because it allows preemption. Allowing preemption is not a sufficient condition, but a necessary condition, to support hard real-time tasks. For example, MANTIS is a time-sliced multithreading operating system for wireless sensor networks, but does not support hard real-time tasks [18]. In the thread model, the user can also understand the control flow easily because he/she can code tasks as if they dominate a CPU.

However, in contrast to the event model, the thread model does not have a simple structure, and the user must consider conflict management with shared data, and the task switch overheads are high because the thread model operating system has to save the CPU context at every preemption point. In addition, a memory stack is required for each execution stream. These features increase the memory consumption of operating systems. For example, MANTIS occupies less than 500 bytes of RAM and approximately 14 KB of ROM [18]. This is natural because the thread model provides an intermediate layer between the hardware and the software, whereas the event model is placed directly on the hardware.

We summarize the discussion about the event model and the thread model in Table 1. Both models have advantages and disadvantages. The event model is compact, low overheads, and need not manage resource contention. However, the event model cannot handle hard real-time tasks and has high programming complexity. The thread model can support hard real-time tasks, and has lower programming complexity. However,
the thread model is not compact, has high overheads, and need manage resource conliction among tasks. The challenge is to develop an operating system that has the following features: hard real-time support, compactness, low overheads, and low programming complexity.

3. PAVENET OS

We design a compact hard real-time operating system called PAVENET OS with enabling preemption. As mentioned in Section 2.2, the preemption induces preemption overheads and a conlict management problem. To tackle the problems, PAVENET OS provides three functions: a hard real-time task scheduler, a best-effort task scheduler, and a wireless communication stack. The hard real-time task scheduler and the best-effort task scheduler reduce task switch overheads with task specific design. The wireless communication stack mitigates the conflict management problem.

3.1 Hard Real-Time Task Scheduler

PAVENET OS provides a hard real-time task scheduler for real-time tasks, and the real-time tasks have a task priority and preempt lower priority tasks. The real-time tasks include radio management, sensor sampling, and media access control. Although the task priority and the preemption causes task scheduling/switching overheads, PAVENET OS performs real-time tasks in low overheads because PAVENET OS aggressively uses functions on PIC18, namely, dynamic priority levels and a fast return stack. PIC18 is a microcontroller developed by Microchip and has several interrupt sources, e.g., timers, external ports, a Master Synchronous Serial Port (MSSP), and a Universal Synchronous Receiver Transmitter (USART). Each source is dynamically assigned to a high priority level or a low priority level. High-priority interrupt events can interrupt any low-priority tasks and best-effort tasks. Low-priority interrupt events can interrupt any best-effort tasks. PIC18 also has a fast return stack, which automatically saves the CPU context when an interrupt occurs. In control registers, each interrupt has three bits to control their operation: a flag bit, an enable bit, and a priority bit. PIC18 has a high-priority vector at ROM address 0008h and a low-priority vector at ROM address 0018h, and they are coded as:

0008h: call isr_high
000Ah: nop
0018h: call isr_low
001Ah: nop

The isr_high() is coded as:

```c
void isr_high(void)
{
    if(TMR0IP && TMR0IE && TMR0IF){
        TMR0IF = 0;
        task_timer0();
    }
    if(TMR1IP && TMR1IE && TMR1IF){
        TMR1IF = 0;
        task_timer1();
    }
    if(INT1IP && INT1IE && INT1IF){
        INT1IF = 0;
        task_int1();
    }
}
```

The isr_low() is coded as:

```c
void isr_low(void)
{
    if((TMR0IP == 0) && TMR0IE && TMR0IF){
        TMR0IF = 0;
        task_timer0();
    }
    if((TMR1IP == 0) && TMR1IE && TMR1IF){
        TMR1IF = 0;
        task_timer1();
    }
    if((INT1IP == 0) && INT1IE && INT1IF){
        INT1IF = 0;
        task_int1();
    }
}
```

As shown above, PAVENET OS allows the user to place multiple tasks in the same priority level, and each task is executed as a run-to-completion thread.

In PAVENET OS, each real-time task corresponds to each interrupt vector. Therefore, there are no software transaction in task switching and task scheduling. PAVENET OS decides priority of real-time tasks according to their deadlines, and multiple tasks can have same priority: the low priority tasks must
not have smaller deadline than the high priority tasks. The real-time task scheduling is categorized into a class of deadline-monotonic scheduling [19],[20]. Deadline-monotonic scheduling can assign optimized priority to guarantee a deadline in a single CPU [19].

We can test the sufficient condition of the schedulability with deadline monotonic scheduling [19],[20]. The following is a schedulability test presented in [20].

All tasks are characterized by

\[ C_i \leq D_i \leq T_i \]

where \( C_i \) is the computation time, \( D_i \) is the deadline, and \( T_i \) is the period of task \( \tau_i \). In addition, task \( \tau_1 \) represents the highest priority task and \( \tau_n \), the lowest priority task, respectively. Then, schedulability test is given by:

\[ \forall i : 1 \leq i \leq n : \frac{C_i}{D_i} + \frac{I_i}{D_i} \leq 1 \]  \hspace{1cm} (1)

where \( I_i \) is a measure of higher priority tasks interfering with the execution of \( \tau_i \):

\[ I_i = \sum_{j=1}^{i-1} \left( \frac{D_j}{T_j} \right) C_j, \]  \hspace{1cm} (2)

If a task \( \tau_i \) satisfies equation (1), the task \( \tau_i \) is schedulable.

In Equations (1) and (2), the scheduler has \( n \) priority levels and each priority corresponds to a task. However, PAVENET OS has only two priority levels and can assign multiple tasks to a priority level. Therefore, the schedulability test for PAVENET OS is as follows.

Suppose there are \( n \) high-priority tasks. The schedulability test for high-priority tasks is given by:

\[ \frac{C_i}{D_i} + \frac{I_i}{D_i} \leq 1 \]  \hspace{1cm} (3)

where \( I_i \) is a measure of tasks having the same priority interfering with the execution of \( \tau_i \):

\[ I_i = \left( \sum_{j=1}^{n} \frac{D_j}{T_j} \right) C_j - \left( \frac{D_i}{T_i} \right) C_i = \left( \sum_{j=1}^{n} \frac{D_j}{T_j} \right) C_j - C_i. \]

Therefore, Equation (3) is:

\[ \frac{C_i}{D_i} + \frac{I_i}{D_i} = \frac{C_i}{D_i} + \left( \sum_{j=1}^{n} \frac{D_j}{T_j} \right) C_j - \left( \frac{D_i}{T_i} \right) C_i \]

\[ = \sum_{j=1}^{n} \frac{D_j}{T_j} C_j - D_i C_i \leq 1. \]  \hspace{1cm} (4)

If a task \( \tau_i \) satisfies equation (4), the task \( \tau_i \) is schedulable.

Next, we show the schedulability test for low-priority tasks \( \tau_k \). When there are \( n \) high-priority tasks and \( m \) low-priority tasks, the schedulability test is given by:

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
name     & size & meaning   \\
\hline
\text{tid} & 8 bit & thread ID   \\
\text{state} & 8 bit & thread state \\
\text{pc} & 16 bit & program counter \\
\text{sleep_time} & 8 bit & time to wake \\
\hline
\end{tabular}
\caption{Table 2 Task control block. A task control block in PAVENET OS uses only 40 bits per thread.}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\text{function name} & transaction & setting state   \\
\hline
\text{add_task(funcname)} & add task to scheduler & execute   \\
\text{os_yield()} & yield control & execute \\
\text{sleep(time)} & sleep time sec & sleep   \\
\text{sig_wait()} & wait signal & wait   \\
\text{suspend_task(pid)} & let pid to wait & execute \\
\text{signal_task(pid)} & let pid to execute & execute \\
\text{kill_task(pid)} & let pid to be dead & execute \\
\hline
\end{tabular}
\caption{Table 3 Task control functions. PAVENET OS provides seven system calls for task management.}
\end{table}

3.2 Best-Effort Task Scheduler

PAVENET OS performs hard real-time tasks with preemption, as described in Section 3.1, and other tasks are performed with a best-effort task scheduler. The best-effort tasks include hop-by-hop routing, delay writing to flash memory, and replying to sensor data query.

We develop the best-effort task scheduler with the thread model and add three limitations for compactness and low overheads to the threads. First, these threads only switch context cooperatively. Cooperative task switching eliminates the need for conflict management and only preserves a program counter as CPU context. Second, these threads can only yield the top level of a function. Although we can yield anywhere in the function if the scheduler preserves the entire call stack, we do not use this approach because it consumes a great deal of memory and computation time. Third, these threads do not use stack memory. Because of this limitation, CPUs need not have a stack memory. Although PIC18 has a call stack, it does not have a stack memory, and a heap memory is assigned to variables at compilation time. Because of these limitations, these threads forbid reentrance and duplication.

Table 2 represents a task control block on the best-effort task scheduler. The block, \text{tid} is the thread identifier, which the scheduler allocates to a thread when the thread is created. The
management in pbuf. The APIs for pbuf are: provides APIs, which hides exclusive controls for the buffer. Each layer only needs to copy small identifiers. Pbuf also realizes modularity at each communication layer, and the user need not consider conflict management. The wireless communication stack, and the scheduler increments jiffies every 100 ms.

Table 3 lists the task management functions. Since PA VENET OS ticks jiffies, and the scheduler increments jiffies every 100 ms.

3.3 Wireless Communication Stack

Preemption caused by hard real-time tasks causes a conflict management problem. To reduce user fatigue caused by conflict management, we use a characteristic of wireless sensor nodes: most conflicts occur between communication layers. For example, when a physical layer receives a packet, the physical layer accesses a receive buffer in a media access control (MAC) layer, and the MAC layer also accesses the receive buffer to run a MAC protocol. PAVENET OS hides these exclusive controls in the wireless communication stack, and the user need not consider conflict management. The wireless communication stack also realizes modularity at each communication layer, and the user can easily develop various communication protocols according to application demands.

Figure 1 shows the wireless communication stack, including a physical layer, a MAC layer, a network layer, a socket layer, and an application layer. To exchange data among layers efficiently, PAVENET OS provides a buffer management mechanism called pbuf, which is a lightweight version of BSD mbuf [21]. Since the pbuf assigns a small identifier to a buffer, each layer only needs to copy small identifiers. Pbuf also provides APIs, which hides exclusive controls for the buffer management in pbuf. The APIs for pbuf are:

```c
uint8 get_new_pbuf(void);
byte *get_pbuf_next(uint8 index, uint8 size);
byte *get_pbuf_head(uint8 index);
uint8 release_pbuf(uint8 index);
```

3.4 Operation Example

In this section, we show an example operation of hard real-time tasks, best-effort tasks, and the wireless communication stack. In this example, a physical layer task and a MAC layer task are implemented as hard real-time tasks, and a network layer task is implemented as a best-effort task on a PAVENET module which is shown in Fig. 2. The physical layer task is invoked by an external interrupt which is associated with a data clock port on CC1000. When CC1000 is running at 19.2 kbps, the period of interrupt is about 52 μs, which corresponds to 1 bit reception. When the physical layer task finishes to receive 1 packet of bits, the packet is inserted to a MAC layer receive queue with up2mac() by the physical layer task. If the MAC layer task is 10 ms slotted TDMA, the MAC layer task is executed at 10 ms intervals by timer0. The MAC layer task controls physical layer’s sending and receiving state, and inserts a packet to network layer’s receive queue with up2net() if the MAC layer’s receive queue has a packet. The network layer task is implemented as a best-effort task, and routinely checks a receive queue with sleep(). If the receive queue has a packet, the network layer task processes the packet: routing, passing to an application layer, and so on.

Each hard real-time task can be characterized with its computation time C, its deadline D, and its period T. If all hard real-time tasks satisfy equation (4) and (5), PAVENET OS guarantees the execution of the hard real-time tasks even when there is frequent packet reception. However, the network layer task, which is a best-effort task, might fail to process some packets when there is frequent packet reception. The fail induces packet loss. In above example, when the MAC layer task calls up2net(), the packet is deleted if the network layer’s receive queue is full.

4. Evaluation

To evaluate PAVENET OS, we compare the precision of the hard real-time task scheduler, the compactness, the execution overheads, and the programming complexity to those of TinyOS 1.10 running on MICA2 [22]. PAVENET OS is implemented with the HI-TECH PICC-18 compiler and runs on PAVENET modules.

Table 4 lists the specifications of PAVENET modules and MICA2, and PAVENET modules are shown in Fig. 2. PAVENET modules and MICA2 have the same level of equipment. PAVENET modules have PIC18LF4620 as a CPU and TI CC1000 as a radio module. The operating frequency of the CPU is 20 MHz, but the number of instructions-per-second is 5 MIPS because PIC18LF4620 performs an instruction per four
Table 4  Evaluated sensor nodes.

<table>
<thead>
<tr>
<th></th>
<th>PAVENET module</th>
<th>MICA2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>PIC18LF4620</td>
<td>ATmega128L</td>
</tr>
<tr>
<td>frequency</td>
<td>7.4 MHz</td>
<td>20 MHz</td>
</tr>
<tr>
<td>instruction per second</td>
<td>7.4 MIPS</td>
<td>5 MIPS</td>
</tr>
<tr>
<td>wireless module</td>
<td>CC1000</td>
<td>CC1000</td>
</tr>
<tr>
<td>wireless frequency</td>
<td>315 MHz</td>
<td>315 MHz</td>
</tr>
<tr>
<td>wireless modulation</td>
<td>FSK</td>
<td>FSK</td>
</tr>
<tr>
<td>power voltage</td>
<td>DC3V</td>
<td>DC3V</td>
</tr>
<tr>
<td>current (receiving)</td>
<td>30 mA</td>
<td>30 mA</td>
</tr>
<tr>
<td>current (sleep)</td>
<td>30μA</td>
<td>30μA</td>
</tr>
<tr>
<td>communication rate</td>
<td>38.4 kbps</td>
<td>19.2 kbps</td>
</tr>
</tbody>
</table>

Table 5  Hard real-time performance.

<table>
<thead>
<tr>
<th></th>
<th>PAVENET OS</th>
<th>TinyOS (default)</th>
<th>TinyOS (optimized)</th>
<th>nano-RK</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>max</td>
<td>min</td>
<td>jitter</td>
<td>max</td>
</tr>
<tr>
<td>Sampling</td>
<td>10.001 ms</td>
<td>10.000 ms</td>
<td>0.001 ms</td>
<td>9.764 ms</td>
</tr>
<tr>
<td>Sampling + RF</td>
<td>10.001 ms</td>
<td>10.000 ms</td>
<td>0.001 ms</td>
<td>10.003 ms</td>
</tr>
</tbody>
</table>

clock cycles. The wireless communication speed of PAVENET modules is 38.4 kbps, but we changed it to 19.2 kbps in order to allow fair comparison with MICA2. MICA2 has Atmel ATmega128 as a CPU and CC1000 as a radio module. The operating frequency of the CPU is 7.4 MHz, and the number of instructions per second is 7.4 MIPS because ATmega128 performs one instruction per one clock cycle. Although ATmega128 and PIC18LF4620 have different architectures, they target the same application area. We note that TinyOS can port to PAVENET modules, but PAVENET OS cannot port to MICA2 because of its CPU-specific architecture.

4.1 Hard Real-Time Tasks

To evaluate hard real-time tasks, we assume tasks in earthquake monitoring [4],[10],[23] as an actual application for wireless sensor networks. Earthquake monitoring requires precise 100 Hz sampling with radio communication because each sampling must be synchronized among sensor nodes. The evaluation results of sampling jitter can adapt to other frequencies because sampling jitter is independent of sampling frequency. The sampling jitter is calculated with maximum and minimum intervals of the sampling. The fluctuation of the intervals is caused by the fluctuation of task switch overheads and the critical section period of other tasks. The distribution of the intervals depends on sampling rate. However, the sampling jitter does not depend on sampling rate because the maximum and minimum intervals can be derived from the maximum fluctuation of task switch overheads and the maximum critical section period of other tasks, which are independent of sampling frequency.

Table 6  Kernel footprint.

<table>
<thead>
<tr>
<th>module</th>
<th>RAM (byte)</th>
<th>ROM (byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>task scheduler</td>
<td>29</td>
<td>490</td>
</tr>
<tr>
<td>wireless communication stack</td>
<td>628</td>
<td>930</td>
</tr>
<tr>
<td>total</td>
<td>657</td>
<td>1,420</td>
</tr>
</tbody>
</table>

Table 5 shows evaluation results. ‘Sampling’ is the sensor node performing only a 100 Hz sampling task, and ‘Sampling + RF’ is the sensor node performing a 100 Hz sampling task while receiving packets from another node, which sends a packet every 50 ms. All of the packet loss rates were 0%. ‘TinyOS (default)’ uses the Timer component [24] for sampling, and ‘TinyOS (optimized)’ uses the MicroTimer. The optimized TinyOS assumes a same setting with [6]. As mentioned in Section 2.1, the use of the MicroTimer breaks simplicity of the event model. The ‘nano-RK’ runs on MICAz because nano-RK does not support MICA2. To sample precisely at 100 Hz, the sensor node must generate precise 10 ms intervals. We measured the intervals 2,000 times and obtained the maximum value, the minimum value, and the jitter. The results indicate that PAVENET OS realizes 100 Hz sampling much more precisely than TinyOS, as shown in Table 5. The default TinyOS cannot perform precise 100 Hz sampling, even if performing only the sampling task. The Timer component on TinyOS adjusts the timer firing timing between 9-10 ms. Therefore, when the default TinyOS samples with radio communication, the sampling error becomes significant because the adjustment is tumbled by the radio communication tasks. The optimized TinyOS can perform precise 100 Hz sampling with 0.003 ms or 0.03% jitter, if performing only the sampling task. The jitter becomes significant at 0.062 ms or 0.62%, when the optimized TinyOS performs the sampling task with radio communication. On the other hand, PAVENET OS can always sample precisely at 100 Hz, even with radio communication, and the jitter is much smaller at 0.001 ms or 0.01%.

The results also indicate that nano-RK is not appropriate for precise 100 Hz sampling. The nano-RK has larger jitter than ‘TinyOS (default)’ even if performing only the sampling task. The huge jitter is caused by software implementation of a hard real-time scheduler. The software implementation induces scheduling and task switch overheads. Additionally, when nano-RK samples with radio communication, the sampling jitter becomes significant because the radio physical layer task in nano-RK’s wireless protocol stack heavily uses critical sections.

As described in [25], the sampling jitter induces a sensing error. The requirement for the sensing error reduction is different among sensor network applications. For example, earthquake monitoring, which is our main application target, needs less than 1 ms jitter [23]. ‘TinyOS (default)’ and ‘nano-RK’ do not satisfy 1 ms jitter, but ‘PAVENET OS’ and ‘TinyOS (optimized)’ satisfy 1 ms jitter. In view of other applications, lower sampling jitter indicates a wider application area.

4.2 Compactness

We show that PAVENET OS has compactness comparable to that of TinyOS. To evaluate the compactness, we measure the RAM sizes and ROM sizes of PAVENET OS and TinyOS. First, we measure the footprint of the scheduler and the wireless communication stack, as shown in Table 6. The task sched-
Table 7 Footprint size on the sample applications.

<table>
<thead>
<tr>
<th></th>
<th>PAVENET OS</th>
<th>TinyOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blink (byte)</td>
<td>63</td>
<td>1,183</td>
</tr>
<tr>
<td>BlinkTask (byte)</td>
<td>64</td>
<td>1,271</td>
</tr>
<tr>
<td>CntToLeds (byte)</td>
<td>64</td>
<td>1,209</td>
</tr>
<tr>
<td>CntToRfm (byte)</td>
<td>676</td>
<td>11,336</td>
</tr>
<tr>
<td>CntToLedsAndRfm (byte)</td>
<td>676</td>
<td>11,366</td>
</tr>
</tbody>
</table>

Table 8 Average execution cycles on sample applications not including radio communication.

<table>
<thead>
<tr>
<th></th>
<th>PAVENET OS</th>
<th>TinyOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blink (cycle)</td>
<td>8.5</td>
<td>19.5</td>
</tr>
<tr>
<td>BlinkTask (cycle)</td>
<td>134.5</td>
<td>123.5</td>
</tr>
<tr>
<td>CntToLeds (cycle)</td>
<td>147.0</td>
<td>155.5</td>
</tr>
</tbody>
</table>

Table 9 Average execution times on sample applications including radio communication.

<table>
<thead>
<tr>
<th></th>
<th>PAVENET OS</th>
<th>TinyOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CntToRfm (ms)</td>
<td>17.0</td>
<td>17.1</td>
</tr>
<tr>
<td>CntToLedsAndRfm (ms)</td>
<td>16.9</td>
<td>17.2</td>
</tr>
</tbody>
</table>

4.3 Overhead

Figure 3 shows the task switch overheads for various tasks, and the tasks are the net task and the user task as best-effort tasks, the mac task as a low-priority real-time task, and the phy task as a high-priority real-time task. The low-priority task can preempt best-effort tasks in 66 cycles. The high-priority task can preempt best-effort tasks or low-priority tasks in 23 cycles. When a best-effort task yields the CPU, the best-effort task switches to another best-effort task in 92 cycles. Task switch overheads are relatively low. For example, MANTIS switches tasks in approximately 400 cycles [18], nano-RK, in approximately 333 cycles (45 μs on FireFly) [26], and TinyOS, in 51 cycles [8], respectively.

PAVENET OS has low overheads, on the same level as TinyOS, when implementing the same sample applications provided by TinyOS. These applications are described in Section 4.2. First, we measured the average execution cycle from the timer being fired until the end of the sequence of tasks on Blink, BlinkTask, and CntToLeds, as shown in Table 8. The average cycle was calculated from 100 execution cycles. We use “cycle” as the unit because PAVENET modules and MICA2 have different clock frequencies. In Blink and CntToLeds, PAVENET OS is slightly faster than TinyOS because TinyOS has small overheads at joints between modules. In BlinkTask, PAVENET OS is slightly slower than TinyOS and the result corresponds to the task switch overheads.

Second, we measured the average execution time from the timer being fired until the end of packet transmission on CntToRfm and CntToLedsAndRfm, as shown in Table 9. We use time, rather than cycles, as the unit because the execution time is strongly related to the packet length and communication overheads. Although PAVENET OS has the wireless communication stack and PAVENET modules run slower MIPS than TinyOS, the results are almost the same. The results represent the wireless communication stack on PAVENET OS has relatively low overheads.
4.4 Programming Complexity

To evaluate the programming complexity of PAVENET OS, we count the lines in sample application source codes as compared to TinyOS. We excluded blank lines and comment lines from the count. We also excluded system codes such as a scheduler and timers. For example, we did not count the Main component and the Timer component in TinyOS. It is difficult to evaluate programming complexity simply by counting the number of lines of code. TinyOS reduces programming complexity with the component-oriented application design of nesC [15] and provides a rich collection of software components. However, we do not have a perfect method to evaluate programming complexity [27]–[33]. We believe the number of lines represents one aspect of programming complexity. At least, a small number of lines make it easier for the user to understand what is going on in the source code.

PAVENET OS can implement the sample applications in shorter code than TinyOS, as shown in Table 10, because PAVENET OS can implement a sequence of tasks into one task. The code size difference tends to be significant when the number of components increases in TinyOS, because nesC, which TinyOS uses, needs to implement a component with multiple event handlers.

The number of users can also be a criteria for programming complexity. Some research projects, including sensor-actuator cooperation services [34], [35], earthquake monitoring [10], [36], context-aware services [37], [38], battery-less sensor networks [39], and a collision detection mechanism [40], have used PAVENET OS. At present, the PAVENET OS community is very small compared to the TinyOS community.

5. Discussion

Although traditional hard real-time operating systems, such as nano-RK, support many priority levels, PAVENET OS only supports two priority levels. In this section, we discuss the relationship between priority levels and task switch overheads.

Sensor nodes have two categories of hard real-time tasks. The first category consists of the tasks whose deadlines are almost equal to the computation time of the tasks (\(D \approx C\)). The \(D \approx C\) tasks have to be executed just after task requests issued. The examples of the tasks are a sensor sampling task and a TDMA timing control task. The execution of a \(D \approx C\) task is guaranteed only if the task has the highest priority. If there are two or more \(D \approx C\) tasks, the operating system cannot guarantee their deadlines due to equations (4) and (5). The increase of priority levels cannot solve the problem since equations (1) and (2) should hold. The only solution is an exclusive execution of the \(D \approx C\) tasks.

The second category consists of the tasks whose deadlines are almost equal to the period of the tasks (\(D \approx T\)). The \(D \approx T\) tasks have to finish until next task request is coming. The examples of the \(D \approx T\) tasks are a radio physical layer task, a UART task, and sensor data processing tasks. PAVENET OS can guarantee the deadline of these tasks if the tasks satisfy equations (4) and (5). The increase of priority levels might improve processor usability if task switch overheads are small.

The two priority levels are sufficient for our main application target which is earthquake monitoring [23]. Increase of priority levels improves processor usability. However, practically, increase of priority levels does not improve processor usability, if task switch overheads are large [41]. For example, a 40 \(\mu\)s period and 40 \(\mu\)s deadline task cannot be guaranteed by nano-RK [26] which has 45 \(\mu\)s task switch overheads and 64 priority levels, and can be guaranteed by PAVENET OS which has 4.6 \(\mu\)s task switch overheads and only two priority levels. If there is new CPU which supports more dynamic priority levels than PIC18, we should realize many priority levels because they together with small task switch overheads improve processor usability.

6. Related Research

There are a number of operating systems for wireless sensor nodes, including TinyOS [8], [15], SOS [16], Contiki [14], nano-RK [26], MANTIS [18], protothreads [17], and t-kernel [42]. Most of these operating systems are designed with the event model. As mentioned in Section 2, the event model cannot support hard real-time tasks, and has high programming complexity.

TinyOS [8] is the de facto standard operating system for wireless sensor nodes. TinyOS was designed with the event model, and so does not support hard real-time tasks and has high programming complexity. TinyOS attempts to reduce the programming complexity through a new event-driven specific language called nesC [15], which enhances reusability of components. However, this solution means that the user has to learn a new language.

SOS [16] is another event model operating system. SOS has a loadable programming module feature, whereas TinyOS has a statically linked system image. The loadable module is lightweight and can be written in C. In SOS, the user can update modules after the sensor nodes are deployed. However, SOS cannot support hard real-time tasks and has high programming complexity.

Contiki [14] is an event model operating system. To reduce the programming complexity, as an option, Contiki can support time-sliced preemptive multitasking by assigning a memory stack to each thread. The memory assignment consumes computational resources. In addition, the threads destroy the simplicity of event models, e.g., the user must manage resource confliction. Moreover, Contiki cannot support hard real-time tasks even if the user uses the threads.

Protothreads [17] have an implementation similar to that of the cooperative task scheduler of PAVENET OS. Protothreads is an extension of the event model designed to reduce the programming complexity. The event model must divide a task into multiple run-to-completion functions. Protothreads provide a conditional blocking wait statement to the event model. The user can then write a program in a thread-like style. When using the conditional blocking wait, the user inserts PT_BEGIN and PT_END at the top and the bottom respectively, of the event handler. Protothreads reduce the programming complexity of the event model, but does not solve all of the problems in the
event model. In particular, protothreads cannot support hard real-time tasks.

The nano-RK [26] is the most closely related work to PAVENET OS. It is a preemptive multitask operating system supporting real-time tasks. Additionally, nano-RK is more portable than PAVENET OS. However, nano-RK has more context switch overheads than PAVENET OS because nano-RK has to preserve CPU context by software. Nano-RK needs several dozens of μs for task switching whereas PAVENET OS needs several μs.

Like PAVENET OS, MANTIS [18] is a thread model operating system. The difference between them is the implementation of the thread model. MANTIS uses time-sliced multitreading, whereas the threading of PAVENET OS is not time-sliced. To realize time-sliced multitreading, MANTIS assigns a stack memory for each task. Therefore, MANTIS consumes more RAM than PAVENET OS. Furthermore, MANTIS does not support hard real-time tasks.

T-kernel [42] is also a thread model operating system, and provides virtual memory and preemptive scheduling. Since the preemptive scheduling has 16 priority levels, the t-kernel might be able to support hard real-time tasks. However, it is not evaluated its schedulability, hard real-time performance, and overheads [42]. In addition, t-kernel does not provide any mechanism to hide exclusive controls like the wireless communication stack on PAVENET OS.

7. Conclusion

The present paper has described PAVENET OS, a compact hard real-time operating system for wireless sensor nodes. It can be implemented on the same amount of computational resources as TinyOS, and, unlike TinyOS, PAVENET OS supports hard real-time tasks and has low programming complexity. In addition, since a wireless communication stack is provided, the user need not consider the exclusive controls caused by hard real-time tasks. The results of the present study imply that hardware support by CPU can extend the functions of an operating system without a loss of compactness. For future wireless sensor nodes, it may be necessary to reconsider the balance between hardware and software.

The authors are currently working on the integration of a CPU design and an operating system design for ultra low-power wireless sensor networks [43]. Ekanayake et al. have already succeeded to implement an ultra low-power processor using software/hardware co-design based on the event model [44]. The authors believe that designing a CPU based on the thread model, such as a many-core design, also dramatically reduces energy consumption and covers a wider range of applications for future wireless sensor networks.

References


Shunsuke Saruwatari

He received the Dr. Sci. degree from the University of Tokyo in 2007. Since 2008, he has been in the University of Tokyo and currently a research associate of the Research Center for Advanced Science and Technology. His research interests are in the areas of wireless networks, sensor networks, and system software.

Makoto Suzuki

He received his B.S., M.S., and Ph.D. degrees from the University of Tokyo, Japan, in 2005, 2007, and 2010, respectively. In 2010, he joined the faculty of the University of Tokyo, where he is currently a Project Research Associate of RCAST. His research interests include wireless sensor networks, distributed system, and time synchronization.

Hiroyuki Morikawa

He received the B.E., M.E. and Dr. Eng. degrees in electrical engineering from the University of Tokyo, in 1987, 1989, and 1992, respectively. Since 1992, he has been in the University of Tokyo and currently a full professor of RCAST. His research interests are in the areas of computer networks, ubiquitous networks, mobile computing, wireless networks, photonic internet, and network services.