Energy Bucket: a Tool for Power Profiling and Debugging of Sensor Nodes

Jacob Andersen and Morten Tranberg Hansen
Department of Computer Science
Aarhus University
Aarhus, Denmark
Email: {jacand,mth}@cs.au.dk

***Abstract*—The ability to precisely measure and compare energy consumption and relate this to particular parts of programs is a recurring theme in sensor network research. This paper presents the *Energy Bucket*, a low-cost tool designed for quick empirical measurements of energy consumptions across 5 decades of current draw. The Energy Bucket provides a light-weight state API for the target system, which facilitates easy score-keeping of energy consumption between different parts of a target program. We demonstrate how this tool can be used to discover programming errors and debug sensor network applications. Furthermore, we show how this tool, together with the target system API, offers a very detailed analysis of where energy is spent in an application, which proves to be very useful when comparing alternative implementations or validating theoretical energy consumption models.**

***Index Terms*—power profiling, debug, tool, sensor network.**

I. Introduction

Energy efficiency is one of the main concerns in the development of sensor networks. The resource restricted sensor nodes have limited energy supplies, and in order to increase their lifetime, energy efficiency has been considered in many aspects of sensor network research, from platform design [1], to link layers [2], to network layers [3] and applications [4]. This motivates the need for an instrument that offers empirical energy measurements, ranging from detailed evaluations and comparisons of algorithms to verifications of energy consumption models. Such an instrument should not only be used for benchmarking of the final application or protocol implementation; instead, the instrument should be available throughout the development process, instantly showing improvements—or revealing setbacks and errors.

The goal of this work has been to create a tool which will be useful for the sensor network programmer. It must be easily integrated into the development cycle of coding, compiling, loading, executing, and evaluating. Furthermore, it must be low-cost and small, so that it can be used on the desktop—or wherever the programmer wants to use it. The use of big and expensive equipment (such as oscilloscopes) discourages this type of use and leads to experiments in a lab detached from the development cycle. This means that the programmer will not benefit from an instant energy profile of minor code updates, which may reveal intricate information about the running program.

Most energy measurement solutions samples the current through the target system every small time interval in order to produce a graphical representation of the current as a function of time. This leads to enormous amounts of data, but often the question, which the programmer seeks to answer, is something like "how much power did this section of the application consume?" or "under which conditions will protocol A be more energy efficient than protocol B?". To answer such questions, a single scalar value showing the total energy consumption would suffice; perhaps supported by a low-resolution graph for clarity. Hence, all the tiny details picked up with the conventional methods (using, e.g., an oscilloscope) are only occasionally useful.

We claim that often the programmer will benefit more from a tool which offers a concise overview of the energy consumption in chosen parts of the running program as a simple table—where the relevant parts are selected using annotations in the source code. This information can be more useful than a high-resolution (current vs. time) data set, in which the programmer would manually have to identify the relevant time intervals and perform the necessary integrations to obtain the same results.

The *Energy Bucket* is an energy meter designed specifically to be used for sensor network programming. The tool is accompanied by a low memory footprint library which allows the programmer to easily annotate sections of the target program with state numbers. The Energy Bucket will then report the amount of energy spent in each state. To speed up the development process, the use of the Energy Bucket may be incorporated in the build system to launch an energy measurement right after installing a program on a sensor node.

In section II we review related work and in section III we describe and evaluate the precision of the Energy Bucket. In section IV we demonstrate the usage of the tool with three case studies before we conclude and describe future work in section V.

II. Related Work

Most energy measurements found in sensor network research has been performed in the classic lab setup using a digital storage oscilloscope, a specially designed data acquisition board [5], or even a sensor node [6] to measure the voltage over a shunt resistor in series with the target system, followed by integration using some software, typically MATLAB. There

are, however, quite a few alternative energy measurement methods designed for a number of different purposes.

One of these alternative methods, presented in [7], was designed for an accelerated evaluation of battery lifetime for sensor node programs. Instead of powering the node from a battery cell, a very large capacitor is used. The node will discharge the capacitor while executing the target program and die when the capacitor is depleted. The lifetime of the application is measured and used to predict the lifetime when using a real battery.

Another method, explored in [8], is the use of a clamp ammeter (a.k.a. tong-tester). This method has the advantage of being completely unobtrusive. Unfortunately, it is also very susceptible to noise. The setup used in [8] was only capable of measuring 2 decades of current (0.4 mA to 40 mA) before hitting the noise-floor.

In a number of situations, a sensor node will benefit from being capable of measuring its own energy consumption—for instance routing protocols may use this information to route packets around nodes with almost depleted batteries. The *iCount* presented in [9] may be used on sensor nodes that use a typical DC-DC boost converter in its power supply. This is basically just a matter of connecting the control signal of the boost controller IC to a counter input of the microcontroller (MCU). This adds energy measurement to the sensor node for "free"—in the sense that if the MCU has an unused counter input and if the sensor node needs a boost converter anyway, no overhead is added. The accuracy is only within 20 % which makes it unsuitable when precise energy measurements are needed.

More accurate measurements can be obtained using SPOT [10] (or the similar solution in [11]). The SPOT is a sensor board containing a complete and very accurate energy meter that can be read using the sensor node I^2C bus. This solution also enables the sensor node to measure its own energy consumption, but the problem using it in a deployment is that the energy meter consumes a fair amount of power itself.

Rather than measuring the energy consumption on actual hardware, it is possible to completely simulate it. PowerTOSSIM [12] is one way of realising such simulations. Based on detailed measurements of the current draw for each component (MCU, radio, flash, LEDs, sensors) in each mode (sleep, active, etc.), the energy consumption may be calculated for each node in the simulation. This approach has the advantage that it can be performed entirely in software on the developer's workstation, avoiding the need for any measurement equipment.

When it is possible to simulate the energy consumption, why not perform the simulation on the sensor node itself? This is the basic idea of [13], where a sensor OS is modified slightly to monitor the state of each hardware component, keeping a count of the total energy consumption—with only a small computational overhead. This has the advantage over PowerTOSSIM that the simulation is performed in a realistic environment which may capture real-life phenomena that are not captured by the simulated world of PowerTOSSIM. This can also be used as an iCount [9] alternative to keep track of remaining energy in a deployed network.

Although all of the above methods *can* be used during development, it will be quite tedious and laborious when a programmer needs a concise overview of the energy spent in different program parts instead of the total energy consumption.

[14] explores a way to create an energy consumption overview on the process level of Linux-based sensors. The kernel task scheduler is modified to keep energy accounts for each process. The energy spent by each process may be monitored in real-time by the user through 'etop' (a modified version of the 'top' utility) which lists the energy consumption and actual current draw for each running process.

III. The Energy Bucket

As explained in the introduction, we want an energy meter which can produce a simple table of the energy consumption of different sections of a program—based on source code annotations. The SPOT [10] solution mentioned above can be adapted to do this; however, this solution may interfere with the program under test in a couple of ways. First, the I^2C bus, which is used to fetch the readings, is also used for radio and external flash communication on many platforms. Second, all the extra logic required to manage and store the readings (and transmit them to a host) may even end up spending significant time and energy compared to the program being tested. Furthermore, we need a tool that is usable with multiple hardware platforms, rather than being a "sensor board" designed for one particular hardware platform.

The Energy Bucket hardware delivers a constant voltage of 3.0 V to the target system while measuring the delivered charge (note that energy equals charge times voltage). In addition to the power supply output, the Energy Bucket has three high-impedance digital input lines which may be used to read the state of the target program (the particular choice of 3 inputs was arbitrary, and more may be added later).

A small TinyOS 2.x [15] library exporting the following standard C function through a header file, may be included anywhere in the target program:

```
void energy_state(const energy_state_t state);
```

At compile-time, either the real library or an alternative skeleton (with no implementation) may be chosen. The real library outputs the chosen state (0–7) to three general I/O pins on the MCU which are connected to the three inputs lines on the Energy Bucket. Most sensor prototype platforms have plenty of general I/O lines available, so this requirement should be easily met.

A. Measurement Setup

Figure 1 shows a typical measurement setup. The target system is connected to the development computer for easy re-programming through a modified USB cable, and to the Energy Bucket in order to collect online measurement data. The USB cable is disabled by the Energy Bucket when running

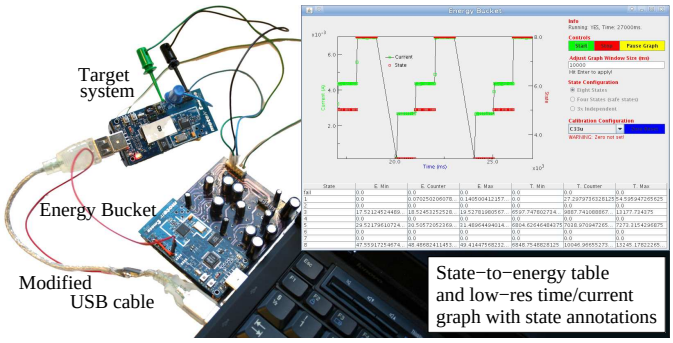

Figure 1. Using the Energy Bucket

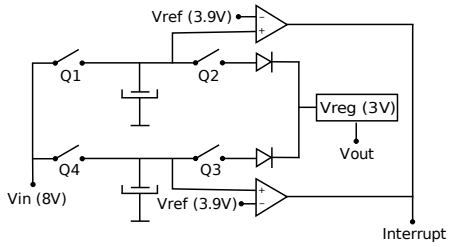

Figure 2. Overview schematic

so that the target system does not draw any current from the USB connection. This setup enables easy integration of the Energy Bucket in the previous mentioned development cycle. Figure 1 also shows how the developer can follow the online current draw and a charge to state table on the development computer.

B. Energy Measurement Method

Sensor node power profiles are characterised by long periods of ultra-low current combined with short bursts of high(er) current. As both types may contribute significantly to the overall energy consumption, neither can be ignored in the measurements [10].

The common method of sampling the voltage over a shunt resistor using some A-D conversion (e.g., a digital storage oscilloscope) requires a subsequent integration of the data. Hence, this method yields precise results only when the ADC's voltage uncertainty is negligible compared to the minimal voltage over the shunt resistor, and the ADC's time period is negligible compared to the width of the bursts. As sensor node current consumption ranges over 5 decades (1 μA – 100 mA), this implies that a (linear) ADC must have a voltage resolution of 18 bits or more, which makes most affordable oscilloscopes (with the typical 8 bits resolution) unsuitable for this task.

The approach taken by the Energy Bucket is to count the number of charge/discharge cycles of a buffer capacitor. The Energy Bucket precisely controls the voltages which the capacitor is charged to and allowed to discharge to. Thus, each cycle will always transfer the same amount of charge—one "bucketful". Furthermore, by keeping the output voltage at a fixed level, each bucketful of charge equals a bucketful of energy, hence the name "Energy Bucket".

Figure 2 shows a block schematic of the Energy Bucket hardware. The four switch symbols represent transistors controlled by the Energy Bucket software running on a Tmote Sky [16]. Two identical electrolytic capacitors are used as buffer caps, so that one can be charged to 8.0 V, while current to the target circuitry is drawn from the other cap. When the voltage over the discharging cap falls to 3.9 V, the two caps are switched. A comparator for each buffer cap detects when it is time to trigger a switching and signals this to the Tmote. A voltage regulator adjusts the output voltage down to 3.0 V. Apart from the Tmote Sky, all parts used are common low-cost off-the-shelf components with a total price of around €50.

Currently, all switching events are time-stamped and sent to the host computer, on which a Java program will perform all calculations. This, however, causes a bottleneck at the serial communication between the Tmote Sky and the host computer, as the maximum UART baud rate is 115,200. Each packet is currently 10 bytes long, so this limits the Energy Bucket to 1152 switchings per second. Since the measured frequency is proportional to the current draw of the target system, which ranges over about 5 decades, this implies that the worstcase time between switchings may be as long as 2 minutes. This is not a problem, if all we want to do, is to measure the overall energy consumption. Still, it becomes difficult to measure the time and energy spent in each individual state, when significant time elapses between the readings. The current solution to this problem is simple: when a state change happened during the elapsed interval, half of the total time and energy of this interval is assigned to the new state and the previous state respectively. However, the Energy Bucket also keeps track of minimum and maximum energy and time measurements for each state and these counters will be increased by either zero or the full time and energy of the interval. We will know for sure, that the real energy consumption and time will be within these "confidence intervals", so if these intervals are reasonable narrow, the result is acceptable. In the Future Work section, we propose yet another approach to fix this problem.

The Energy Bucket circuitry was dimensioned to deliver currents up to 150 mA. In order to reach this current, the minimal capacitance is given by:

$$C = \frac{150 \text{ mA}}{1152 \text{ Hz} \cdot 4.1 \text{ V}} = 31.8 \ \mu\text{F}$$

and we chose to use 33 μF capacitors. Alternatively, if we performed all calculations locally on the Tmote Sky, the UART limitation would vanish, and the maximum frequency would be many times higher, resulting in a much improved temporal resolution.

C. Evaluation and Calibration

In order to calibrate and evaluate the accuracy of the Energy Bucket, we adjusted the output to 3.0 V (using a regular low-cost multimeter) and performed 49 measurements with different fixed combinations of resistors (0.1 % tolerance) in the range 20 Ω–30 MΩ, corresponding to 0.1 μA–150 mA.

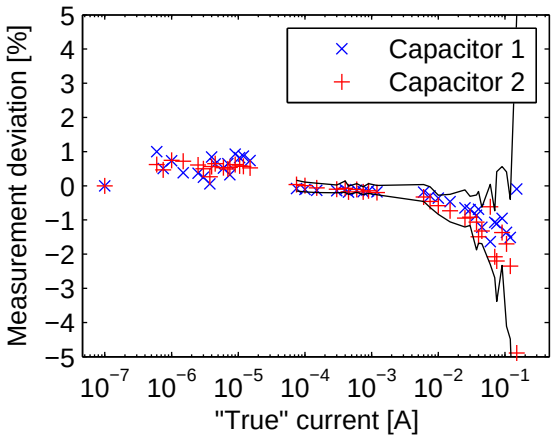

Figure 3. Evaluation results

We used 30 out of these 49 measurements (those in the 1 μA–1 mA interval) for calibration, while all measurements were used to evaluate the resulting calibration.

We chose to perform two separate calibrations, one for each capacitor, as small differences between the capacitors and other components in each of the two circuit paths can be expected. Hence, from each measurement dataset from resistors under 100 kΩ, we picked 100 switching timestamps—50 for each capacitor; and from the remaining (much smaller) datasets, we picked only two timestams, i.e., 1 per capacitor, as the duration for each discharge grows to almost half an hour for the 30 MΩ resistor.

As the total charge is expected to be proportional to the number of switchings (a fixed amount of charge, Q, is transfered each time), we expected the current (which is always a constant in these measurements) between switchings to be proportional to $1/t$, where t is the time between the two switchings, or at least a linear function, $I(t) = Q \cdot (1/t) - I_0$, where I_0 is a constant current leak caused by component imperfections. Analysing the data, we found that this was not exactly the case. A double logarithmic plot of the data showed that in fact the current between switchings is a function $I(t) = Q \cdot (1/t)^e - I_0$ where the exponent, e, is a constant slightly smaller than 1. Experiments indicated that capacitors of same type and value have almost identical e values, while different capacitors (either different type *or* different value) can have significantly different $(1 - e)$ values.

Using the 30 measurements with currents in the interval 1 μA–1 mA, we get the values for Q and e, and using the 30 Ω measurement, we get the value for $I_0 = -17$ nA, i.e., a small current is leaking *into* the circuit:

$$I_1(t) = 144.7\ \mu C \cdot (\tfrac{1}{t})^{0.9897} + 17.3\ nA$$
$$I_2(t) = 146.7\ \mu C \cdot (\tfrac{1}{t})^{0.9892} + 17.8\ nA$$

A plot of the percentual deviation of the measured current from the "true" current is shown in figure 3. As mentioned earlier, each measurement point for resistors below 100 kΩ represent an average of 50 switching timestamps. The black lines show the minimal and maximal value among each of the 100 values of both capacitors. The "true" current values are really calculated as 3.0 V/R, where R is the resistor value used, and it is an implicit assumption that the output voltage is constant. This assumption is not accurate, however. In fact, under high load (>100 mA) the voltage fluctuates a bit and may even drop as low as 2.8 V—which is a 6 % drop in voltage giving rise to up to a 6 % error in the "true" current as well. This explains the reduced accuracy and general drop in the curves of figure 3, and from the measurements shown on this graph, we can conclude, that the Energy Bucket measures *current* and *charge consumption* within ± 2 % or better, over more than five decades of current consumption (1 μA–100 mA). The instrument also measures *power* and *energy consumption*, but due to saturation of the output voltage regulator, the accuracy drops a bit when the current draw exceeds 50 mA.

Component tolerances—as high as 20 % for the buffer capacitors—suggest that environment (primarily ambient temperature) and ageing effects should be considered. All our experiments were carried out in an office under normal room temperature conditions. Each time the Energy Bucket is used, we also make a few test measurements using the reference resistors in order to check the calibrations, and so far no discrepancies have been observed. As the device is only 6 months old, ageing effects may still appear in the future. However, as the primary purpose of the tool is *not* to deliver accurate absolute measurements, but rather to do comparative studies (such as comparing the energy consumption of different implementations of a component), using the latest calibration will be sufficient most of the time. Redoing the calibration would be recommended prior to any measurement where absolute accuracy is essential.

IV. Case Studies

In this section we demonstrate the use of the Energy Bucket as a tool for debugging, comparing alternative implementations, and validating energy consumption models.

A. Debugging: Telos A vs. Telos B

In order to back up our claim that measuring the energy consumption should be a fundamental part of writing and debugging programs, we present a concrete case where the Energy Bucket—in fact, the first time it was ever used—revealed a programming flaw in an application, which had gone unnoticed for more than 6 months.

One of the authors developed a communication library containing some extremely timing-critical parts, and in order to debug some library procedures, two digital signals were output to a multi-channel oscilloscope. The use of these outputs would be enabled by a precompiler constant, defined in the Makefile.

The library was developed and tested on Tmote Sky motes, which is a telos revision B design [16]. The library, as well as an application using the library, was tested on this platform as well, including a test of its energy consumption—which was within the design limits. The application was, however,

deployed on a mixture of Tmote Sky and BSN [17] motes—the latter being a telos revision A design.

After installing the application on a BSN mote, the Energy Bucket revealed that this node was drawing 20 mA more current than expected. We proceeded to use the Energy Bucket in an iterative "alter code, compile, load, execute and measure" process in order to narrow down the reason for this current drain. The error turned out to be in the Makefile, as the above-mentioned precompiler constant definition was never deleted. One of the debug ports used was port 6.7 on the MSP430 MCU, which is exported for external use in the Telos revision B design. On the Telos revision A design, however, this port is connected directly to the positive supply rail, so a low output on this port caused a short-circuit.

Since the application worked perfectly well on both platforms, and we had no plans of performing extra lab tests on the BSN mote, since the application had already been thoroughly tested on Tmote Sky motes, this bug would probably have gone unnoticed, if we did not have this tool. Furthermore, this experience proved, that having instant energy measurements available while debugging (and programming in general) is indeed a very powerful tool, as the power fingerprint of an application may reveal a lot more about what is going on, than three LEDs.

B. Comparison: CC2420 vs. MSP430 CCM security

Sensor network researchers and developers often evaluate novel problem solutions by comparing them to previously proposed ones [4]. The dominant performance metrics in such comparisons include time and energy: time measurements are important when estimating the throughput of a routing or data processing algorithm and energy measurements are important with regard to lifetimes. In this section we show how to use the Energy Bucket to compare the energy and time consumption of the CC2420 radio inline Counter and Cipher Block Chaining Message Authentication Code (CCM) mode security mechanism [18] to a similar software implementation[1] when used with packet transmission. It is known that the CC2420 inline CCM security mechanism will outperform a similar software implementation with relation to a time metric [19], but the fact that the radio could be turned off when doing the software security operation could favour it with relation to an energy metric.

The CC2420 radio inline CCM security mechanism works on packets already present in the CC2420 transmit and receive buffers. Doing packet transmission the read/write to these buffers needs to be done anyway, so the security overhead only consist of initialising the security (setting keys and nonce) and performing the actual encryption/decryption.

We implemented the CC2420 inline CCM security operation and ported the similar software implementation to TinyOS 2.x [15] running on an MSP430 MCU based Tmote Sky [16] connected to the Energy Bucket that recorded energy consumption for the interesting program states. Table I shows a comparison of the measured energy and time consumption with confidence intervals for one encryption using the two different implementations. Each value is an average of 12000 encryption operations. The CC2420 inline CCM security implementation makes use of the default TinyOS 2.x CC2420 radio stack which puts the radio in receive mode when on. This favors the software implementation as the CC2420 security operations only require the radios oscillator to run.

[1] http://gladman.plushost.co.uk/oldsite/AES/

Table I
ENERGY BUCKET CCM EVALUATION RESULTS

	Energy [mJ]			Time [ms]		
	min		max	min		max
CC2420	10.05	11.13	12.21	0.17	0.21	0.24
MSP430	50.13	50.37	50.58	8.56	8.62	8.68

Table II
STATES

State	Description
Starting	Start the radio's voltage regulator, start the radio's oscillator, wait for oscillator to stabilize, enable receive state.
Transmitting	Set packet headers and transmission power, write packet to the radio's transmit buffer, wait an initial back-off time, do clear channel assessment, transmit the packet, wait for acknowledgment (optional).
Stopping	Stop the radio's voltage regulator.

We see that according to the time measures the CC2420 radio inline CCM security mechanism is $8.62/0.21 = 42$ times faster and uses $50.37/11.13 = 4.5$ times less energy than the favored similar software implementation running on the MSP430 MCU. To verify the time measures we did another experiment in which we measured the duration of one encryption operation using the MCU's microsecond timer. Note that these time measures also verifies the related energy measures as time and energy is allocated to the states in the same way. Averaged over 250 encryptions, the CC2420 inline CCM security mechanism and the similar software implementation took 0.189 ms. and 8.68 ms, respectively. These values are within the confidence intervals of the times measured by the Energy Bucket.

C. Model validation: Packet transmission

Evaluating sensor network programs with regard to energy efficiency is often done from an energy consumption model of the program [4], [12], [13]. The model divides the program into a set of fixed states, S_i, with an associated current, I_i, that can be derived from experimental evaluation or datasheet lookups. The total charge consumption of the program is then calculated from the time, T_i, spent in each state:

$$Q = \sum_i T_i \cdot I_i$$

In this section we show how we used the Energy Bucket to validate an energy consumption model.

Inspired by a problem from the SensoByg project [20] we evaluate the energy consumption of a single sensor node that periodically broadcasts its acquired data to a potential receiver.

Table III
MODEL VALIDATION RESULTS

Model	NoAck	AckReliable	AckUnreliable
Starting time	2.64 ms	2.64 ms	2.64 ms
Transmitting time	10.20 ms	12.45 ms	18.02 ms
Stopping time	0.21 ms	0.21 ms	0.21 ms
Calculated charge*	0.23 mC	0.27 mC	0.37 mC
Energy Bucket	NoAck	AckReliable	AckUnreliable
Measured charge*	0.20 mC	0.24 mC	0.34 mC
Deviation	NoAck	AckReliable	AckUnreliable
Absolute	17.36 %	12.82 %	8.53 %
Corrected	-2.74 %	-3.75 %	-3.22 %

* per sent packet

For the sake of simplicity, we leave out the sensor readings and instead transmit a static 28 byte data packet. The transmission of the data packet consists of starting the radio, transmitting the packet, and stopping the radio again. The application was implemented in TinyOS 2.x [15] running on a Tmote Sky [16]. The states of the program are described in detail in Table II.

The transmission of a packet can be done with or without an acknowledgment. When enabling acknowledgments, the time spent in the *transmitting* state depends on the time the sensor node has to wait for the acknowledgment. We derive three variations of the model: one without the use of acknowledgments (NoAcks), one where a receiver acknowledge the packet immediately using software acknowledgments (AcksReliable), and one where the expected acknowledgment from the receiver is lost (AcksUnreliable).

We measured the time spent in each state for the three variations of the model using the MCU's microsecond timer and calculated the charge consumption per transmitted packet based on the currents listed in the CC2420 radio datasheet [18]. The TinyOS 2.x CC2420 radio stack implementation puts the radio into receive mode when started, so we let the current in the *starting* and *stopping* states be the current of the radio in receive mode (19.7mA) and the current in the *transmitting* state be the current of the radio in transmit mode (17.4mA). Note that this is an overestimate of the current in the *starting* and *stopping* states as the radio will not be fully on the entire time. On the other hand, this could be neutralised by the smaller current drawn by the MCU, not included in the model, and the underestimate of the *transmitting* state as it is inevitable that the radio will not spend some time in receive mode here. The first part of Table III shows the time measure for each state and the calculated charge consumption per packet (time multiplied by the theoretical current) for the three variations of the model. The time values are averaged over 1000 sent packets with a variance of zero for the starting and stopping times and 7.58, 11.84, and 7.51 for the NoAck, AckReliable, and AckUnreliable transmitting times, respectively. These variances are due to the randomized initial backoff time and delay in the acknowledgement (for AckReliable). The experimental values from the Energy Bucket shown in the second part of Table III is the average charge consumption per packet calculated from the consumption of a sensor node transmitting 1000 packets.

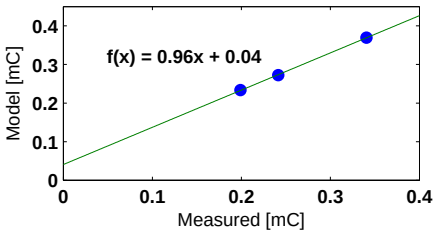

Figure 4. Relation between model and measured charge

Our experiments show that the calculated charge consumption values from the model deviate from the measured values. In Figure 4 we plot the relation between the model and measured values and see that the regression curve shows a constant deviation of 0.04 mC. This has to be due to the overestimate of the charge consumption in the *starting* and *stopping* states and explains why the absolute deviation shown in Table III is decreasing with the increasing transmission time. If we correct the model with this constant factor the corrected deviation between the model and the measurement stays within an acceptable 4% (c.f. Table III) and we conclude that this corrected model is valid.

V. CONCLUSION AND FUTURE WORK

We designed and evaluated the Energy Bucket, a tool for relating energy consumption to parts of a target program by measuring the energy usage. In case studies, we demonstrated that the Energy Bucket is a valuable tool for programmers when writing and debugging programs, comparing alternative implementations, and validating energy consumption models. The case studies emphasized the benefits of having a tool that offers instant power profiling and can be integrated into the development cycle. The tool facilitates early programming error discovery and conveys a better understanding of the target system's behaviour.

The Energy Bucket achieved an accuracy of less than 2 % over five decades of current draw. However, the accuracy of the energy to state measured by the Energy Bucket depends on the time resolution of the measurements. Using smaller capacitors will increase the resolution, but the current system has performance limitations with regard to how fast it can alternate between the capacitors.

Future work include optimizing the Energy Bucket for performance by reducing the packet size for the serial connection or by eliminating online serial communication by calculating the charges used in each state on the Tmote Sky instead of the host computer. In situations where very high time resolution is required the capacitor switching could be implemented in hardware to decrease the source of error caused by the switching delay. Using hardware-based switching and feeding the switching signal to a timer input on the MCU, the maximum switching frequency could be as high as 10 MHz on the current MCU [21]. Choosing capacitors such that, say, 5 MHz would correspond to 100 mA, a current draw of 2 μA would correspond to 100 Hz. An orthogonal approach to improve

the accuracy of the energy to state measurements is to force a capacitor switch at each state change. This requires a way to measure the unused charge remaining in the capacitors when they are switched out—e.g., using one of the ADCs available on the MCU. This solution would be highly accurate, similar to the approach used in [22].

ACKNOWLEDGEMENT

The research presented in this paper was partially funded by the Activity-Based Computing project (http://activity-based-computing.org/) and the SensoByg project (http://www.sensobyg.dk/english/) at Aarhus University.

REFERENCES

[1] J. Polastre, R. Szewczyk, and D. Culler, "Telos: enabling ultra-low power wireless research," in *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*. Piscataway, NJ, USA: IEEE Press, 2005, p. 48.

[2] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks," in *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. New York, NY, USA: ACM, 2004, pp. 95–107.

[3] K. Lin and P. Levis, "Data discovery and dissemination with dip," in *IPSN '08: Proceedings of the 7th international conference on Information processing in sensor networks*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 433–444.

[4] R. Musaloiu-E, C.-J. Liang, and A. Terzis, "Koala: Ultra-low power data retrieval in wireless sensor networks," *Information Processing in Sensor Networks, 2008. IPSN '08. International Conference on*, pp. 421–432, April 2008.

[5] I. Haratcherev, G. Halkes, T. Parker, O. Visser, and K. Langendoen, "PowerBench: A Scalable Testbed Infrastructure for Benchmarking Power Consumption," in *Int. Workshop on Sensor Network Engineering (IWSNE)*, 2008.

[6] L. Selavo, G. Zhou, and J. Stankovic, "Seemote: In-situ visualization and logging device for wireless sensor networks," Oct. 2006, pp. 1–9.

[7] H. Ritter, J. Schiller, T. Voigt, A. Dunkels, and J. Alonso, "Experimental evaluation of lifetime bounds for wireless sensor networks," *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pp. 25–32, Jan.-2 Feb. 2005.

[8] A. Milenkovic, M. Milenkovic, E. Jovanov, D. Hite, and D. Raskovic, "An environment for runtime power monitoring of wireless sensor network platforms," *System Theory, 2005. SSST '05. Proceedings of the Thirty-Seventh Southeastern Symposium on*, pp. 406–410, March 2005.

[9] P. Dutta, M. Feldmeier, J. Paradiso, and D. Culler, "Energy metering for free: Augmenting switching regulators for real-time monitoring," *International Conference on Information Processing in Sensor Networks*, vol. 0, pp. 283–294, 2008.

[10] X. Jiang, P. Dutta, D. Culler, and I. Stoica, "Micro power meter for energy monitoring of wireless sensor networks at scale," *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, pp. 186–195, April 2007.

[11] T. Trathnigg and R. Weiss, "A runtime energy monitoring system for wireless sensor networks," *Wireless Pervasive Computing, 2008. ISWPC 2008. 3rd International Symposium on*, pp. 21–25, May 2008.

[12] V. Shnayder, M. Hempstead, B.-R. Chen, and M. Welsh, "PowerTOSSIM: Efficient power simulation for tinyos applications," in *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2004. [Online]. Available: http://www.eecs.harvard.edu/~shnayder/ptossim/

[13] A. Dunkels, F. Osterlind, N. Tsiftes, and Z. He, "Software-based on-line energy estimation for sensor nodes," in *EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors*. New York, NY, USA: ACM, 2007, pp. 28–32.

[14] T. Stathopoulos, D. Mclntire, and W. Kaiser, "The energy endoscope: Real-time detailed energy accounting for wireless sensor nodes," April 2008, pp. 383–394.

[15] P. Levis, D. Gay, V. Handziski, J.-H. Hauer, B. Greenstein, M. Turon, J. Hui, K. Klues, C. Sharp, R. Szewczyk, J. Polastre, P. Buonadonna, L. Nachman, G. Tolle, D. Culler, and A. Wolisz, "T2: A second generation os for embedded sensor networks." Tech. Rep., 2005.

[16] "Tmote sky datasheet." [Online]. Available: http://www.sentilla.com/pdf/eol/tmote-sky-datasheet.pdf

[17] "BSN node specifications." [Online]. Available: http://bsn-web.org/index.php?article=926

[18] "Cc2420 2.4 ghz ieee 802.15.4/zigbee-ready rf transceiver." [Online]. Available: http://focus.ti.com/lit/ds/symlink/cc2420.pdf

[19] M. T. Hansen, "Asynchronous group key distribution on top of the cc2420 security mechanisms for sensor networks," in *WiSec '09: Proceedings of the second ACM conference on Wireless network security*. New York, NY, USA: ACM, 2009, pp. 13–20.

[20] "The sensobyg project." [Online]. Available: http://www.sensobyg.dk/english

[21] "Msp430x15x, msp430x16x, msp430x161x mixed signal microcontroller (slas368e)." [Online]. Available: http://focus.ti.com/lit/ds/symlink/msp430f1611.pdf

[22] N. Chang, K. Kim, and H. G. Lee, "Cycle-accurate energy consumption measurement and analysis: case study of arm7tdmi," in *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*. New York, NY, USA: ACM, 2000, pp. 185–190.