Design and Analysis of Almost-Always-Sleeping Schedulers for Embedded Systems

Biswajit Mazumder, Hao Jiang, and Jason O. Hallstrom
School of Computing
Clemson University
Clemson, SC 29634, USA
{bmazumd, hjiang, jasonoh}@cs.clemson.edu

Abstract—Limited energy resources dictate the design of many embedded applications composed of small, modular tasks, scheduled periodically. In this paper, we describe a series of task schedulers for AAS systems designed to maximize sleep time. We consider four scheduler designs, model their performance, and present detailed performance analysis results under varying load conditions. This is the first systematic analysis of this important class of schedulers.

Keywords—Wireless sensor networks; scheduling; power consumption.

I. INTRODUCTION

A significant class of embedded applications are characterized by low duty-cycle operation and time-triggered, periodic execution. These systems sleep for relatively long periods, wake in response to a timer interrupt, perform a short computation, and return to sleep. We refer to these systems as almost-always-sleeping (AAS) systems. The wireless sensor network domain is rife with representative examples. Environmental monitoring networks [12], [17], [18], for instance, comprise distributed sensors that periodically wake to collect and transmit environmental stimuli before returning to sleep. Indeed, nearly every sensing system adopts a variant of this strategy, as do numerous other embedded applications.

The broad adoption of AAS designs is due to the energy efficiency they afford. Modern microcontrollers (MCUs) support sleep states in which internal circuitry may be powered-down, reducing energy consumption by several orders of magnitude. As an example, common wireless sensor networking platforms consume tens of milliwatts in the active state, and only tens of microwatts when idle [13]. For devices that exhibit this two-phase consumption profile, the best conservation strategy is to sleep as often as possible.

The active period of an embedded device is partitioned into two components: the time spent executing application code (tasks), and the time spent executing scheduling code. Reducing the runtime of individual tasks can only be achieved on an application-by-application basis. Reducing the scheduling overhead, however, can be achieved through careful analysis and design of the underlying scheduling system — our focus.

Contributions. In this paper, we detail the design and implementation of four progressively more efficient scheduling systems designed to support AAS embedded applications. The designs are applicable to virtually any modern MCU. For the sake of presentation, we focus on the popular ATmega family of devices, which are used in a number of sensor networking platforms [14]–[16]. For each scheduler implementation, we present a closed-form algebraic model that captures the scheduling overhead as a function of task load and other parameters. These models are used to characterize the comparative performance among the designs. To supplement this analysis, we also conduct physical power profiling studies using an ATmega644-based sensor networking platform. The results provide a clear picture of the power consumption profile associated with each design, as well as the comparative lifetime benefits they provide.

We emphasize that these designs are practically motivated. They evolved over the course of 18 months while developing a large-scale environmental monitoring network deployed in the city of Aiken, South Carolina [8]. In 2011, the city’s stormwater treatment system was redesigned to reduce the environmental impacts associated with stormwater runoff. The monitoring network was installed in targeted areas throughout the city to monitor the modified treatment system. Our subteam was responsible for the design of the wireless sensor platforms and the associated firmware used to construct the network. The design process was guided by the need to support continuous, uninterrupted data collection in the face of unattended operation (since Aiken is relatively remote). Maximizing the lifetime of our almost-always-sleeping system was a principal goal. In addition to yielding a successful network deployment, the experience resulted in the first systematic analysis of AAS schedulers, which we present here.

In Section II we provide a formal definition for the scheduling problem in AAS systems and present the related work in Section III. In Sections IV and V we present the designs and the corresponding algebraic models for the schedulers, respectively. The comparative analysis and experimental results are presented in Section VI. Section VII concludes the paper.

II. PROBLEM STATEMENT

The smallest unit of work that may be scheduled in an AAS system is a task, an action taken in response to a timer event. When a scheduler wakes and has no tasks to execute, a small amount of time is expended, referred to as the null activation period, denoted by \( A_1 \). The amount of time expended when the scheduler wakes and there are tasks to execute, including
task execution time, is referred to as the task activation period, denoted by $A_{T\text{ASK}}$.

In a given time period $N$, a scheduler experiences $A_1$ and $A_{T\text{ASK}}$ multiple times and sleeps the rest of the time. The number of times the scheduler experiences $A_1$ and $A_{T\text{ASK}}$ in a time period $N$ is given by $n_1$ and $n_2$, respectively. Each instance, $i$, of $A_{T\text{ASK}}$ within $N$ consists of time spent executing the task functions, given by $\omega_i$, and the rest of the time expended prior to, in between, and after task execution, denoted by $A_2$. The relationship between $A_1$, $A_2$, $A_{T\text{ASK}}$, $\omega_i$, and $N$ is illustrated in Figure 1. The total time spent executing the task functions in the time period $N$ is given by $W$, calculated as the sum of all $\omega_i$, where $i = \{1, 2, ..., n_2\}$. In the $i^{th}$ occurrence of $A_{T\text{ASK}}$, $\omega_i$ is calculated as the sum of all $\omega_{i,j}$, where $j = \{1, 2, ..., n_{\text{executed}}\}$; $n_{\text{executed}}$ denotes the number of task functions executed in the $i^{th}$ task activation period. Assuming all tasks are periodic, and $n_{\text{executed}}$ is constant for all values of $i$, the total time taken to execute all task functions, $W$, in time period $N$ is calculated as:

$$W = \sum_{i=1}^{n_2} \sum_{j=1}^{n_{\text{executed}}} \omega_{i,j}$$

The scheduler load $\alpha$ is the fraction of time the system is either busy scheduling tasks or executing them within time period $N$. The task load $\beta$ is the fraction of time the system is busy executing just the task functions, given by $W$, within time period $N$. Assuming $n_1$, $n_2$, $W$, and $N$ are fixed, and $n_2A_{T\text{ASK}} = n_2A_2 + W$, $\alpha$ can be expressed as:

$$\alpha = \frac{n_1A_1 + n_2A_{T\text{ASK}}}{N}$$

$$= \frac{(n_1A_1 + n_2A_2 + W)}{N}$$

$$= \frac{n_1A_1 + n_2A_2}{N} + \beta$$

**Objective.** In an ideal scheduler, with no scheduling overhead, $\alpha = \beta$. To minimize the value of $\alpha$, both $A_1$ and $A_2$ need to be minimized. Our objective is to design a scheduler with the least possible $A_1$ value; since $n_1 \gg n_2$ in AAS systems, a lower $A_1$ value, even at the expense of a higher $A_2$ value, will help in maximizing the efficiency and battery life expectancy of a scheduler.

### III. Related Work

Levis et al. present TinyOS [10], one of the most widely-used sensor network operating systems. TinyOS includes a task scheduler that executes non-preemptive tasks posted for later execution. TinyOS uses a fixed-length, FIFO scheduler by default. To reduce energy consumption, the scheduler puts the processor to sleep whenever the task queue is empty. Its successor, TinyOS2 [11], uses a similar FIFO scheduler; an earliest-deadline-first implementation is also available. Compared to TinyOS, TinyOS2 introduces more overhead when posting and executing a task, but less overhead when the task queue is empty.

Han et al. present SOS [9], another event-driven operating system. Software modules communicate using direct calls and message passing via a FIFO scheduler with two levels of priority. High priority messages are reserved for time critical events, such as hardware interrupts.

Dunkels et al. present Contiki [7], another event-based operating system with support for event prioritization. A non-preemptive event scheduler schedules asynchronous and synchronous events. Asynchronous events are deferred procedure calls enqueued in a FIFO handling queue. Synchronous events are immediately scheduled at the front of the queue.

Bhatti et al. present MANTIS [2], a multi-threaded sensor network operating system. In MANTIS, a fixed thread table maintains all threads, which are executed using round-robin scheduling within priority levels. The scheduler is driven by a timer interrupt, which triggers context switching among threads. MANTIS also allows users to specify the sleep period of threads. The scheduler calculates the earliest wake-up time and uses an idle background thread to put the CPU to sleep when all other threads are blocked.

Chen et al. present Enix [6], a cooperative threading solution for sensor networks, which uses setjump and longjump to implement low overhead context switching. It supports priority-based and round-robin scheduling policies using linear search and bitmap-based thread lookups. Other multi-threaded sensor network operating systems, including LiteOS [3] and RETOS [5], use similar schedulers. In particular, LiteOS supports priority-based and round-robin scheduling policies, and RETOS supports POSIX scheduling, which boosts the priority of a thread when events need to be handled quickly.

While each has its advantages, none of these systems are well matched for AAS scheduling. Event-based schedulers using FIFO mechanisms or priorities are not designed to account for the sleep requirements of AAS systems. Thread-based schedulers are also inefficient in this context; POSIX-like solutions introduce significant overhead, while the use of small epochs in other multi-threaded solutions is energy-inefficient. By contrast, our work focuses on the systematic design and analysis of scheduling solutions suited specifically
to AAS systems.

Caracas et al. describe an energy efficient optimization strategy based on variable sleep intervals [4]. They define a knapsack problem to compute the minimum number of (pre-specified) sleep intervals required to achieve a given sleep period, but do not provide the solution details. We present the implementation details for a similar variable-sleep scheduling strategy, where we use a greedy solution to the knapsack problem and analyze its complexity and performance characteristics.

IV. AAS SCHEDULING

We focus on a canonical implementation of an AAS scheduler, where a task is composed of a function pointer, a task type, a period, and a due date. The function pointer points to the executable task body. The task type is either one_shot or periodic, corresponding to a task that expires after it has been executed, and a task that is continually rescheduled, respectively. The period specifies how often the task should be activated. The due date records the time at which the task should occur next.

The basic scheduling functions in our implementation are scheduler_init(), schedule_task(), and scheduler_run(). scheduler_init() handles scheduler initialization during system start-up, and schedule_task() is used to schedule new tasks. The system spends much of its lifetime in scheduler_run(); it contains the core of the scheduling logic and is invoked to start the scheduler.

The scheduler designs presented in the next sections depend on the hardware system, particularly the timer mechanism. The target MCU implements the system clock using an 8-bit counter register, driven by an external oscillator oscillating at a rate of 32,768KHz. A prescaler of 128 results in an overflow interrupt being triggered once per second; this suspends the executing instruction and begins the interrupt service routine (ISR), where the system time is updated. If the processor is in a sleep state, it wakes and enters the ISR. Upon completion, the processor resumes execution following the call to sleep.

A. A Basic Scheduler

We present a basic AAS scheduler implementation that parallels the design of existing embedded task schedulers [9]–[11]. system_task_buffer, an N-element array, is initialized (with NULL entries) within scheduler_init(). schedule_task() finds the first empty slot and stores the task passed as argument.

scheduler_run(), shown in Listing 1, iterates indefinitely in the outer while loop. In each iteration, referred to as an execution cycle, the scheduler steps through system_task_buffer and executes each task with an expired due date. When a one_shot task completes, the task is removed from system_task_buffer. When a periodic task completes, its due date is updated based on its period. When there are no tasks to execute, the scheduler enters its sleep cycle.

This simple scheduler has a significant power consumption footprint due to the time required to determine whether there are tasks to execute. Even when there are no tasks to execute, the scheduler wakes and cycles through the entire task buffer. Since the time expended is bounded by N, an increase in task capacity degrades system performance. A scheduler that could perform a constant-time lookup into the task array for available tasks would be more desirable.

Listing 1. scheduler_run() (Basic Scheduler)

```c
void scheduler_run() {    
    while(true) {        
        bool task_executed = false;        
        uint32_t current_time = current_system_time();        
        uint32_t task_index = 0;
        for(task_index = 0; task_index < TASKQUEUE_CAPACITY; task_index++) {            
            if(system_task_buffer[task_index].task != NULL) &
                (current_time >= system_task_buffer[task_index].due_date) {                
                // execute the task function                
                system_task_buffer[task_index].task(current_time);                
                // handle rescheduling / removal                
                if(system_task_buffer[task_index].type == ONE_SHOT) {                    
                    system_task_buffer[task_index].task = NULL;                    
                } else {                    
                    system_task_buffer[task_index].due_date += system_task_buffer[task_index].period;                    
                }                
                task_executed = true;                
            }        }
        }    
}    
```

Listing 2. scheduler_run() and run_task() (O(1) Scheduler)

B. The O(1) Scheduler

The O(1) scheduler is based loosely on the Linux 2.6.8.1 scheduler [1]. Adapted to our system, when there are no tasks in the queue, the scheduler performs a constant-time lookup and returns to sleep. This scheduler also uses system_task_buffer to store scheduled tasks. Two supporting queues are also introduced; the active task queue stores tasks which must be executed in the current execution cycle, and the idle task queue stores tasks that have been executed, but which must be re-evaluated the next time the system wakes. To achieve constant-time task lookup, the queues are implemented using bitmaps; a bit at position n indicates a task in the n-th element
of system_task_buffer. At boot time, schedule_task() locates the first free index in the task buffer and the corresponding location in the active and idle bitmaps are set and cleared, respectively.

In the execution phase, a call to ffs() is performed on the active task bitmap, as shown in Listing 2. The ffs() function, provided by the Atmel AVR C library [19], returns the position of the least significant bit set in a 16-bit word; or 0, if none are set. If a task is identified in the active task queue with a due date greater than the current system time, its index position is cleared in the active task bitmap and set in the idle task bitmap. If the identified task has an expired due date, it is executed by run_task(), followed by its removal or rescheduling. Task removal entails removal of the corresponding task bit from the active task bitmap. Task rescheduling involves updating the two bitmaps and the due date of the task in system_task_buffer.

During the execution cycle, if there are no tasks to execute, the scheduler performs an O(1) lookup into the active task queue and returns to sleep. While O(1) run-time is desirable, a large constant results in increased power consumption. We next consider a design that introduces increased overhead when there are tasks to execute, but very little overhead when there are no tasks to execute — our common case.

C. The O(n) Scheduler

The O(n) scheduler removes the call to the expensive ffs() function; it requires constant time to identify a task to execute, and linear time to reschedule the task post-execution.

Tasks are stored as nodes in a linked list instead of the statically allocated task array. Slab allocation is implemented using a static block of memory capable of holding N task nodes, task_free_list, a pointer to the list of free memory (within the static memory block), and task_queue, a pointer to the linked list of tasks. Task scheduling involves allocating a node from task_free_list, populating the node, and inserting the node in task_queue based on due date.

Listing 3. scheduler_run() (O(n) Scheduler)

```c
1 void scheduler_run() {
2   int32_t system_sleep_cycle_counter = 0;
3   while (true) {
4     bool task_executed = false;
5     do {
6       task_executed = false;
7       system_sleep_cycle_counter = current_system_time();
8       while (task_queue != NULL) {
9         // execute the task
10        task_ptr = task_queue;
11        task_ptr = task_ptr >> task_priority;
12        task_ptr = task_ptr >> task_priority;
13        task_queue = task_queue >> task_priority;
14        if (task_ptr->due_date <= current_system_time()) {
15          // handle task; sleeping is removed
16          if (task_ptr->due_date == ONE_SECOND) {
17            free task_free[task_free_list] (node_ptr) task_ptr;
18          } else {
19            task_ptr->due_date = task_ptr->due_date;
20            insert task_in_sleeping_queue(task_queue, task_ptr);
21          }
22        } else {
23          system_sleep_cycle_counter = task_queue->due_date - current_system_time;
24          while (system_sleep_cycle_counter >> 31)
25            set_sleep_mode(SLEEP_MODE_PWR_SAVE);  // current system time is recorded at the end of an execution
26          while (system_sleep_cycle_counter >> 31)
27            set_sleep_mode(SLEEP_MODE_PWR_SAVE);  // current system time is recorded at the end of an execution
28        }
29     } while (task_executed);
30   }
31 }
```

Listing 4. scheduler_run() and intelligent_sleep() (Intelligent Sleep Scheduler)

Listing 4 presents the scheduler_run() implementation. The difference between the earliest task due date and the current system time is recorded at the end of an execution
cycle. The system then invokes `intelligent_sleep()`, which partitions this value into multiple divisors, so as to calculate the least number of sleep cycles that can be created from 1, 2, and 8-second intervals.

The current rate at which the interrupt is triggered is called an `epoch`. Changing the clock prescaler (and the epoch) at any arbitrary instant causes the 8-bit counter register to contain a value less than 256, accounting for the partial second of elapsed time since the last overflow interrupt. Since epoch values vary over time, the semantics of this `partial_time` change in a complex way. Let the epoch be $e_1$ at time $t_1$ when the overflow interrupt is triggered. Let the epoch assume the value $e_2$ at $t_2$. Partial time is defined as $(t_2 - t_1)$, calculated as a function of $e_1$ and the value in the 8-bit counter register when the epoch was changed to $e_2$. Partial times for each epoch (i.e., 1, 2, 8) are stored in an array.

```c
#define PARTIAL_TIME_UPDATE() \ 
// update system time based on partial time accumulation \ 
system_clock_cycles = system_time + system_clock_cycles; \ 
if (system_clock_cycles > 0) \ 
    system_time = system_clock_cycles; \ 
    system_clock_cycles = 0; \ 
}

int sleep_cycle();

// interrupt system time based on current epoch \ 
system_time += system_clock_cycles; \ 
if (system_clock_cycles & 0x1F) \ 
    system_clock_cycles = 0; \ 
else if (sleep_cycle()) \ 
    system_clock_cycles = 0;

// 8-second sleep required; decrement cnt, change prescaler if required \ 
if (system_time >= 8) \ 
    system_time = system_time - 8; \ 
    TCCR1B = 0x00; \ 
while (ASR & 0x0F); \ 

// 2-second sleep required; decrement cnt, change prescaler if required \ 
if (system_time >= 2) \ 
    system_time = system_time - 2; \ 
    TCCR1B &= 0x7E; \ 
while (ASR & 0x0F); \ 

// 1-second sleep required; decrement cnt \ 
if (system_time >= 1) \ 
    system_time = system_time - 1; \ 
    TCCR1B &= 0x7C; \ 
while (ASR & 0x0F); \ 

// 0.5-second sleep required \ 
if (system_time >= 0.5) \ 
    system_time = system_time - 0.5; \ 
    TCCR1B &= 0x78; \ 
while (ASR & 0x0F); \ 

// 0.25-second sleep required \ 
if (system_time >= 0.25) \ 
    system_time = system_time - 0.25; \ 
    TCCR1B &= 0x74; \ 
while (ASR & 0x0F); \ 

while (ASR & 0x0F); \ 

PARTIAL_TIME_UPDATE();
```

Listing 5. Overflow ISR (Intelligent Sleep Scheduler)

To obtain the least accumulated partial epoch, the overflow ISR is identified as the optimal place to change the prescaler. Thus, after an execution cycle, the processor enters a 1-second sleep period, waits for the ISR to be triggered, and then changes the prescaler. Listing 5 contains the code for the updated overflow ISR. The overflow ISR ensures that the prescaler is set to the 1-second interval for the mandatory sleep cycle after the 2 and 8-second sleep cycles have been executed.

At the start of the ISR, the system time is updated using the value of the current epoch. Next, the change of prescaler (and epoch) is performed, if needed. If the clock prescaler is updated, the corresponding partial time is recorded, and the value of accumulated partial time is calculated as the sum of its previous value and the product of the current partial time and the last epoch value. Since every 256 fractions represents 1 second of time, if accumulated partial time is greater than or equal to 255, the system time is incremented and the accumulated partial time is appropriately updated.

V. Algebraic Models

The schedulers were implemented for the MoteStack, a state-of-the-art in-situ sensing platform, which uses an ATMega644 Atmel 8-bit AVR RISC-based MCU operating at 10 MHz at 3.3V (gcc -0s). A line-by-line code analysis was performed with the assistance of AVR Studio, a cycle accurate simulator, to derive the closed-form algebraic models.

A. The Basic Scheduler

In the basic scheduler, the null activation period, $A_1$, is given (in $\mu$s) by:

$$A_1 = 8.9 + 1.5 \cdot n_{queue\_capacity} + 1.3 \cdot n_{in\_queue}$$

where $n_{queue\_capacity}$ denotes the capacity of the task queue, and $n_{in\_queue}$ denotes the number of tasks in the queue.

$A_2$ (in $\mu$s) is given by the following formula:

$$A_2 = 8.9 + 3.1 \cdot n_{executed} + 2.6 \cdot n_{iter} + (1.5 \cdot n_{queue\_capacity} + 1.3 \cdot n_{in\_queue}) \cdot n_{iter}$$

Recall that $n_{executed}$ denotes the number of task functions executed in the current task activation period; $n_{iter}$ denotes the number of times the main scheduler loop executes (Listing 1, lines 4-24). Assuming that $\forall i$, $(w_i + A_2) \leq 1$ second, the value of $n_{iter}$ is calculated as follows:

$$n_{iter} = 1 + \left[ \frac{1}{\text{task\_period}_{min}} \right]$$

where $\text{task\_period}_{min}$ is the smallest period value present in the task queue associated with a task that has a due date earlier than the current system time.

B. The $O(1)$ Scheduler

In the $O(1)$ scheduler, $A_1$ is given by:

$$A_1 = 14.5 + 24 \cdot n_{in\_queue} + (2.8 \cdot \frac{\left( n_{queue\_capacity} \right) - 1}{16}) \cdot n_{in\_queue}$$

$A_2$ for the $O(1)$ scheduler is given as follows:

$$A_2 = 19.9 + 6.5 \cdot n_{executed} + 24 \cdot n_{in\_queue} \cdot (n_{iter} - 1) + 2.8 \cdot \left( \frac{\left( n_{queue\_capacity} \right) - 1}{16} \right) \cdot n_{in\_queue} \cdot (n_{iter} - 1)$$

C. The $O(n)$ Scheduler

The $O(n)$ scheduler has a constant null activation period of $7 \mu$s ($A_1$).

$A_2$ is given by the following formula:

$$A_2 = 14.4 + (13.7 + t_{ins}) \cdot n_{executed} + 5.6 \cdot (n_{iter} - 1)$$

$t_{ins}$ denotes the time spent within the insertion sort during rescheduling, post task execution. The value of $t_{ins}$ is given by the following formula:

$$t_{ins} = \begin{cases} \frac{0.2}{n_{in\_queue}} & \text{if } n_{in\_queue} = 0; \\ \frac{3.7 \cdot \left( n_{in\_queue} \right)}{n_{in\_queue}} & \text{if } n_{in\_queue} > 0; \end{cases}$$

where $\left( n_{in\_queue} \right)$ denotes any value between 1 and $n_{in\_queue} - 1$. 

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-207-3 288
31,746 occurrences of A for these 31.5 seconds. Partial time accumulation is small. Even if it does accumulate, interval case, the prescalar is set to 128, and the probability of oscillation of the external oscillator. Even in the 1-second time accumulated is approximately 31.5 seconds. Partial update logic (lines 35-38) for modeling purposes. The latest possible invocation of the update logic requires 51.4 seconds shown in Figure 2.

The accumulation of partial time fractions in the clock update logic requires 51.4 \mu s. However, this value is ignored for modeling purposes. The latest possible invocation of the partial update logic (lines 35-38, Listing 5) is approximately 31.5 \mu s after the start of the ISR. Thus, the maximum partial time accumulated is approximately 31.5 \mu s, close to a single oscillation of the external oscillator. Even in the 1-second interval case, the prescalar is set to 128, and the probability of partial time accumulation is small. Even if it does accumulate, for these 31.5 \mu s intervals to total 1 second, approximately 31,746 occurrences of A1 or A2 are required. Hence, the time is assumed to be negligible.

A2 for the ISS is given by:

\[ A_2 = \tau_{ISR} + 13.4 + 5.6 \times (n_{iter} - 1) + (13.7 + \tau_{ins}) \times n_{executed} \]  

where \( n_{iter} \), \( n_{executed} \), \( \tau_{ins} \), and \( \tau_{ISR} \) are defined as before.

VI. RESULTS

We first consider the performance of the schedulers based on the algebraic models of their behavior. We then measure the scheduler power consumption for a given set of tasks on physical hardware.

A. Comparative Analysis

We compare the scheduling overhead of each scheduler under varying load conditions; results are shown in Figures 3 and 4. Due to the number of variables in the equations for A1 and A2, we make some assumptions to limit the evaluation space. We fix both \( n_{queue \_capacity} \) and \( n_{in \_queue} \) to 128, and \( n_{iter} \) to 2 (limiting \( task \_period_{min} \) to greater than or equal to 1 second – Eq. (5)). We generate the values of \( \tau_{ins} \) using a pseudo random number generator and fix the values for all subsequent calculations across the schedulers. For each scheduler, we measure the scheduling overhead, given by \( n_1 A_1 + n_2 A_2 \), in seconds, on the Z-axis, when \( N \) is set to 500 seconds. \( N \) is composed of \((n_1 + n_2)\) 1-second counts. We plot the fraction of tasks executed on the X-axis, given by \( \frac{n_{task \_executed} \_over \_n_{inst \_queue}}{n_{task \_executed} \_over \_n_{inst \_queue}} \), and the load factor (given by \( n_2 \) over \((n_1 + n_2)\)) on the Y-axis. The system load factor is helpful in understanding the interplay between \( A_1 \) and \( A_2 \).

Figures 3(a) and 3(b) show the results for the basic and \( O(1) \) schedulers, respectively. The planar slopes for both graphs are similar, owing to the fact that both schedulers yield \( A_2 \) values that depend primarily on similar \( n_{queue \_capacity} \) and \( n_{in \_queue} \) coefficients. At higher load factors, where \( n_2 >> n_1 \), the
We now characterize the power consumption profiles of the four schedulers. For this purpose, we installed a test application on the MoteStack device, using each scheduler. The application schedules a periodic null task with a duration of 750 ms, executed every 10 s. We connected a 10Ω resistor in series with the power supply of the MoteStack and measured the voltage difference across the resistor, using an oscilloscope. The voltage change is directly proportional to the current draw (and power consumption, when voltage is constant) by Ohm’s Law. Figures 5(a)–5(d) summarize the consumption profiles for the four schedulers. In each graph, the horizontal axis represents time, and the vertical axis represents current draw. The bottom halves of the figures show the complete consumption profile; the task activation periods are visible. The top halves show a magnified view of the profile, such that the null activation periods can be seen. The peaks for the null activation periods can be observed at the end of each second in Figures 5(a), 5(b), and 5(c), while fewer such peaks can be noticed in Figure 5(d), indicating longer sleep periods.

We sample data over a 10-second window, which captures current draw values for a single task activation period, multiple null activation periods, and the associated sleep periods. We calculate the average overall and $A_{TASK}$ current draws—the $A_{TASK}$ values vary due to the inherent scheduler designs. The average current draw for the basic scheduler (Figure 5(a)) over the window is 0.613 mA (average $A_{TASK}$ current draw is 5.52 mA), while the average current draw for the $O(1)$ scheduler (Figure 5(b)) is 0.605 mA (average $A_{TASK}$ current draw is 5.28 mA). The average current consumption for the $O(n)$ (Figure 5(c)) and the intelligent sleep (Figure 5(d)) schedulers is 0.616 mA (average $A_{TASK}$ current draw is 5.56 mA) and 0.603 mA (average $A_{TASK}$ contribution is 5.49 mA), respectively.

Figure 6 presents the life expectancy of a 1000mAh battery.

---

$O(1)$ scheduler performs worse than the basic scheduler, but at lower load factors, the differences are negligible. Figures 3(c) and 3(d) show the results for the $O(n)$ scheduler and ISS, respectively; again the curves are similar. The $O(n)$ scheduler and ISS incur less overhead than the basic and $O(1)$ schedulers at load factors below 0.8, as they are not dependent on $n_{queue\_capacity}$. We also observe that at higher load factors, the value of $n_{task\_executed}$ affects all schedulers significantly. At lower load factors, both the $O(n)$ and intelligent schedulers exhibit very low overhead (<2% for load factors of 0.3). To further differentiate the two schedulers, we consider their performance at very low load factors, on the order of 0.001, typical in AAS systems. Since the overhead contribution of $A_1$ is significantly larger than $A_2$ at very low load factors, we focus on the impact of $A_1$ in isolation. In Figures 4(a) and 4(b), we measure, for each scheduler, the contribution of $A_1$, given by $n_1A_1$, on the Y-axis, against the load factor, given by $n_2$ over $n_1 + n_2$, on the X-axis. With a side-by-side comparison, we see that the basic and $O(1)$ schedulers have a much higher null activation period contribution than the other two schedulers—approximately three orders of magnitude larger and are relatively inefficient at lower load factors. We also observe that the ISS performs the best among all the schedulers presented. Its ability to sleep for longer periods of time gives the ISS an edge over schedulers which need to wake every second.

B. Power Consumption Profile

We now characterize the power consumption profiles of the four schedulers. For this purpose, we installed a test application on the MoteStack device, using each scheduler. The application schedules a periodic null task with a duration of 750 ms, executed every 10 s. We connected a 10Ω resistor in series with the power supply of the MoteStack and measured the voltage difference across the resistor, using an oscilloscope. The voltage change is directly proportional to the current draw (and power consumption, when voltage is constant) by Ohm’s Law. Figures 5(a)–5(d) summarize the consumption profiles for the four schedulers. In each graph, the horizontal axis represents time, and the vertical axis represents current draw. The bottom halves of the figures show the complete consumption profile; the task activation periods are visible. The top halves show a magnified view of the profile, such that the null activation periods can be seen. The peaks for the null activation periods can be observed at the end of each second in Figures 5(a), 5(b), and 5(c), while fewer such peaks can be noticed in Figure 5(d), indicating longer sleep periods.

We sample data over a 10-second window, which captures current draw values for a single task activation period, multiple null activation periods, and the associated sleep periods. We calculate the average overall and $A_{TASK}$ current draws—the $A_{TASK}$ values vary due to the inherent scheduler designs. The average current draw for the basic scheduler (Figure 5(a)) over the window is 0.613 mA (average $A_{TASK}$ current draw is 5.52 mA), while the average current draw for the $O(1)$ scheduler (Figure 5(b)) is 0.605 mA (average $A_{TASK}$ current draw is 5.28 mA). The average current consumption for the $O(n)$ (Figure 5(c)) and the intelligent sleep (Figure 5(d)) schedulers is 0.616 mA (average $A_{TASK}$ current draw is 5.56 mA) and 0.603 mA (average $A_{TASK}$ contribution is 5.49 mA), respectively.

Figure 6 presents the life expectancy of a 1000mAh battery.
when it is supplying power to a MoteStack, running the four schedulers under different almost-always-sleeping scenarios. Data for Figure 6 was obtained by extrapolating the average current draw and average $A_{\text{TASK}}$ current draw values from Figures 5(a) – 5(d), and applying them to applications which sleep for 5, 10, 15, 30, 45, and 60 minutes between task executions. We observe that the intelligent sleep scheduler consistently yields higher battery longevity for all the applications.

Specifically, consider the application which sleeps for 15 minutes between tasks, a typical sampling period for environmental monitoring networks of the type deployed in Aiken, SC. A MoteStack running this application and drawing its power from a 1000mAh battery would last approximately 5,374 hours, while the ISS offers the longest runtime, of approximately 5,980 hours – 10% longer than any of the other schedulers. This is a significant increase in longevity in the context of large sensor network deployments. Though all the scheduler designs dictate a linear decrease in power consumption with an increase in the time period between task activation periods, not surprisingly, the rate of the decrease for the ISS is higher compared to the others, due to its ability to sleep for longer periods, thus enabling a longer battery life.

VII. CONCLUSION AND FUTURE WORK

In this paper, we presented the design, implementation, and analysis of four progressively more efficient schedulers designed to support almost-always-sleeping embedded applications. This is the first systematic consideration of this increasingly relevant class of schedulers. We presented a basic scheduler which uses a rudimentary array to store tasks. We next presented the $O(1)$ scheduler based on the Linux 2.6.8.1 scheduler. This design incurs performance penalties due to an expensive call to ffs(). Next, we presented the $O(n)$ scheduler, which uses a priority queue to store tasks and improves its tracking of sleep cycles, performing significantly better than the previous schedulers. Finally, we presented the intelligent sleep scheduler, in which we make use of hardware features to extend physical sleep cycles, design a variable-sleep scheduling strategy, and further reduce scheduling overhead. On analyzing the scheduler runtimes, we observed that the $O(n)$ and the intelligent sleep schedulers work well below a certain load factor. However, under lower load cycles, the intelligent sleep scheduler design performs markedly better than all other designs due to its variable-sleep strategy, even at the expense of added code complexity.

VIII. ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation (awards CNS-0745846, CNS-1126344).

REFERENCES