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 model, the embedded device wakes, executes a task-set, and returns to sleep. These systems spend most of their time in a state of deep sleep to minimize power consumption. We refer to these systems as *almost-always-sleeping* (AAS) systems. 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 10s of *milli*watts in the active state, and only 10s of *micro*watts 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.

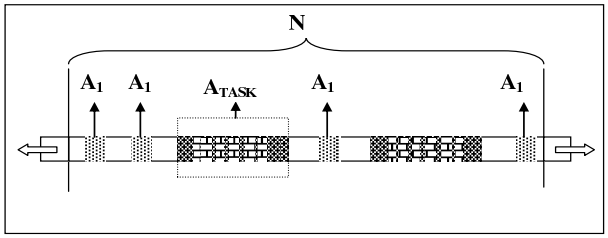
We emphasize that these designs are practically motivated. They evolved over the course of 18 months while developing a large-scale environmental monitoring network deployed in the city of Aiken, South Carolina [8]. In 2011, the city's stormwater treatment system was redesigned to reduce the environmental 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.

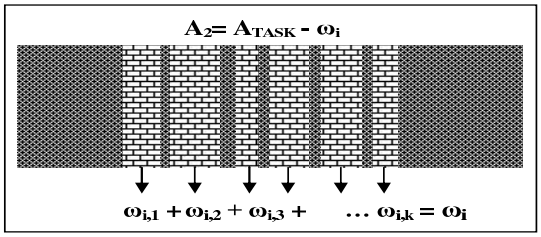
II. Problem Statement

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

 ISBN: 978-1-61208-207-3



(a) Components of N



(b) A_{TASK} Expanded

Fig. 1. A_1, A_2, A_{TASK}, ω_i, and N

task execution time, is referred to as the *task activation period*, denoted by A_{TASK}.

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

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

The *scheduler load* α is the fraction of time the system is either busy scheduling tasks or executing them within time period N. The *task load* β 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 $n_2 A_{TASK} = n_2 A_2 + W$, α can be expressed as:

$$\begin{aligned} \alpha &= \frac{(\mathtt{n_1 A_1 + n_2 A_{TASK}})}{\mathtt{N}} \\ &= \frac{(\mathtt{n_1 A_1 + n_2 A_2 + W})}{\mathtt{N}} \\ &= \frac{\mathtt{n_1 A_1 + n_2 A_2}}{\mathtt{N}} + \beta \end{aligned} \tag{2}$$

Objective. In an ideal scheduler, with no scheduling overhead, $\alpha = \beta$. To minimize the value of α, 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 soutions 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

 ISBN: 978-1-61208-207-3

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.

```
1 void scheduler_run() {
2  while(true) {
3   bool task_executed;
4   do {
5    task_executed = false;
6    uint32_t current_time = current_system_time();
7    uint8_t task_index;
8    for(task_index = 0; task_index < TASK_QUEUE_CAPACITY; task_index++) {
9     // if the current (non-empty) task is due
10    if((system_task_buffer[task_index].task != NULL) &&
11       (current_time >= system_task_buffer[task_index].due_date)) {
12     // execute the task function
13     (*system_task_buffer[task_index].task)();
14     // handle rescheduling / removal
15     if(system_task_buffer[task_index].type == ONE_SHOT) {
16      system_task_buffer[task_index].task = NULL;
17     } else {
18      system_task_buffer[task_index].due_date +=
19       system_task_buffer[task_index].period;
20     }
21     task_executed = true;
22    }
23   }
24  } while(task_executed);
25  set_sleep_mode(SLEEP_MODE_PWR_SAVE);
26  sleep_mode();
27 }
28 }
```

Listing 1. `scheduler_run()` (Basic Scheduler)

```
1 void scheduler_run() {
2  while(true) {
3   bool task_executed;
4   do {
5    task_executed = false;
6    uint32_t current_time = current_system_time();
7    uint8_t task_index = 0;
8    while((task_index = 16 - ffs(task_bitmap_active)) < 16) {
9     if (current_time >= system_task_buffer[task_index].due_date) {
10     task_executed = run_task(task_index,current_time);
11    } else {
12     task_bitmap_active &= ~(1 << (15 ^ task_index));
13     task_bitmap_inactive |= (1 << (15 ^ task_index));
14    }
15   }
16  } while(task_executed);
17  task_bitmap_active = task_bitmap_inactive;
18  task_bitmap_inactive = 0;
19  set_sleep_mode(SLEEP_MODE_PWR_SAVE);
20  sleep_mode();
21 }
22 }
23
24 static inline bool run_task(uint8_t task_index,uint32_t current_time) {
25  // execute the task
26  (*system_task_buffer[task_index].task)();
27  // handle rescheduling / removal
28  if(system_task_buffer[task_index].type == ONE_SHOT) {
29   task_bitmap_active &= ~(1 << (15 ^ task_index));
30  } else {
31   system_task_buffer[task_index].due_date +=
32    system_task_buffer[task_index].period;
33   if(system_task_buffer[task_index].due_date > current_time) {
34    task_bitmap_active &= ~(1 << (15 ^ task_index));
35    task_bitmap_inactive |= (1 << (15 ^ task_index));
36   }
37  }
38  return (true);
39 }
```

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 1 at bit position n indicates a task in the n^{th} element

 ISBN: 978-1-61208-207-3

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.

```
1 void scheduler_run() {
2  uint32_t system_sleep_cycle_counter = 0;
3  while(true) {
4   bool task_executed;
5   do {
6    task_executed = false;
7    uint32_t current_time = current_system_time();
8    while((task_queue != NULL) &&
9          (task_queue->due_date <= current_time)) {
10    // execute the task
11    task_node_ptr_t task_ptr = task_queue;
12    (task_ptr->task)();
13    task_executed = true;
14    task_queue = task_queue->next;
15    // handle rescheduling / removal
16    if(task_ptr->type == ONE_SHOT) {
17     free_list_free(&task_free_list, (node_ptr_t) task_ptr);
18    } else {
19     task_ptr->due_date += task_ptr->period;
20     insert_task_in_scheduling_queue(&task_queue, task_ptr);
21    }
22   }
23   system_sleep_cycle_counter = task_queue->due_date - current_time;
24  } while(task_executed);
25  set_sleep_mode(SLEEP_MODE_PWR_SAVE);
26  while(system_sleep_cycle_counter--) {
27   sleep_mode();
28  }
29 }
30 }
```

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

scheduler_run(), shown in Listing 3, traverses the list of scheduled tasks and executes those that are due. The removal of one_shot tasks is handled by freeing the corresponding task node and returning it to task_free_list. Rescheduling of periodic tasks is handled by updating the corresponding task's due date and re-inserting the task at the correct position in the priority queue.

Since tasks are ordered by due date, it is straightforward to determine when the next task needs to be executed, just prior to sleeping. When the system is done executing tasks, the difference between the earliest task due date and the current system time is recorded. After the system wakes up, a simple check on this value allows the scheduler to decide if there are any tasks to execute, and saves it from having to access the node list. The scheduler therefore experiences shorter wake cycles when there are no tasks to execute.

Since an AAS system typically wakes to find nothing to execute, even a small amount of time expended during a wake cycle can add performance penalties. With the given hardware and interrupt design, where the processor has to wake every second, this is the best performance that could be achieved. However, a scheduler capable of altering the interrupt behavior would yield even better performance.

D. The Intelligent Sleep Scheduler

The basis of the intelligent sleep scheduler (ISS) is the $O(n)$ scheduler, with multiple updates to the wake, sleep, and clock logic. The central idea is that the rate at which the overflow interrupt is generated can be changed by choosing a different clock prescaler, thus making the duration of the processor sleep period tunable; the clock prescaler can be set to 128, 256, or 1024, so that overflow interrupts are triggered at 1, 2, and 8 second intervals, respectively.

```
1 void scheduler_run() {
2  uint32_t system_sleep_cycle_counter = 0;
3  while(true) {
4   bool task_executed;
5   do {
6    task_executed = false;
7    uint32_t current_time = current_system_time();
8    while((task_queue != NULL) &&
9          (task_queue->due_date <= current_time)) {
10    ... same as O(n) scheduler ...
11   }
12   system_sleep_cycle_counter = task_queue->due_date - current_time;
13  } while(task_executed);
14  intelligent_sleep(system_sleep_cycle_counter);
15 }
16 }
17
18 inline void intelligent_sleep(uint32_t int_system_sleep_counter) {
19  int_system_sleep_counter = int_system_sleep_counter - 1;
20  // determine the number of 1, 2, and 8 second sleep cycles.
21  // 1 second sleep required?
22  sleep_cycle[0] = (int_system_sleep_counter & 0x1);
23  // 2 second sleep required?
24  int_system_sleep_counter >>= 1;
25  sleep_cycle[1] = (int_system_sleep_counter & 0x1);
26  int_system_sleep_counter >>= 1;
27  sleep_cycle[1] += ((int_system_sleep_counter & 0x1) << 1));
28  // 8 second sleep required?
29  int_system_sleep_counter >>= 1;
30  sleep_cycle[2] = int_system_sleep_counter;
31
32  // compute total number of sleep cycles and begin sleeping
33  int_system_sleep_counter = sleep_cycle[0]
34    + sleep_cycle[1] + sleep_cycle[2];
35  set_sleep_mode(SLEEP_MODE_PWR_SAVE);
36  do {
37   sleep_mode();
38  } while(int_system_sleep_counter--);
39 }
```

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

 ISBN: 978-1-61208-207-3

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.

```
1 #define PARTIAL_TIME_UPDATE() \
2   // update system time based on partial time accumulation \
3   system_clock_cycles += system_time_fraction*temp_system_time_epoch; \
4   if(system_clock_cycles & ~(0xFF)) { \
5    system_time += system_clock_cycles >>8; \
6    system_clock_cycles &= 0xFF; \
7   }
8
9 // timer2 overflow handler
10 ISR(SIG_OVERFLOW2, ISR_BLOCK) {
11  // increment system time by current epoch
12  system_time += system_time_epoch;
13  if (sleep_cycle[0]) {
14   // 1-second sleep required; current epoch 1-second
15   sleep_cycle[0] = 0;
16  } else if (sleep_cycle[1]) {
17   // 2-second sleep required; decrement cntr, change prescaler if required
18   sleep_cycle[1]--;
19   if (system_time_epoch != 2) {
20    temp_system_time_epoch = system_time_epoch;
21    system_time_epoch = 2;
22    TCCR2B = (1 << CS22) | (1 << CS21);
23    while(ASSR & 0x1F);
24    system_time_fraction = TCNT2;
25    TCNT2 = 0x0;
26    while(ASSR & 0x1F);
27    PARTIAL_TIME_UPDATE();
28   }
29  } else if (sleep_cycle[2]) {
30   // 8-second sleep required; decrement cntr, change prescaler if required
31   sleep_cycle[2]--;
32   if (system_time_epoch != 8) {
33    ... analogous to above case ...
34   }
35  } else if (system_time_epoch != 1) {
36   // all counters are 0; prescaler reset for mandatory 1-second sleep
37   ... analogous to above case ...
38 }
39}
```

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 μs) by:

$$A_1 = 8.9 + 1.5 * \mathtt{n}_{\mathtt{queue\_capacity}} + 1.3 * \mathtt{n}_{\mathtt{in\_queue}} \quad (3)$$

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

A_2 (in μs) is given by the following formula:

$$A_2 = 8.9 + 3.1 * \mathtt{n}_{\mathtt{executed}} + 2.6 * \mathtt{n}_{\mathtt{iter}} + (1.5 * \mathtt{n}_{\mathtt{queue\_capacity}} + 1.3 * \mathtt{n}_{\mathtt{in\_queue}}) * \mathtt{n}_{\mathtt{iter}} \quad (4)$$

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

$$\mathtt{n}_{\mathtt{iter}} = 1 + \lceil \frac{1}{\mathtt{task\_period}_{\mathtt{min}}} \rceil \quad (5)$$

where $\mathtt{task\_period}_{\mathtt{min}}$ 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:

$$A_1 = 14.5 + 24 * \mathtt{n}_{\mathtt{in\_queue}} + (2.8 * (\lceil \frac{\mathtt{n}_{\mathtt{queue\_capacity}}}{16} \rceil - 1)) * \mathtt{n}_{\mathtt{in\_queue}} \quad (6)$$

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

$$A_2 = 19.9 + 6.5 * \mathtt{n}_{\mathtt{executed}} + 24 * \mathtt{n}_{\mathtt{in\_queue}} * (\mathtt{n}_{\mathtt{iter}} - 1) + 2.8 * (\lceil \frac{\mathtt{n}_{\mathtt{queue\_capacity}}}{16} \rceil - 1) * \mathtt{n}_{\mathtt{in\_queue}}) * (\mathtt{n}_{\mathtt{iter}} - 1) \quad (7)$$

C. The $O(n)$ Scheduler

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

A_2 is given by the following formula:

$$A_2 = 14.4 + (13.7 + \mathtt{t}_{\mathtt{ins}}) * \mathtt{n}_{\mathtt{executed}} + 5.6 * (\mathtt{n}_{\mathtt{iter}} - 1) \quad (8)$$

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

$$\mathtt{t}_{\mathtt{ins}} = \begin{cases} 0.2, & \text{if } \mathtt{n}_{\mathtt{in\_queue}} = 0; \\ 3.7 * [1, \mathtt{n}_{\mathtt{in\_queue}}) & \text{if } \mathtt{n}_{\mathtt{in\_queue}} > 0; \end{cases} \quad (9)$$

where $[1, \mathtt{n}_{\mathtt{in\_queue}})$ denotes any value between 1 and ($\mathtt{n}_{\mathtt{in\_queue}}$ - 1).

 ISBN: 978-1-61208-207-3

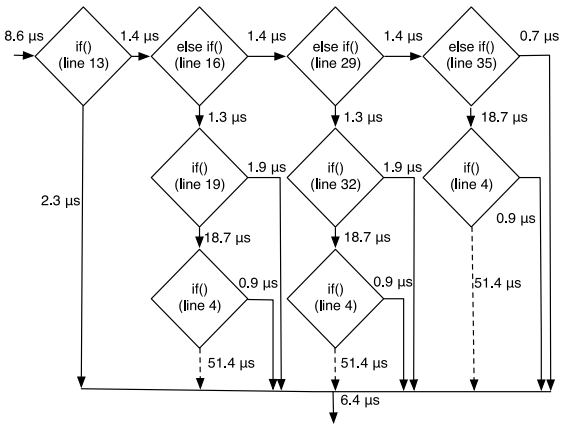


Fig. 2. ISR Execution Profile ($\mathtt{t_{ISR}}$) (Intelligent Sleep Scheduler)

D. The Intelligent Sleep Scheduler

The null activation period (A_1) for the ISS is the hardest to analyze due to its complex ISR control flow paths. A flow chart indicating the different paths is shown in Figure 2. The value A_1 assumes in a given null activation period depends on the values of the various system variables in that specific period and is given by:

$$\mathtt{A_1} = 0.6 + \mathtt{t_{ISR}} \tag{10}$$

$\mathtt{t_{ISR}}$ denotes the amount of time elapsed between the start of the overflow ISR (*line 10*, Listing 5) and the start of the scheduling loop of `scheduler_run()` (*line 6*, Listing 4), shown in Figure 2.

The accumulation of partial time fractions in the clock update logic requires 51.4 μs. 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 μs after the start of the ISR. Thus, the maximum partial time accumulated is approximately 31.5 μs, 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 μs intervals to total 1 second, approximately 31,746 occurences of A_1 or A_2 are required. Hence, the time is assumed to be negligible.

A_2 for the ISS is given by:

$$\begin{aligned} A_2 = {} & \mathtt{t_{ISR}} + 13.4 + 5.6 * (\mathtt{n_{iter}} - 1) \\ & + (13.7 + \mathtt{t_{ins}}) * \mathtt{n_{executed}} \end{aligned} \tag{11}$$

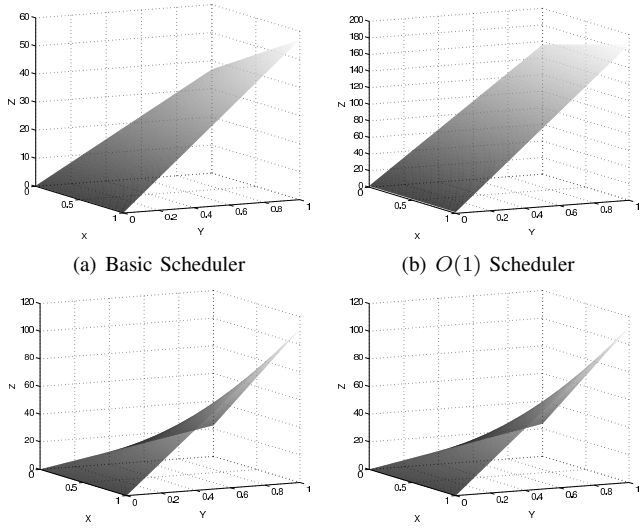
where $\mathtt{n_{iter}}$, $\mathtt{n_{executed}}$, $\mathtt{t_{ins}}$, and $\mathtt{t_{ISR}}$ are defined as before.

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

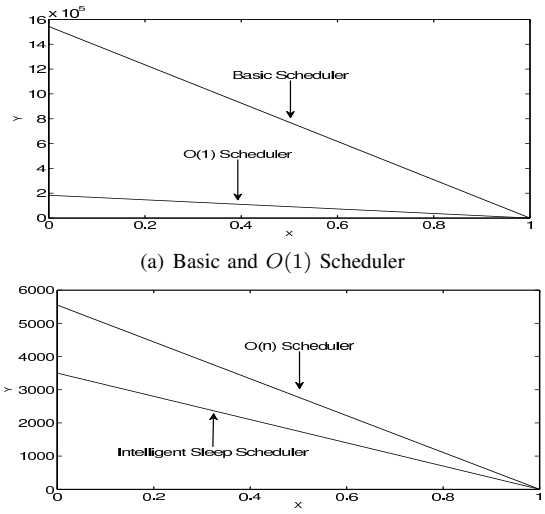
We compare the scheduling overhead of each scheduler under varying load conditions; results are shown in Figures 3 and 4. Due to the number of variables in the equations for A_1 and A_2, we make some assumptions to limit the evaluation



(a) Basic Scheduler (b) $O(1)$ Scheduler

(c) $O(n)$ Scheduler (d) Intelligent Sleep Scheduler

Fig. 3. X = $\frac{n_2}{n_1+n_2}$, Y = $\frac{\mathtt{n_{task\_executed}}}{\mathtt{n_{in\_queue}}}$, Z = $n_1A_1 + n_2A_2$



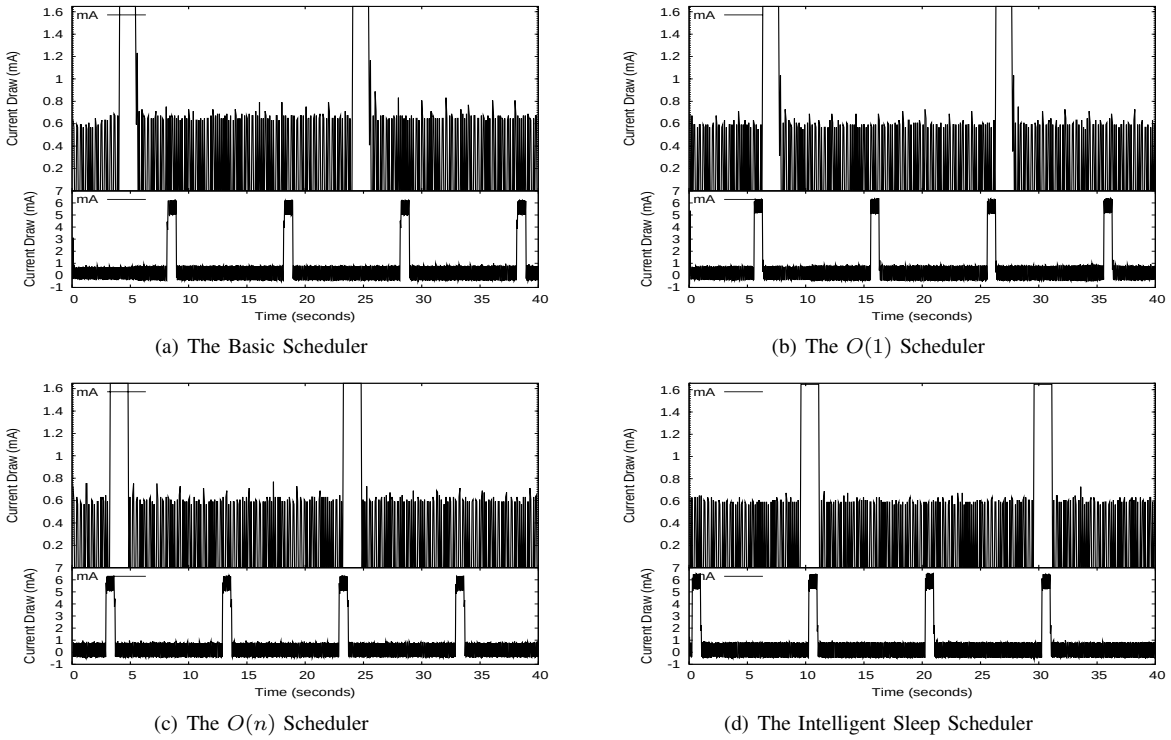
(a) Basic and $O(1)$ Scheduler

(b) $O(n)$ and Intelligent Sleep Scheduler

Fig. 4. *Null Activation Period* Contributions (X = $\frac{n_2}{n_1+n_2}$, Y = n_1A_1 μs)

space. We fix both $\mathtt{n_{queue\_capacity}}$ and $\mathtt{n_{in\_queue}}$ to 128, and $\mathtt{n_{iter}}$ to 2 (limiting $\mathtt{task\_period_{min}}$ to greater than or equal to 1 second – Eq. (5)). We generate the values of $\mathtt{t_{ins}}$ 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 $n_1A_1 + n_2A_2$, 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 $\mathtt{n_{task\_executed}}$ over $\mathtt{n_{in\_queue}}$, 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 $\mathtt{n_{queue\_capacity}}$ and $\mathtt{n_{in\_queue}}$ coefficients. At higher load factors, where $n_2 >> n_1$, the



(a) The Basic Scheduler

(b) The $O(1)$ Scheduler

(c) The $O(n)$ Scheduler

(d) The Intelligent Sleep Scheduler

Fig. 5. Scheduler Power Consumption Profiles

$O(1)$ 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 $\mathtt{n_{queue\_capacity}}$. We also observe that at higher load factors, the value of $\mathtt{n_{task\_executed}}$ 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 $n_1 A_1$, 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 $750ms$, executed every $10s$. 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,

 ISBN: 978-1-61208-207-3

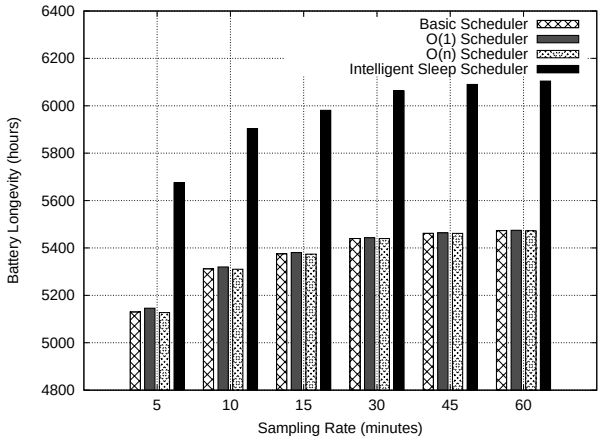


Fig. 6. Battery Life Expectancy

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_{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 5,375 hours using the basic scheduler. The same MoteStack would last for 5,380 hours using the $O(1)$ scheduler. A MoteStack using the $O(n)$ scheduler would last for 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

[1] Josh Aas. Understanding the linux 2.6.8.1 cpu scheduler. *Silicon Graphics International, 2005. http://joshaas.net/linux/* [retrieved: June, 2012].

[2] Shah Bhatti et al. MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10:563–579, 2005.

[3] Qing Cao, Tarek Abdelzaher, John Stankovic, and Tian He. The LITEOS operating system: Towards UNIX-like abstractions for wireless sensor networks. IPSN '08, pages 233–244. IEEE, 2008.

[4] Alexandru Caracas et al. Energy-efficiency through micro-managing communication and optimizing sleep. In *SECON*, pages 55–63, 2011.

[5] Hojung Cha et al. RETOS: Resilient, expandable, and threaded operating system for wireless sensor networks. IPSN '07, pages 148–157. ACM, 2007.

[6] Yu-Ting Chen, Ting-Chou Chien, and Pai H. Chou. ENIX: A lightweight dynamic operating system for tightly constrained wireless sensor platforms. SenSys '10, pages 183–196. ACM, 2010.

[7] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. CONTIKI - a lightweight and flexible operating system for tiny networked sensors. *Annual IEEE Conference on Local Computer Networks*, pages 455–462, 2004.

[8] Gene W. Eidson et al. The South Carolina digital watershed: End-to-end support for real-time management of water resources. *International Journal of Distributed Sensor Networks*, 2010.

[9] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. MobiSys '05, pages 163–176. ACM, 2005.

[10] Philip Levis et al. TINYOS: An operating system for sensor networks. In *Ambient Intelligence*, pages 115–148. Springer, 2005.

[11] Philip Levis et al. T2: A 2^{nd} generation OS for embedded sensor networks. 2005.

[12] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02*, pages 88–97. ACM Press, 2002.

[13] Joseph Polastre, Robert Szewczyk, and David Culler. TELOS: Enabling ultra-low power wireless research. In *IPSN '05*, pages 364–369. IEEE Press, 2005.

[14] Crossbow Technologies. Iris datasheet. http://bullseye.xbow.com:81/. [retrieved: June, 2012]

[15] Crossbow Technologies. Mica2 datasheet. http://bullseye.xbow.com:81/. [retrieved: June, 2012]

[16] Crossbow Technologies. MicaZ datasheet. http://bullseye.xbow.com:81/. [retrieved: June, 2012]

[17] Andreas Terzis et al. Wireless sensor networks for soil science. *International Journal of Sensor Networks*, 7:53–70, 2010.

[18] Gilman Tolle et al. A macroscope in the redwoods. In *SenSys '05*, pages 51–63. ACM Press, 2005.

[19] Joerg Wunsch et al. AVR Libc. http://www.nongnu.org/avr-libc/. [retrieved: June, 2012]

 ISBN: 978-1-61208-207-3