Selection of Computing Elements for Energy Efficiency in Wireless Sensor Networks using a Statistical Estimation Method

Steven Corroy[▷] Jan Beiten[▷◦] Junaid Ansari[◦] Heribert Baldus[▷] Petri Mähönen[◦]

[▷]Philips Research, Distributed Sensor Systems
HTC 37-51, NL-5656AE, Eindhoven, The Netherlands, steven.corroy@philips.com

[◦]Department of Wireless Networks, RWTH Aachen University
Kackertstrasse 9, D-52072, Aachen, Germany, jan@mobnets.rwth-aachen.de

Abstract

A wide range of wireless sensor network applications are characterized by local processing of the sensed data and only meager data communication requirements. Since sensor nodes are battery powered and wireless communication bears a high energy cost, data transmission can be traded for on-node-computing in order to extend the lifetime of a node as well as the network. Furthermore, the energy consumption can be reduced significantly by selecting and realizing the application on an appropriate processing element. In this article, we propose a new statistical technique for energy consumption estimation for a specific application on various platforms. We have empirically verified the methodology on various classes of embedded processors commonly used for sensor nodes. The methodology is also applicable to multiprocessor platforms. Our solution is not only capable of achieving high degree of accuracy but also facilitates the application developer to evaluate different platforms without actually implementing the application on each of these platforms. Our experimental evaluation results on different platforms will help understand the implications of using different processing elements and their effects on the lifetime of a wireless sensor network.

Keywords: Wireless sensor networks; energy efficiency; energy consumption estimation.

1 Introduction

Wireless Sensor Networks (WSNs) have a wide range of applications with long operational lifetime requirements. Since WSN nodes are generally battery operated, achieving long lifetime just by changing batteries is either too cumbersome or impossible. Therefore, efficient use of the available battery source becomes an important issue in WSNs. A sensor node essentially includes a microcontroller, a radio transceiver and a few sensors. The usage of all the components needs to be optimized for battery conservation. Radio communication exhibits the highest energy consumption budget in many applications while others are dominated by computations. Low power MAC protocols are designed to optimize the use of radio resource by periodically turning on/off the radio in order to save energy, while the radio is inactive. One class of MAC protocols is the common schedule based protocols like S-MAC [17] and its variants. These protocols coordinate the active periods of the nodes so that messages are sent only when all the nodes are active at a common wake-up period. Another common class of MAC protocols is the preamble sampling MAC protocols like B-MAC [13] and its variants, where nodes sleep and wake-up asynchronously and rely on long preambles to signal the upcoming data.

While MAC protocols play an important role in applications where data communication is dominant in terms of energy consumption, these have little influence in computationally intensive applications with meager communication requirements. In order to meet the computing requirements, many types of microcontrollers with different characteristics are used. These include ASIC (Application Specific Integrated Circuit), ASIP (Application Specific Integrated Processor), RISC (Reduced Instruction Set Computer) and CISC (Complex Instruction Set Computer). A DSP (Digital Signal Processor) is an ASIP specialized in digital signal processing. RISCs and CISCs are both GPPs (General Purpose Processors). Each of these different classes of processing elements suits better to different application demands. For instance, a DSP is more suitable for performing signal processing, e.g. computing Fast Fourier Transformation (FFT), while it is inefficient for controlling Serial Peripheral Interface (SPI) communication. In order to select the most power efficient processor, it is necessary to determine

the anticipated energy consumption of a processing element executing the application.

Code execution depends on several factors such as acquiring sensor data or other external inputs. Thus the energy consumed can only be measured at runtime. For a runtime measurement, the application needs to be implemented on the WSN platform. The implementation efforts are relatively high due to different instruction sets and processing element specifics. It is neither cost nor time efficient to implement an application on all the available WSN platforms in order to determine the optimal one. The number of implementations required can be lowered with the experience of a code developer, but still the number of possible platforms remains relatively high. Therefore, a technique to determine power consumption estimation without actual implementation effort is desirable.

Simulators such as a circuit simulator cannot produce accurate results for energy consumption, since they cannot completely simulate the wireless sensor node environment. Furthermore, simulators are hardware specific and are not available for each platform. The effort required to implement an application on a simulator is very similar to the effort required to implement the application on a real hardware platform. Therefore, it is easier to estimate energy consumption on a higher level independent from the actual implementation. The next higher level of abstraction that allows energy estimation is the code level. We describe in the following a methodology for estimating energy consumption of a specific application on a specific WSN platform at the code level. Our method is also applicable to platforms with multiple processors as we describe in the later sections. This article is an extended version of earlier paper [1], published in the International Conference on Sensor Technologies and Applications, SENSORCOMM 2008.

The rest of the article is organized as follows: Section 2 presents the related work in this area, Section 3 and 4 detail our solution, Section 5 presents the results that we obtained by applying our methodology on a single processing element on two different classes of application requirements. Section 6, describe the methodology applicable on multiprocessor architecture. Finally, in Section 7 we conclude the article.

2 Related Work

The energy consumption of a specific application depends on the hardware platform. The suitability of the hardware to the required software functionality allows to achieve energy efficiency. The energy consumption estimation for computing elements has been an on-going research issue [3, 5, 9]. However these studies concentrate on determining the energy consumption on one specific platform for a specific application.

Feinstein *et al.* [5] have developed a method to measure power consumption using a single measurement point. Their method assigns a unique ID to each process and logs in the corresponding real time power consumption and the execution time of the task. Although this approach enables a precise estimation of the energy consumption at the task-level, it requires a sensor node, for carrying out the measurements and an implementation of the application on each of the target platforms.

Bircher *et al.* [3] proposes to use event counters (e.g., DMA accesses or interrupts) for modeling the power consumption of a whole computing platform. From specific events to the microcontroller, their approach derives values for the rest of the platform e.g. I/O power consumption. Similar to the method by Feinstein *et al.* [5], this approach also requires the implementation of the application on each target platform.

PowerTOSSIM [14] is a tool for estimating the power consumption of applications developed in the TinyOS operating system environment for the supported platforms. Since it is a high level power consumption estimator, the accuracy is not high enough. PowerTOSSIM is based on the TinyOS simulator TOSSIM, which lacks the ability to model the simulations accurately owing to its inability to handle preemption of tasks and lackness of precise timings the execution of different functions [7], PowerTOSSIM also remains inaccurate.

A. Dunkels *et al.* [4] developed a software based technique for measuring online power consumption of a certain application developed for Contiki operating system running on Moteiv Inc.'s TmoteSky sensor node platform. Since the power consumption of the node in each state is known beforehand, a sensor node can estimate its own energy consumption by time-stamping each of its states. This technique however requires a full implementation of the application on the supported platform and hence makes it a time-taking and tedious job for the code developer.

Circuit simulators for energy consumption can give cycle accurate results but require long execution time. Niar *et al.* [9] proposes to combine statistical simulation to circuit simulation. It generates a short synthetic program representing the original application using statistical values, e.g., instruction distribution or cache miss rate. It then simulates the short synthetic program, giving substantial speed gains with only an average error of 3.8%.

The work by H. Joe *et al.* [6] aims at designing an accurate instruction level power estimator for sensor networks. They developed a power estimation tool using a machine instruction level simulator and correspondingly an energy consumption estimation module. The energy consumption estimation module is instructed by the instruction level simulator for profiling the energy consumption during the run time of the application. A post-processing module handles

the adjustments for the function call/returns and the I/O accesses to estimate the per node energy consumption in function calls and hardware components. Again this provides a very low level power consumption estimation which requires a pre-implementation of the application. Furthermore, for large simulation of sensor network applications, this approach is very slow.

In contrast to the approaches described above for power consumption estimation, we aim at estimating and evaluating the power consumption of a particular application on different platforms without implementing it. We trade-off estimation precision for ease of use. In order to evaluate and verify our work, we used three different microcontrollers in our experiments which represent the three different classes of computing elements. This includes the 8051 architecture [8], the MSP430 architecture [16] and the Coolflux architecture [10].

The 8051 is an 8-bit CISC type processor with Harward architecture as part of the CC2430 [15]. It was one of the first embedded processors that included CPU, RAM, ROM, I/O, interrupt logic and timers in a single package. Despite its design age, 8051 is still in use for many embedded system applications. The MSP430 is a 16-bit RISC with von Neumann processor architecture. Its design was specifically developed as a RISC architecture with very low power consumption. There is a wide selection of MSP430s in the market, providing different combinations of peripherals.

The NXP CoolFlux is a 24-bit DSP with features such as pipelining, MMU, register file structure and a very efficient Multiplication/Accumulation (MAC) implementation.

3 Methodologies for Power Consumption Estimation

Calculating the required number of instructions for a specific application and analyzing the required energy consumption enables to compare the energy efficiency of different computing elements. While considering different processing platforms, the operating clock frequencies vary significantly from one platform to another. It implies that the possible number of executable instructions within a certain time-frame varies correspondingly. For all the processing elements, Million Instructions Per Second (MIPS) can be calculated and is used as the common performance reference. However, the computational power consumption of a single instruction differs much from one platform to another. Certain instructions may require only one cycle on a RISC platform and take several hundred cycles on a CISC platform. Therefore, MIPS is not a reliable means to derive energy comparison. Other benchmarks are not widely available for embedded systems and are bound to only a small set of applications due to a different combination of instructions. So a new approach needs to be devised in order to analyze the energy consumption with a wider focus than conventional benchmarking. In the following, we present a method for estimating power consumption of an application utilizing the code level. It consists of two steps. First the code of a new application is divided in short blocks of code. We call *weight* a physical measure applying to a block of code, e.g., run time or energy consumption. In the following we explain the methodology restraining the meaning of *weight* to energy consumption for clarity purpose. Second, the empirical weight of the blocks of code is multiplied with their number of occurrence which gives an estimate of the overall power consumption of the application. Next we explain how to choose the relevant blocks for splitting the application and how to calculate their weights.

3.1 Blocks of Code Granularity

A high level programming language is composed of different instructions. An application is a sequence of instructions. The shortest atomic element of an application is thus an instruction. But an application can also be described as a sequence of different combination of instructions. For estimating an application we split it up into a large number of blocks of code. A block of code is a combination of instruction. The size of a block of code is determined by the number of instructions it contains. When splitting an application all blocks of code have the same size. The first step of our approach is to identify the relevant blocks independently. The granularity of the blocks is decisive for the accuracy and the complexity of the estimation. Long blocks of code give more accurate results because they contain more dynamic effects, like for instance compiler optimization. On the other side, increasing the length of the blocks of code increases quadratically the complexity of the method. Assume l is the length of the blocks of code and k the number of possible blocks (all different combination of instructions of size l), increasing to $2l$ the length of the blocks increases to k^2 the number of possible blocks. Our experiments show that single instructions ($l = 1$) such as additions, jumps and explicit memory accesses are flexible and efficient processing blocks for multiple platforms.

3.2 Blocks of Code Weight

The second step is to determine the weights assigned to blocks of code. A simple approach is to measure the energy consumption of all single blocks on each platform. We implement a program for each block of code (e.g. an addition or jump instruction) containing a sequence of the block in a row. We measure the energy expended by this program and derive the energy consumption of the block of code. Applying the calculated weights to real world WSN applications show that the results have a 95% confidence interval within

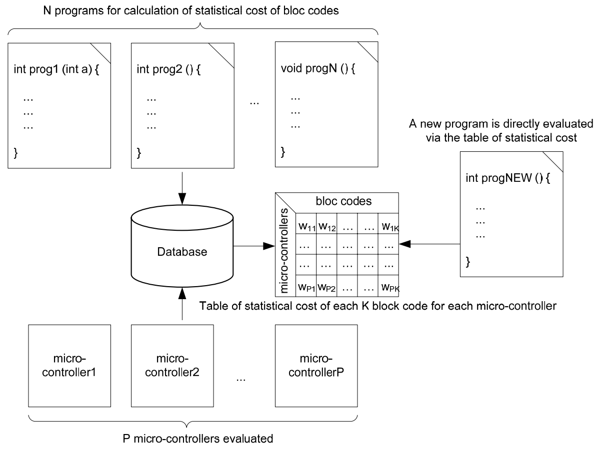

Figure 1. Overall system for optimal computing element selection.

[-27% 23%]. The imprecision of the results is caused by the compiler optimizations and runtime effects. In order to improve the accuracy of the results, a statistical approach is introduced which decreases the influence of the compiler optimizations.

3.2.1 System Description

It is assumed that the effects of the compiler optimizations in reducing the energy consumption are statistically foreseeable. Instead of using the measured energy consumption of constant-sized-blocks-of-code to determine the weights of the blocks, we measure the energy consumption of several representative WSN applications and derive statistically the cost of each block of code. This approach has the advantage of taking into account much of the dynamic effects occurring due to compiler optimizations while keeping shorter blocks of code for fast and flexible power estimation. Our overall system is illustrated in Figure 1. We implement on P platforms N applications {prog1,...,progN} such as simple instructions concatenation and also more sophisticated algorithms such as FFT or field calculations. The power consumption and the runtime of those applications are empirically measured. We maintain a database with tables T_1 and T_2 for storing:

1. The runtime t_{np} and the energy consumption e_{np} of each application for each platform with $n \in [1, N]$ and $p \in [1, P]$.

$$T_1 = \begin{pmatrix} (t_{11}, e_{11}) & (t_{12}, e_{12}) & ... & (t_{1P}, e_{1P}) \\ (t_{21}, e_{21}) & (t_{22}, e_{22}) & ... & (t_{2P}, e_{2P}) \\ ... & ... & ... & ... \\ (t_{N1}, e_{N1}) & (t_{N2}, e_{N2}) & ... & (t_{NP}, e_{NP}) \end{pmatrix}$$

2. The number of occurrence c_{kn} of each blocks of code in each application (assuming K possible blocks of code) with $k \in [1, K]$ and $n \in [1, N]$.

$$T_2 = \begin{pmatrix} c_{11} & c_{12} & ... & c_{1K} \\ c_{21} & c_{22} & ... & c_{2K} \\ ... & ... & ... & ... \\ c_{N1} & c_{N2} & ... & c_{NK} \end{pmatrix}$$

We use a linear programming solver to find the weights w_{kp} of each block of the code on each platform with $k \in [1, K]$ and $p \in [1, P]$. The problem to solve has the following form for energy consumption:

$$\text{for each } n \in [1, N],\ p \in [1, P],\ \sum_{k=1}^{K} c_{kn}.w_{kp}^{e} = e'_{np} \quad (1)$$

and for run time

$$\text{for each } n \in [1, N],\ p \in [1, P],\ \sum_{k=1}^{K} c_{kn}.w_{kp}^{t} = t'_{np} \quad (2)$$

where e'_{np} is the statistical energy consumption and t'_{np} the statistical run time of a specific program on a specific platform ($e'_{np} \geq e_{np}$ because compiler optimizations tend to remove or regroup instructions, thus making the code more efficient and more compact than originally written). We put the following constraints which represent the fact that the energy consumption of one operation must be positive:

$$\text{for each } k \in [1, K],\ \ p \in [1, P],\ w_{kp}^{e} > 0, w_{kp}^{t} > 0 \quad (3)$$

Finally, we minimize the sum of the squared relative error between e'_{np} and e_{np} among all the applications:

$$\text{for each } p \in [1, P],\ \sum_{n=1}^{N} \left(\frac{e'_{np}}{e_{np}} - 1 \right)^2 \quad (4)$$

Respectively, we minimize the error in terms of time

$$\text{for each } p \in [1, P],\ \sum_{n=1}^{N} \left(\frac{t'_{np}}{t_{np}} - 1 \right)^2 \quad (5)$$

From now on, if one wants to estimate the power consumption (respectively the run time) of a new application, one just needs to count the number of occurrences of each block of the code in the program (e.g., with a parser) and to multiply them with their statistical cost. We maintain a database table T_3 for the weights of the form

$$T_3 = \begin{pmatrix} (w_{11}^t, w_{11}^e) & (w_{12}^t, w_{12}^e) & ... & (w_{1P}^t, w_{1P}^e) \\ (w_{21}^t, w_{21}^e) & (w_{22}^t, w_{22}^e) & ... & (w_{2P}^t, w_{2P}^e) \\ ... & ... & ... & ... \\ (w_{K1}^t, w_{K1}^e) & (w_{K2}^t, w_{K2}^e) & ... & (w_{KP}^t, w_{KP}^e) \end{pmatrix}$$

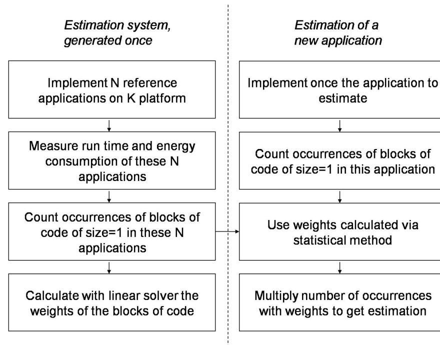

Figure 2. Overview of estimation methodology.

Instruction	*8051 8 bit*	*8051 16 bit*	*8051 float*	*CF 24 bit*
Summation	16.17	53.79	158.07	0.01
Subtraction	21.12	49.5	816.42	0.10
Multiplication	31.68	86.46	611.49	0.85
Division	76.23	127.05	712.80	2.27
IF Then	4.95	42.24	273.57	0.62
IF Then ELSE	138.93	111.21	2749.89	0.00
Loop Repetition	5.61	4.95	28.38	0.00
Array access	9.57	0.00	106.26	0.10
Assignment	20.13	29.70	130.68	0.00
Modulo	667.92	714.78	113.85	0.61

Table 1. Statistical energy consumption of processing elements in nano Joules.

We illustrate a global overview of the methodology in Figure 2 The left part of the schema illustrates how the estimation system is built. This computation work is processed only once to generate all the statistical weights. The right part illustrates the work which take place when estimating a new application, this is the part that needs to be fast and simple.

3.2.2 Results

The results for finding the weight of blocks of code are shown in Table 1 (note that only a subset of all possible blocks of code is presented). Using those weights, we estimated the energy consumption of several other applications (two of them are presented in more detail in Section 5) and verified the results with measurements. Our method shows significantly better results for all the platforms as shown in Table 2. The algorithm is reliable and accurate for both computational time and computational energy consumption.

3.2.3 Multiprocessor Approach

The evaluation method that we developed is not only capable to forecast energy consumption for a single computing element but also for a multi-processor platform. We assume a proper mapping of the application on the different processors. We estimate the energy consumption of the code running on each processing element separately. Then, we calculate the power required for communication between all processors (more details in Section 4) and add it to the previously estimated processor consumptions. This approach enables to find the optimal WSN platform for a specific application when it requires more than one processor (the case study in Section 5.2 highlights it). For example combining a CISC for controlling the peripherals and a DSP for computing may be beneficial as long as the overhead in terms of internal communication and double-powering does not counter-balance the gains in terms of processing efficiency.

Processing element	*Word-Width*	*95% Confidence interval*
8051	8	[-13 %;13 %]
8051	16	[-14 %;14 %]
MSP430	16	[-13 %;13 %]
Coolflux	24	[-8,3 %; 8,3 %]

Table 2. Confidence intervals for energy estimation on 8051, MSP430 and Coolflux.

4 WSN platform Power Consumption

In Section 3, we presented a method to determine the power consumption of a specific application on a specific computing element. Because our main interest is in WSN applications, we propose a method to estimate the energy consumption of the whole WSN platform using the forecast for the computing element. We first indicate the dominant factors for WSN platforms with respect to power consumption.

4.1 Power Consumption Factors in WSN Platforms

Data Acquisition is the sampling of data from a sensor of the processing element. It contains the power consumption of the sensor (taken from the data sheet or measured once) and the energy needed by the processing element to control the communication interface. This last value can be measured one time for each platform and reuse for any new application to be benchmarked.

Computation is evaluated as described in Section 3.

Power Mode Change is the action for the processing element to go from active mode to sleep mode (and vice-versa). It is specific for each platform, but only have to be measured once for each platform.

Power Down Mode or Sleep Mode represents the energy consumed by the processing element while sleeping. Usually processing elements provide a set of power down modes, diverging with available peripherals, timers and memory. Each power mode was measured once per platform.

Task	*Coolflux*	*8051*	*MSP 430*
Data acquisition	654	306	170
Computation	0.05	291	46
Power mode changes	0.00	585	158
Idle power	1211	565	794
Overall	1866	1747	1167

Table 3. Power consumption of the first case (in μJ) on the evaluated platforms.

Wireless Communication is predicted using methods described in [2] [11].

4.2 Methodology

In order to evaluate the total power consumption of a WSN platform, we add the individual power consumption of all those factors together. In order to predict the time that an application spends in sleep mode or to foresee the number of power mode changes, we use a periodic model which is used in most of the WSN applications and consists of six phases: 1) Power-up mode, 2) Sample data from the sensor, 3) Process data, 4) Transmit data, 5) Power down mode and 6) Sleep mode. The total period of one cycle is chosen by the programmer and the sleep time is determined by subtracting the foreseen time for data sampling, computation, transmission and power mode change.

5 Case Studies

In this section, we describe the performance results obtained from our implementation. We carried out the implementation and evaluation of two very typical sensor network applications and discuss their energy efficiency on different processing elements. We consider single processor as well as multiprocessor solutions in this regard.

5.1 Applications with Moderate Computational Requirements

In order to provide a concrete example on how to apply our method, the first application that we consider is acceleration sensing. An accelerometer acquires 16 bit values at a frequency of 50Hz. If the values are greater than a specific threshold, an alert is transmitted to a hub. The computing element is shut-down between each sample. Table 3 shows the break-down of power consumption of the application on various platforms (over 1s). In the following, we describe the various sources of energy consumption in the sample application.

5.1.1 Data Acquisition

Out of the previously described classes of processing elements, GPPs are the most efficient for data acquisition. Since GPP platforms have integrated peripheral interfaces such as UART and SPI, these can communicate with the sensors very efficiently. Both MSP430 and 8051 have an integrated SPI. The activation of the interface consumes only 60 μW of power, which is insignificant compared to the active power consumption of the two processors. In the sample application, the processing elements cannot utilize the idle time during the data acquisition for computations as there is only a single task running. Furthermore, energy savings cannot be obtained just by switching into the sleep mode as the time interval is too short.

Coolflux platform requires a significant amount of energy for data acquisition owing to the absence of a dedicated serial interface and can only emulate the serial protocol in the software (bit-banging). This causes very high acquisition time and current consumption on Coolflux. Therefore, a Coolflux is the least efficient processing element during data acquisition.

5.1.2 Computation

From the computational point of view, Coolflux is the most energy and time efficient processor element in our study. Coolflux contains two Arithmetic Logical Units (ALUs) and can therefore execute multiple instructions in a single cycle. In the 16-bit data processing application, Coolflux can work on its 24-bit native word-width. 8051 and MSP430 require a comparable run-time for computations. 8051 has an 8 bit ALU. Therefore, it requires additional cycles for each 16 bit operation. MSP430 as a RISC processor, is more cycle effective and additionally can work at its native word-width. Therefore, the power consumed by MSP430 is significantly lower for this application. Overall, Coolflux is magnitude times more energy and time effective than the CC2430 and the MSP430.

5.1.3 Power Mode Changes

8051 requires a significant amount of energy for a single power mode change. For a sampling rate of 50 Hz, the energy spent in mode transitions is more than the computation itself. 8051 spends about 94 % of its time in power down mode and 4 % of time in power mode changes, but through the low energy consumption in power down mode, most of the energy is spent in the mode changes. MSP430 requires about the same time to switch to sleep mode, but through its lower current, the energy consumption for that is significantly lower. Contrary to the GPPs, Coolflux platform decouples the clock network and does it within a few cycles. The energy consumed for the power mode change is only 0.16 nW.

5.1.4 Energy Consumption in Sleep Mode

Coolflux can change its power mode very fast. On the other hand, the power down mode itself is less effective. Coolflux consumes about 25 % of its active power in sleep mode. By spending a long time in sleep mode, the overall power spent in sleep mode exceeds the power spent in all the other operations in the sample application.

The power down modes of MSP430 and 8051 consume about the same amount of energy. 8051 also supports more effective power modes than the chosen power mode in our application, but only supports half of its memory. MSP430 also supports other power down modes but since it has a relatively large switching time, these are not used in our sample application.

5.1.5 Efficiency depending on the Sampling Rate

We change the accelerometer sampling rate to understand its impact on the overall energy efficiency. Results are shown in Figure 3. Note that CC2430 relates to a platform containing a 8051. If the sample rate is very low, 8051 turns out to be the most energy efficient processing element, because of its low power consumption in power down mode. MSP430 is more effective in data acquisition and power mode changes. Therefore, starting at a sample rate of 18 Hz, MSP430 becomes more efficient for our sample application.

It may be noted that Coolflux becomes more power efficient than 8051 beyond a certain sampling rate. Although the Coolflux is less effective in data acquisition, its ability for computation and energy effective power mode changes compensates it. Overall, the MSP430 is relatively the most preferred processing element for high sampling rates.

The maximum supported sampling rates are 1310 Hz for Coolflux, 2564 Hz for 8051 and 3322 Hz for MSP430. A lower maximum sampling frequency of Coolflux is caused by its inability for higher data acquisition rate. The difference between 8051 and MSP430 is caused by the computational ability of MSP430.

5.1.6 Efficiency depending on the Computation

Depending upon the computational power, the efficiency of the processing elements is also evaluated. We modify the number of calculations in the application. Figure 4 shows the energy consumption depending on the proportion of computations compared to the initial application. It allows estimating, how much computation would make it worth to implement the algorithm on another processing element. 8051 becomes less effective than Coolflux for the application if the number of computations is doubled. If the amount of computation is increased by a factor of 20, Coolflux becomes more efficient than MSP430.

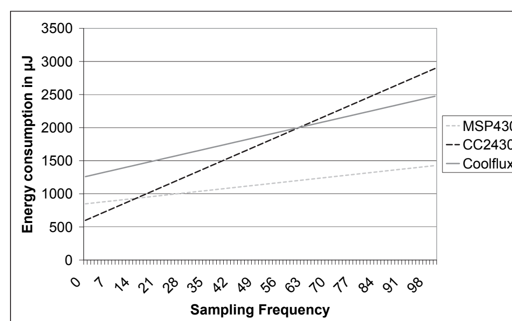

Figure 3. Influence of the sensor sampling rate on the energy consumption of the 8051 (inside CC2430), MSP430 and Coolflux.

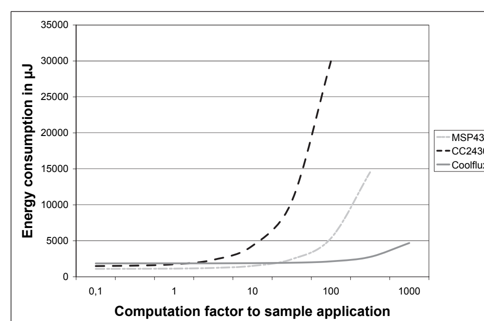

Figure 4. Influence of the computational load on the energy consumption of the 8051 (inside CC2430), MSP430 and Coolflux.

In conclusion, the presented example highlights the attributes of different classes of processing elements. As a DSP is more effective for calculations, GPPs are more effective during data acquisition. The sample application considered above is very simple and requires only a very limited number of computations. Also, the number of computations required are clearly dominated by conditional jumps and not by arithmetic operations. Therefore, a simple processor efficiently supporting fast switching modes and low energy consumption in power down mode is the best choice for this kind of application. Multiprocessor concepts are not suitable, since the complexity of the algorithm is low and the energy spent in power down mode exceeds all the possible energy improvements.

5.2 Applications with High Computational Requirements

A second study case is the recognition of heart rate from ECG data using a real-time algorithm. The heart rate must

Task	*8051*	*MSP430*	*MSP430+Coolflux*
Data acquisition	7.5	3.0	3.0
Computation	6	3.9	0.1
Wireless communication	1.2	1.3	1.2
Serial communication			0.1
Power mode changes	5.9	2.4	2.4
Idle power	0.0	0.2	1.3
Overall	20.6	10.8	8.1

Table 4. Power consumption of the second case (in mJ) on the evaluated platforms.

be sampled at high frequency to obtain reliable results. The amount of computations is high because of the real-time nature of the algorithm. The heart rate calculation is sent regularly to a hub. We evaluated an in-house developed heart rate analysis code using a sampling rate of 500 Hz. It calculates the heart beat using an algorithm described in [12]. This algorithm does not store all the heart beat samples and therefore requires little memory. The algorithm is dominated by comparison operations and 32 bit calculations. The application is divided into data acquisition, computation and wireless communication.

A comparison of a platform combining MSP430 and Coolflux is shown in Table 4 (over 1s). A stand alone Coolflux solution is not appropriate, because of its ineffective data acquisition. On the combined platform, data acquisition is done on MSP430 because there is no ADC on Coolflux. Due to the high sampling rate, it takes a significant amount of time and energy. The multiprocessor solution has the advantage to allow higher sampling rates than the single processor solution. The computations are very effective on Coolflux DSP. Indeed the 12-bit data gets double after multiplication and fits very well to the 24-bit architecture of Coolflux unlike MSP430 or 8051. Therefore, the Coolflux is magnitude times more effective than MSP430 or 8051. Altogether with the multiprocessor approach, the power consumption is reduced by a factor of 28%. The proposed multiprocessor solution benefits from the strenghs of both classes of processing elements, namely the computational ability of DSPs and the fast data acquisition of GPPs, but this at the expense of size and complexity.

6 Multi-processor Scheduling

It is evident from the previous section that in some cases, an application can be implemented more energy efficiently on multiple specialized processing elements rather than on a single multi-purpose processing element. This leads to finding the best combination of the selected hardware. Simultaneously, it is required to split-up the application into tasks that are can be assigned to the selected hardware. The hardware selection and the correspondingly task scheduling is done mostly manually, which remains inefficient. With a wider range of energy efficient processing elements and furthermore their distributed nature such as in wireless sensor networks, the task becomes increasingly complex and unmanageable.

In Section 3, we have developed an approach that can be used to support power efficient implementation of a given application on multi-processor hardware. Tasks such as data acquisition, performing calculations, data communication, power mode switching and energy consumption in idle mode are the fundamental blocks of the overall energy consumption on a sensor node. In our approach, after the system designer specifies an application, the code is broken down to granular blocks and weights are calculated for each of the block on all the processing elements under consideration. In addition to the application related facts, the designer also specifies timing and data dependency of the application tasks. The hardware and software requirements are formulated as conditions. Each combination of hardware with the regarding scheduling has to meet all the conditions. The sum of the weights of each block shows the total energy consumption of a working solution. The minimum of this sum shows the most energy efficient solution of the evaluated hardware.

The solution space for the described conditions grows exponentially with the number of considered processing elements and tasks. In order to calculate the energy consumption of a wide range of possible solutions, NP solvers are used. Integer Linear Programming (ILP) solvers can directly be used to calculate the most energy efficient solution. In the following, we describe the detailed formulation of the multi-processing scheduling problem to be solved through ILP.

6.1 Attributes of the Tasks

Each task requires a set of constraints in order to describe their behaviour and to allow the formulation of equations for their scheduling.

$E_{\beta,\gamma}$	Energy consumed by task β on hardware γ
$T_{\beta,\gamma}$	Average time required per second for task β on node γ
T_{β}	Average maximum execution time required for task β
$C_{\beta 1,\beta 2}$	Number of exchangeable bits from task $\beta 1$ to $\beta 2$
Ξ	Number of all tasks to be scheduled
O_{β}	Average number occurrence of task β

$E_{\beta,\gamma}$ describes the energy consumed by task β on hardware γ. Also the run-time of a task $T_{\beta,\gamma}$ on a certain hardware has to be ascertained. It is referenced against the

scaled real-time constraint T_β. These values are scaled to the required time per second. This allows to consider the number of occurrences and the elapsed time on the hardware simultaneously. If the number of exchangeable bits $C_{\beta 1,\beta 2}$ for a task is greater than zero, both parts are dependent on each other. Additionally, this value is used to calculate the energy consumption cost of the communication task.

6.2 Attributes of the Networked Nodes

The networked nodes require a set of attributes in order to distribute the hardware according to the specification.

$S_{\alpha,\beta,\gamma,\delta}$	Task β is scheduled on node α on hardware γ with ID δ
Υ	Available sensor nodes
Δ	Available hardware IDs

The variable $S_{\alpha,\beta,\gamma,\delta}$ describes the scheduling of task β on hardware γ with hardware ID δ to the node α. Furthermore, the scheduling implicitly contains a hardware selection γ. It is possible to have multiple instances of γ in the network node and these instances are numbered with δ.

6.3 Attributes of the Hardware

The hardware requires attributes in order to describe the hardware for the solver and checking the suitability of specific tasks. The hardware attributes optimize the hardware for the constraints given by the tasks and the network nodes' attributes.

EI_γ	Energy consumed by hardware γ in idle mode
$CC_{\alpha 1,\alpha 2,\gamma 1,\gamma 2,\delta 1,\delta 2}$	Communication cost between hardware $\gamma 1$ and $\gamma 2$
Γ	Available hardware
EC_γ	Energy consumption of power mode change on hardware γ

The idle state energy consumption EI_γ becomes significant when a processing element is used very rarely. Each hardware has certain available peripherals $AP_{\gamma,\epsilon}$. The communication cost of two hardware elements $CC_{\alpha 1,\alpha 2,\gamma 1,\gamma 2,\delta 1,\delta 2}$ depends on hardware location. EC_γ describes the energy required to change power mode on hardware γ.

6.4 Computing the Overall Energy Consumption

Integer Linear Programming requires an objective function that is either maximized or minimized. The term to be minimized in our case is the energy consumption of the overall network. It consists of the energy consumption of all the tasks running on different nodes. The run-time and real-time constraints are applied to the summed energy consumption formulation in order to obtain the optimum.

$S_{\alpha,\beta 1,\gamma,\delta 1}$ is used as Boolean including the scheduled elements. The first sum describes the energy consumption of the tasks.

$$\sum_{tasks} = \sum_{\alpha_\epsilon \Upsilon,\beta_\epsilon \Xi,\gamma_\epsilon \Gamma,\delta_\epsilon \Delta} S_{\alpha,\beta,\gamma,\delta} E_{\beta,\gamma}. \quad (6)$$

The second sum contains the energy consumed in idle mode. Therefore the average run-time in idle mode is calculated by subtracting the time spent in active mode. This is multiplied by the energy of the elements in idle mode

$$\sum_{idle} = \sum_{\alpha_\epsilon \Upsilon,\gamma_\epsilon \Gamma,\delta_\epsilon \Delta} (1 - \sum_{\beta_\epsilon \Xi} S_{\alpha,\beta,\gamma,\delta} T_{\beta,\gamma}) EI_\gamma. \quad (7)$$

The next sum is the communication costs between the tasks. Again $S_{\alpha,\beta 1,\gamma,\delta 1}$ is used as Boolean to exclude not scheduled tasks, whereas $C_{\beta 1,\beta 2}$ describes the amount of communication and $CC_{\alpha 1,\alpha 2,\gamma 1,\gamma 2,\delta 1,\delta 2}$ the communication costs

$$\sum_{com} = \sum_{\alpha_\epsilon \Upsilon,\beta 1_\epsilon \Xi,\beta 2_\epsilon \Xi,\gamma_\epsilon \Gamma,\delta 1_\epsilon \Delta,\delta 2_\epsilon \Delta} C_{\beta 1,\beta 2} S_{\alpha,\beta 1,\gamma,\delta 1} * S_{\alpha,\beta 2,\gamma,\delta 2} CC_{\alpha 1,\alpha 2,\gamma 1,\gamma 2,\delta 1,\delta 2}. \quad (8)$$

The last sum describes the energy spent on task mode changes. The assumption of diverging sample rates of the different applications does not allow to combine tasks to save energy through a decrease of power mode changes.

$$\sum_{changes} = \sum_{\alpha_\epsilon \Upsilon,\beta_\epsilon \Xi,\gamma_\epsilon \Gamma,\delta_\epsilon \Delta} S_{\alpha,\beta 2,\gamma,\delta 2} O_\beta EC_\gamma. \quad (9)$$

All considered energy consumptions can be summed together in

$$\sum_{total} = \sum_{tasks} + \sum_{idle} + \sum_{com} + \sum_{changes} \quad (10)$$

The implementations constraints need to be formulated. For example equation 11 describes the real-time constraints of the problem. It is necessary to be sure, that the run-time of all scheduled tasks on the hardware is smaller than the real-time constraint of the task.;

$$\sum_{\alpha_\epsilon \Upsilon,\gamma_\epsilon \Gamma,\delta_\epsilon \Delta} S_{\alpha,\beta,\gamma,\delta} T_{\beta,\gamma} \leq T_\beta. \quad (11)$$

Additional constraints are required for computational restrictions, fixed hardware assignments, identification of hardware in a specific node and hardware restrictions. These are formulated analogously.

This formulation as ILP of minimization of the global energy consumption of the network, although not allready solved, provides some understanding of the problem. It highlights the trade-off between computation, communication, power mode changes and idle mode that need to be take in account to reach an optimal solution.

7 Conclusions and Future Work

In this article, we described a new method for estimating the power consumption of a particular application on different wireless sensor node platforms. The method involves slicing down a whole application into smaller granular blocks of code. We use a linear programming solver to determine the weights associated for each of the code block on each platform. Through a detailed case study, we analyzed the trade-offs among CISC, RISC and DSP approaches for WSN nodes and showed empirically that our method is accurate. We carried out the evaluation of our methodology on 8051, MSP430 and Coolflux processors, representing the three processor classes. Our method provides easy way to estimate the power consumption but trades-off the accuracy. However, it is worth noticing that our method achieved a worst-case accuracy of 86%. We evaluated an application with meager computing requirements as well as a computationally intensive application. Our scheme requires just a single code implementation of the application for determining the most power efficient computing element, which will help code developers easily select the most suitable platform. The presented scheme also allows efficient benchmarking of the processing elements and determining the most energy efficient element, thereby saving costs and implementation efforts. We have also presented an extension of the scheme to multi-processor architectures. We have applied linear programming approach to efficiently schedule different tasks on a multi-processor platform. Real world applications are typically realized on a number of network nodes rather than just on a single node. Therefore, a local energy optimal solution may not necessary be the global energy optimal solution. In order to find a globally optimized estimate, a system wide hardware selection and scheduling is required. It is possible to build up a system of linear conditions, which represent the restrictions for a practical WSN. Using linear programming, the global energy optimal implementation of a network can be calculated based on the local estimation provided by our methodology.

Acknowledgments

We would like to thank the financial support from E.U. (project IST-034963-WASP), Philips Research, Deutsche Forschungsgemeinschaft through the UMIC-excellence cluster and RWTH Aachen University.

References

[1] S. Corroy, J. Beiten, J. Ansari, H. Baldus, and P. Mähönen. Energy efficient selection of computing elements in wireless sensor networks. In *International Conference on Sensor Technologies and Applications (SENSORCOMM 2008)*, pages 312–318, 2008.

[2] M. Achir and L. Ouvry. QoS and energy consumption in wireless sensor networks using CSMA/CA. Technical report, Electronics and Information Technology Laboratory Atomic Energy Commission, 2005.

[3] W. Bircher and L. John. Complete system power estimation: A trickle-down approach based on performance events. In *IEEE International Symposium on Performance Analysis of Systems & Software*, April 2007.

[4] A. Dunkels, F. Österlind, N. Tsiftes, and Z. He. Software-based sensor node energy estimation. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 409–410, 2007.

[5] D. Feinstein, M. Thornton, and F. Kocan. System-on-chip power consumption refinement and analysis. In *6th IEEE Dallas Circuits and Systems Workshop on SoC*, 2007.

[6] H. Joe, J. Park, C. Lim, D. Woo, and H. Kim. Instruction-level power estimator for sensor networks. *ETRI Journal*, 30(1):47–58, February 2008.

[7] O. Landsiedel, H. Alizai, and K. Wehrle. When timing matters: Enabling time accurate and scalable simulation of sensor network applications. In *Proceedings of the 7th international conference on Information processing in sensor networks*, pages 344–355, 2008.

[8] I. S. MacKenzie. *The 8051 Microcontroller*, volume 4th Edition. Prentice Hall, 2001.

[9] S. Niar and N. Inglart. Rapid performance and power consumption estimation methods for embedded system design. In *7th IEEE Int. Workshop on Rapid System Prototyping*, pages 47–53, June 2006.

[10] NXP, http://www.coolfluxdsp.com. *Coolflux DSP*, 2004-05.

[11] L. Ouvry and M. Achir. Probabilistic model for energy estimation in wireless sensor networks. *Lecture Notes in Computer Science*, 3121/2004:157–170, 2004.

[12] J. Pan and W. Tompkins. A real time QRS detection algorithm. *IEEE Trans. On Biomedical Engineering*, 32, 1985.

[13] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *Proc. of SenSys*, pages 95–107, 2004.

[14] V. Shnayder, M. Hempstead, B. rong Chen, G. W. Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 188–200, 2004.

[15] Texas Instrument, http://focus.ti.com/docs/prod/folders/print/cc2430.html. *CC2430*, 1995-2007.

[16] Texas Instruments, http://focus.ti.com/docs/prod/folders/print/msp430f1611.html. *TI MSP430x1611 - Mixed Signal Microcontroller*, 2005.

[17] Y. Wei, J. Heidemann, and D. Estrin. Medium access control with coordinated adaptive sleeping for wireless sensor networks. *IEEE Trans. on Net.*, 12(3):493–506, June 2004.