On Frequency Optimization for Power Saving in WSNs

Andreea Maria Picu  
INRIA ARES  
69621 Villeurbanne, France  
andreea.picu@insa-lyon.fr

Antoine Fraboulet  
INSA de Lyon/INRIA ARES  
69621 Villeurbanne, France  
antoine.fraboulet@insa-lyon.fr

Eric Fleury  
ENS Lyon/INRIA ARES  
69634 Lyon, France  
eric.fleury@inria.fr

Abstract

One of the most challenging problems in wireless sensor networks (WSNs) research is energy management. We propose two concepts aiming at saving power in low duty cycle applications. We first suggest a methodology for using hardware timers effectively. Then, we provide a way to calculate microcontroller (µC) configurations with various clock frequency setpoints, while respecting several types of constraints imposed on these frequencies, e.g., by other components of the µC, by protocol specifications, by external factors. Our evaluation shows that this approach can respect constraints while saving as much as 11.12% of energy when compared to a popular WSN operating system (OS).

1. Introduction

In recent years, embedded sensor networks have found their way into a wide variety of applications and systems with very diverse requirements and characteristics: disaster relief, environment monitoring, emergency medical response and home automation. However, in the collective conscience, the definition of sensor networks hardly changed since the early days of their military applications. This definition no longer holds for the civilian application areas mentioned above. Given the general trend towards diversification, a design space, rather than a definition, is now needed. Sensor networks should be conceived differently for groups of similar applications based on their characteristics and constraints with respect to the design space. Only then will WSNs truly be application-oriented.

Many WSN projects are currently using generic models based on popular OSes like TinyOS [8] or Contiki [2]. However, few of them have discussed the importance of specific models for sensor network programming and reconfiguration until now. Although it has non-negligible benefits, delegating this problem to generic frameworks often suffers from several drawbacks: no support for application professionals, failure to use and/or manage hardware efficiently, reductive energy management etc. Our work addresses these last two related issues.

Energy is a vital resource for mobile computing and there is unanimous consensus that advances in battery technology and low-power circuit design cannot, by themselves, meet the energy needs of future mobile systems. This is why energy management strategies must be developed for all levels: component, system, network, application etc. Schemes for power saving in WSNs often address communication protocols, but in order to account for the unique needs of each application, a global approach to the optimization of energy consumption is essential.

To provide a basis for application-specific energy administration, we discuss application-driven frequency scaling and enhanced hardware timers utilization. We present a software tool using a simple representation of the µC to configure the platform such that user and/or application timing requirements are satisfied and that power drawn from the battery is minimum.

2. Related Work and Motivation

Our work builds on the observation that generic WSN OSes use one unnecessarily high and fixed frequency, while the hardware supports several variable and much lower frequencies. Reducing the operating frequency will reduce the power dissipation linearly. However, in embedded systems, such as sensor nodes, frequency scaling is a delicate operation. Numerous features depend on and constrain clock frequency (e.g., components of the µC, protocols, applications), therefore reconfiguration will be needed if the frequency is changed. Time management, in particular, will be deeply affected by frequency scaling.

Current power saving mechanisms. There are generally multiple clocks in a µC and, except for the CPU, all other elements must choose from a set of clocks. These clocks themselves are the result of multiplexing several clock generators. Reducing the power consumption by scaling the frequency of the clocks will affect the entire platform. Some peripherals, like timers, are not easy to manage.
even with constant clock frequency. This is why embedded systems often use only one hardware timer to realize a list of software timers, even though many hardware timers are available. Scaling clock frequencies in a µC requires a mechanism to control all parts affected by scaling.

The frequency scaling technique reduces the processor clock frequency, allowing the processor to minimize the energy dissipation linearly. This technique saves energy even when it is not advantageous to go into Low Power Mode (LPM) at the expense of reduced performance. Although dynamic voltage scaling renders the lowest energy dissipation for most µCs, it is not always dramatically better than using a combination of dynamic frequency scaling and LPMs, which is much less expensive to implement [3]. Moreover, reducing power dissipation will have a significant positive impact on battery capacity, as shown in [9] and [12]. Frequency scaling is also essential if we plan to use voltage scaling in the future. Due to rapid advances in µC technology, we expect voltage scaling to be available for chips used in WSNs before long.

Many of today’s WSN OSES claim to be low power but they only consider LPMs for instructions. Similarly, previous studies on frequency scaling are limited to the core processor or µC and only at the circuit level or at most at the OS level. In a typical embedded system, the processor is attached to various peripherals, e.g., timers, serial ports etc. Few efforts have been made around peripheral integration for low power, even though a complete platform integration is essential in embedded systems.

**Time management in WSN OSES.** A critical part of any OS is a reliable and efficient timer service. In WSNs, application timer rates vary from a few events per week to sampling rates of 10 kHz or even higher. Ideally, hardware timers would run at the same frequency as application timers. However, WSN OSES only use one high frequency hardware timer to generate all the required application timers. Moreover, µCs provide two to four hardware timers, only one of which is used by OSES (see Figure 1).

Recent developments, like the abolition of the timer tick, largely improved time management in OSES in general. The trend extended to embedded real-time OSES with the release of TiROS [11]. Although it solves the problem of the trade-off between decent timer resolution (with increased tick frequency) and low power consumption, it still fails to fully use hardware capabilities. Full usage of hardware timers would reduce processing due to time management to a minimum.

To conclude, past efforts concerning frequency scaling and/or time management concentrate on hardware or OS. However, hardware only takes into account the past of the application and the OS handles the present. Only the application itself can really improve power consumption, since it has information about the future.

### 3. Frequency Optimization in WSNs

To optimize the interaction between hardware and software, we worked through several steps, illustrated in Figure 2 (code generation has not been addressed yet). First, we developed a novel timer allocation algorithm, since timers are one of the key µC subsystems in reducing operating frequency. We then used this algorithm to place constraints on hardware timers. These initial constraints allow us to obtain all valid hardware configurations, simply by walking the frequency optimization graph and applying the constraints associated with each vertex to the set of solutions. Both schemes are described in the following sections.

#### 3.1. Timer Management

The allocation of software timers to hardware timers is an important factor in determining the minimum frequency at which the µC can operate. As explained above, WSN OSES assign all software timers to one clock or hardware timer. The frequency is often very high (e.g., 2 MHz for TinyOS 2.x’s timer) relative to its optimal value, in order to accommodate a decent timer resolution. Our contribution is an allocation scheme that will calculate the minimum frequency required to provide all the timers for applications and the OS, while spreading these logical timers throughout the available hardware timers. In short, we switch from the approach presented in Figure 1 to the one in Figure 3.

The aim of our algorithm is to partition the set of software timers \( f_s \) into as many subsets as hardware timers \( f_h \) available. It must do so in a way that minimizes hardware timer frequencies. This is a set partitioning NP-complete optimization problem, that we solved using an
adaptation of Jensen’s algorithm [7]. Our evaluation shows that this algorithm gives far better results than less sophisticated heuristics, e.g. a greedy algorithm. We therefore obtain constraints on the hardware registers of µC timers from user and application constraints.

3.2. Frequency Optimal Configurations

Frequency scaling implies a lot of reconfiguration if we want to continue satisfying user and application requirements. This is why a hardware reconfiguration tool is essential for our project. This tool needs two inputs: a detailed description of relevant hardware and user and/or application requirements translated into constraints on hardware registers. The latter is provided by our timer management algorithm above. The former is presented in the following.

Hardware Description. Although we chose the TI MSP430 for this study, we do not make any assumptions on the µC or on the OS, hence generality is preserved. For the purpose of our analysis, we split the µC into several blocks, corresponding to subsystems sharing the same clock. Our blocks are roughly the equivalents of the µC’s peripherals as presented in [5]: Basic Clock Module, Timers A and B, ADC’s 10 and 12, Flash Controller and USART.

Our description includes the part of the hardware that is relevant to our study as a directed connected acyclic graph, in which source vertices are clock sources and sink vertices are usually frequency division registers. Since our reconfiguration tool only deals with clock frequencies, we represent only those registers that have a direct impact on hardware or timers. For now the TI MSP430 graph comprises the Basic Clock Module and Timers A and B [5]. Our plan is to include all the blocks mentioned in the previous paragraph.

In this frequency optimization graph, hardware registers are vertices and clocks are edges. Currently, we use two types of nodes corresponding to register types: divider and selector, and two other types needed for convenience: clock source and repeater. Nodes and edges are annotated with extra information. The repeater replicates the input edge into as many output edges as necessary, to avoid our structure being a hypergraph. Each type of node has specific information, e.g., possible frequencies for clock sources, division range or set for dividers, association between value of the selector and the selected clock for selectors, etc. The graph for TI MSP430’s Timer A is shown in Figure 4.

Computing Optimal Configurations. Hardware configurations consist of register values: one unique value for each register per configuration. Using our annotated graph, we calculate possible hardware configurations in the following way: we use the depth-first search algorithm to walk the graph in post-order. This allows us to start with sink nodes and work our way up to source nodes (which are all clock sources), while adding an increasing number constraints on the clock frequency on the way. When the walk is complete, we obtain a list of possible clock frequencies and the associated µC configurations. Constraints can be easily added or removed by accessing the graph structure. The traversing operation is different for each type of vertex. For example, in the case of a divider: for all child configurations, multiply the clock frequency by the value of the divider and add that value to the configuration.

Given a hardware description, and user and/or application timing requirements, the reconfiguration tool will generate frequency-optimal hardware configurations. At compile time, this allows any application to have a small set of configurations, each with its own clock frequency, and to freely switch among them. Once the possible configurations are calculated for each application, the programmer can include code in the application or in the operating system (e.g., under the form of a service), that will switch from one hardware configuration to another.

While the offline character of our optimization scheme may be seen as a drawback, it is consistent with the spirit of embedded systems and WSNs, in which applications are very simple and fully determined in advance. Most WSN OSes are designed for a single application tightly coupled with the OS and have one fixed and largely predetermined hardware configuration. Our tool allows for multiple prede-
3.3. Evaluation

Our goal is to achieve energy savings in WSNs by optimizing the frequency of the $\mu C$ for each application. One important consequence of frequency optimization is that it avoids unnecessary wake-ups from LPMs. As a preliminary evaluation of our scheme, we compared the amount of energy consumed by a simple application in TinyOS 2.x and the same application using our optimization scheme.

As shown in Figure 5, we consider an application that sends a temperature sample every $\Delta$ time units. In an ideal situation, the device would not wake up from LPM between two packet transmissions: $\Delta = \delta$. However, in TinyOS, $\delta \approx 1 \sec$ for all values of $\Delta \geq 1 \sec$. Moreover, the hardware timer is configured such that the overflow of the timer’s counter will also issue an interrupt, regardless of the values of $\Delta$ and $\delta$. This counter overflows every $\approx 2 \sec$ (16-bit counter driven by 32 kHz crystal) and while one may think the 1 s interrupts will mask the overflows, this is not true. In reality, the 1 s alarm lacks accuracy, therefore both interrupts are received and the interrupt handlers executed with a small LPM time in between. The inaccuracy is due to the fact that periodic timers are only periodic in software: the timers to minimize error. The maximum time a $\mu C$ can go without waking up is dependent on the width of its timer’s counter and on the minimum timer frequency. For the usual TI MSP430 configuration (ACLK on 32 KHz quartz crystal, 16-bit counter and clock dividers at their maximum), this time is 128 seconds. Therefore, when $\Delta > 128 \sec$, we will have $\Delta > \delta$ even with optimal frequency.

To calculate the energy consumed in both cases, we used component data sheets [4], [6], measures from the WSim hardware platform simulator [1] and the performance overview presented in [10]. Our results are presented in Table 1. As expected, there is no improvement for high duty cycle applications ($\Delta \leq 1 \sec$), but it rapidly increases to reach $\approx 11.12\%$ for low duty cycle applications.

<table>
<thead>
<tr>
<th>$\Delta$</th>
<th>Useless IRQs per $\Delta$</th>
<th>Inevitable IRQs per $\Delta$</th>
<th>Energy Saved (%)</th>
<th>Avg. Error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 sec</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2.555%</td>
</tr>
<tr>
<td>30 sec</td>
<td>44</td>
<td>0</td>
<td>5.598%</td>
<td>2.555%</td>
</tr>
<tr>
<td>1 min</td>
<td>39</td>
<td>0</td>
<td>7.455%</td>
<td>2.555%</td>
</tr>
<tr>
<td>15 min</td>
<td>1.349</td>
<td>7</td>
<td>10.7706%</td>
<td>N/A</td>
</tr>
<tr>
<td>30 min</td>
<td>2699</td>
<td>14</td>
<td>10.9432%</td>
<td>N/A</td>
</tr>
<tr>
<td>1 hour</td>
<td>5399</td>
<td>28</td>
<td>11.0315%</td>
<td>N/A</td>
</tr>
<tr>
<td>1 day</td>
<td>129599</td>
<td>674</td>
<td>11.1174%</td>
<td>N/A</td>
</tr>
<tr>
<td>1 week</td>
<td>907199</td>
<td>4723</td>
<td>11.1206%</td>
<td>N/A</td>
</tr>
<tr>
<td>1 month</td>
<td>3887999</td>
<td>20249</td>
<td>11.1210%</td>
<td>N/A</td>
</tr>
<tr>
<td>1 year</td>
<td>46655999</td>
<td>24299</td>
<td>11.1211%</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 1: Energy saved as compared to TinyOS

4. Conclusion and Current Work

Our work introduces two complementary methods to reduce energy consumption in WSNs. The goal is to save energy while facilitating the collaboration between a very rich hardware platform and the user or the application. Ongoing work deals with further developments of the reconfiguration software: creating a code generator, including other $\mu C$ peripherals in the frequency optimization graph.

A second activity targets a more thorough evaluation of our work. This includes testing it on real sensing devices and better illustrating its progress over current schemes. For example, the preliminary performance evaluation does not illustrate the improvement of our scheme over a well-parametered tickless WSN OS. In a tickless OS, a list of timers is ordered according to their expiration time and the device sets one hardware timer to the nearest deadline. Our scheme avoids unnecessary processing caused by periodic timers by using all available hardware timers.

References