Energy-Aware Device Drivers for Embedded Operating Systems

Markus Buschhoff
TU-Dortmund University
Computer Science XII
Dortmund
markus.buschhoff@tu-dortmund.de

Robert Falkenberg
TU-Dortmund University
Communication Networks Institute
Dortmund
robert.falkenberg@tu-dortmund.de

Olaf Spinczyk
TU-Dortmund University
Computer Science XII
Dortmund
olaf.spinczyk@tu-dortmund.de

ABSTRACT

Energy harvesting solutions with rechargeable batteries are a frequent choice to tackle the problems of supplying continuous power to deeply embedded devices like wireless sensor nodes. However, if the utilization of a node is not thoroughly planned, the battery may be drained too early and a continuous operation of such a device may become impossible. Here, an energy-management solution is required to control the flow of energy. As a foundation for energy management in software, we introduce a concept that allows to model energy consumption of hardware and to synthesize energy aware device drivers from these models. Our drivers are able to account the energy consumption of each driver function call at an accuracy of more than 90%. We provide a detailed overhead and accuracy evaluation of a driver implementation and hence prove the feasibility of our concept.

CCS CONCEPTS

• **Computer systems organization** → **Embedded software**; • **Software and its engineering** → **Power management**; • **Hardware** → *Energy metering*;

KEYWORDS

embedded systems, energy, synthesis, modeling, awareness, operating system, driver, profiling, periphery

ACM Reference Format:
Markus Buschhoff, Robert Falkenberg, and Olaf Spinczyk. 2018. Energy-Aware Device Drivers for Embedded Operating Systems. In *Proceedings of EWiLi'17, Seoul, South Korea.* ACM, New York, NY, USA, 6 pages.

1 INTRODUCTION

The reduction of energy consumption is one of the most important challenges to face in the fields of wireless sensor networks, ubiquitous computing and other deeply embedded system scenarios. This is due to the fact that the lifetime of these systems is often coupled to the lifetime of their batteries. To solve this problem, there is a high demand for battery-free systems that harvest energy from their environment. Yet, this is not easy to achieve, since energy-harvesting has severe constraints: Usually, there is only little energy available in the environment of a system, and prevailing harvesting systems, like solar cells, still have low efficiency. Thus, to enable harvesting for embedded devices, the whole system has to be optimized to limit the maximum energy consumption and to plan energy consumption over time. To facilitate this, a system has to be *aware* of the energy income and consumption.

EWiLi'17, Seoul, South Korea, 2018
© 2018 Copyright held by the owner/author(s).

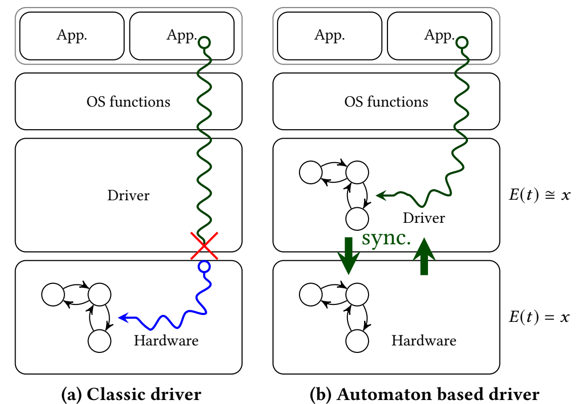

Figure 1: The classic implementation (left) showing the break in the information flow between the hardware and the software layer, while our driver (right) keeps the device's automata model in sync with the software layer and thus can deduce the device's energy consumption.

To achieve this, our approach uses automaton-based energy models for the hardware and creates driver scaffolds to be filled with the actual code required to drive the hardware. Our drivers create a bridge over the semantic gap between hardware state machine models and driver state as shown in Figure 1.

The approaches shown in this paper were developed for smallest-scale embedded systems like sensor network nodes running on 8 or 16 bit micro controllers. They are not intended to scale for larger systems with complex periphery (like GPUs), multiple CPU cores and user-controllable multitasking operating systems.

The following shows a list of our contributions in this paper:

- a methodology to map hardware state machine models to driver functionalities
- the integration of a CPU energy model
- the integration of energy accounting into drivers
- an evaluation of code, memory and energy overheads

The rest of this paper is structured as follows:

To understand the concepts of energy-awareness in operating systems, we start with a comprehensive look at related research. Then, we briefly explain the basics of creating hardware energy models for peripheral devices and the CPU.

After describing our concept of hardware energy models, we address the implementation of hardware drivers from these models that enable energy accounting.

In the last chapter we evaluate the accuracy and overhead of our driver implementations on a typical sensor node platform. We show that our driver concept is feasible for small embedded systems and still shows an accuracy of more than 90%. Additionally, we analyze the energetic overhead imposed by the energy accounting.

2 RELATED WORK

There are several works about driver synthesis that show similarities to our approach, like [6] and [15], which all have in common that they are based on automata models or similar transition systems. This leads to the assumption that device drivers can be described well using such formalisms. In contrast to our contribution, the synthesis approaches in the literature focus on the formal description of the device's functionality to optimize resource consumptions or to prove correctness. In our approach, we focus on the non-functional properties of a device.

Mérillon et al. developed Devil [10], a language to describe the interface of peripheral components. Devil operates on the level of registers and ports, and was implemented to support creating device drivers with a focus on *how* a device is *accessed*. We see our approach as complementary, because our model describes *what* a device *does*. The combination of both approaches could be beneficial and is subject of future work.

In the field of resource modeling and simulation, Steinke et al. describe a highly detailed model for the simulation of the energy consumption of processors that even considers changes in electrical charge on all bus connections [12]. Other publications focus on the logical view of a component: Wang and Yang use an automata-based approach to form an energy model for a sensor node to drive simulations [14]. Similar to our approach, the functional level of the device is represented as an automaton, and the energy consumption is annotated at the states and transitions. Weder uses a similar modeling approach to drive an OMNeT++ simulation with the original firmware for a sensor node [16]. The abstraction from actual hardware is driven further by Tan et al. [13], with the goal to estimate the energy consumption of distinct system services. The authors create an empirical model by measuring the energy consumption of distinct services in a fixed system configuration. A more generic approach is shown by Seceleanu et al. [11]: They do not restrict their model on energy, but model other resources like memory, IO ports and buses, as well. Here, as in our approach, priced timed automata (PTA) models are used for the simulation and analysis of resource consumptions. All these approaches show models that are fundamental to our work, but cannot be calculated efficiently at run-time on a sensor node due to their high level of detail.

Combined approaches that implement models into the runtime system are typically achieved by reducing complex offline models for online usage. Kellner et al. propose an optimization of nodes running TinyDB by load balancing [9]. This is done by simulating a model synchronously to the actual functionality of the system, thus reducing a complex model for a distinct application. Our approach aims at reusable models incorporated into an operating system.

Buschhoff et al. reduce energy models by offline simulation of context scenarios [4]. The simulations are automata driven and implemented in OMNeT++. The results of the simulations are the averaged energy consumptions for usage scenarios. An application can become energy aware by deducing the current scenario from context parameters and choosing the respective value from the energy table. This assumes that usage scenarios can be identified at design time and can be recognized at run-time, which is not always the case.

TinyOS[1] is a popular operating system for sensor networks. TinyOS uses drivers that support power management by keeping track of the on-/off state of peripheral devices. Kellner and Bellosa describe an approach to add energy accounting to TinyOS [8], which is very focused on the special infrastructure of this operating system. They did not evaluate any costs, constraints, or the achieved accuracy.

Zeng et al. show an energy management concept for the ECOSystem operating system [18] based on a virtual energy unit called *Currentcy*. The amount of available Currentcy is calculated from a battery model and given to the application at the beginning of an "energy epoch". We consider this approach complementary to ours, since Zeng et al. focus on energy management and do not have an integrated energy modeling and awareness concept. The idea of our drivers is to fill this gap.

On larger platforms, e.g. Android, plenty of approaches address energy issues. Here, the automatic generation of device and application energy models is a central topic. To address the huge number of different devices and applications in the field, machine learning algorithms are often used to generate models. Since there are a plenty of publications in this area, we refer to Appscope [17], and PowerBooter [19] as popular representatives. Datta et al. show a more complete view on current features and trends in Android power management [7]. Our approach addresses similar, yet different problems: In contrast to Android, the devices and applications used here are usually well defined at the compile time of the system, but the target platform usually has insufficient resources to utilize machine learning algorithms.

3 HARDWARE ENERGY MODEL

In the following we describe the used modeling approach and the synthesis of a driver scaffold from a given model.

The foundation of our driver concept is a set of energy models for peripheral devices and the CPU. We use an extension to *priced timed automata* (PTA) to describe our hardware as state machines. PTAs extend the typical *finite state machine* definitions by a cost model (prices) and timers that enable time-based transitions between the states of the automaton [1, 2].

Our PTAs are basically functional models of a peripheral device. That means, they show all necessary states and transitions for the supported features of the device. This set of states may also be extended by additional non-functional states, which model internal energy consumptions of the device that are not fully visible in pure functional models.

Figure 2 shows a simplified PTA model of a Texas Instruments CC1200 radio transceiver. The costs, in terms of energy and power,

[1] http://www.tinyos.net

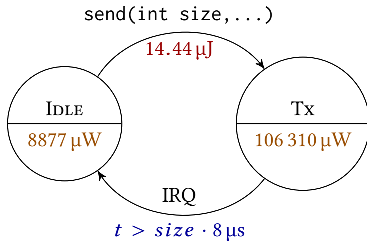

Figure 2: Simplified PTA model of the CC1200 tranceiver.

are annotated at the transitions and the states of the model. While the transitions consume a fixed amount of energy, and thus are annotated by energy values in Joule or Watt-seconds, a state consumes energy over time. So automata states are annotated by their *power* consumption in Watts, in contrast to the *energy* consumptions of transitions.

The shown model consists of the states IDLE, which is the initial state after power-on, and a send state (Tx) that is active during transfers. These states have an energy consumption which is dependent on the time spent in each respective state.

There also may be transitions issued by the hardware at the end of an operation. This may happen after a given amount of time, or is signaled either by an interrupt request (IRQ) to the CPU or by a testable flag (*busy waiting*). PTAs already have a notation for time-triggered transitions. But, also transitions that signal IRQs or busy flags have to be annotated accordingly.

This simple demonstration model consisting of two states and two transitions shows the basic behavior of a typical embedded device driver. In a future extension, this modeling concept may include a cost model for parameters that get passed to a transition, enabling a more accurate modeling. However, this is out of the scope of this paper.

3.1 Driver scaffold

For a given model, a driver scaffold can be generated by transforming every transition (and its parameters) to a function call. By manually filling the functions with the respective code to drive the hardware into the target state, all functional aspects of the hardware can be used.

Transitions that are annotated with the IRQ tag have to be created as an interrupt service routine (ISR). For busy-waiting, a busy-loop has to be created in the program code to request busy information from the device. This has to be done manually, as the energy model cannot handle the various ways of obtaining busy information from actual hardware. The busy loop forms the end of the inbound transition to an operation's state. When the device is not busy anymore, the outbound transition function must be called, which is necessary to perform tasks like energy accounting. In a similar fashion, timed behavior can be implemented: Instead of the busy loop, a delay or timer mechanism has to be issued. After the delay, the outbound transition must be called to signal the end of an operation and to do the accounting.

The generated code scaffold contains code to do the housekeeping and accounting: By using a time-stamp, the time spent in the recent PTA state is determined and the time-stamp for the next state gets prepared. The amount of energy spent is determined by the product of the recent state's duration and power consumption, plus the energy consumption of the transition.

3.2 Modeling CPU Energy Consumption

Accurate energy models for a CPU are hardly available due to the complexity of these devices. However, for small scale (8 and 16 bit low power, single-core microcontrollers), we will show that reasonable results can be achieved by modeling the active and sleep modes as a state machine with averaged power consumption.

This method has some advantages: It is simple, can be calculated online and is compatible to the modeling scheme shown above. In fact, creating a driver scaffold for a *CPU driver* allows us to structure code for the transitions between sleep modes.

In our scenario, the CPU is put to a deep sleep mode when the scheduler of the used operating system is idle. Any interrupt wakes the CPU, hence we added a call to the wake-up transition of the CPU driver at the start of every ISR. This enabled us to establish our energy-accounting mechanisms for the CPU by using the same concept that we used for peripheral drivers.

4 EVALUATION

In this chapter we show an evaluation for utilizing our driver concept within the KratOS[2] operating system [3] for different peripheral devices. In the first part we present the costs of the implementation in terms of code size, memory consumption, CPU time and energy. Here, we point out that our approach is feasible for even smallest-size systems. Next, we evaluate the accuracy of our approach by comparing the values of the implemented energy accounting with values measured during the same runs. We can show that by the use of accurate, yet simple models for real-world periphery, our approach exhibits a constant relative accuracy of over 90

4.1 Memory Costs of the Energy Model

Starting point and ground-truth of this evaluation is an existing driver implementation, that offers functions to change the device state according to the device's functional automaton. The first enhancement to this driver is to add static model data as a foundation for energy calculations.

The amount of additional text segment memory that each PTA driver needs for storing its constant power and energy consumption values in our implementation can be calculated by

$$M_{driver}^{text} = (S + T) \cdot 4\,B \tag{1}$$

Here, S represents the number states, and T represents the number of transitions of the corresponding automaton. We decided to store energy and power values as a 32 bit unsigned integer constant, representing the value in units of nJ or nW, respectively.

4.2 Costs of the Energy Accounting

The implementation of energy accounting on top of our drivers is more complex and has more overhead, because it implements a completely new functionality. We analyzed the static memory

[2]KratOS, acronym for KratOS is a Resource Aware and Tailored Operating System

overhead in the text segment and the overhead on the BSS segment, where global variables are stored.

Basically, we update the driver's energy count at the beginning of a transition. Here, the following operations are required:

- **read** current time (ticks)
- **substract** stored timestamp from current time
- **store** current time as new timestamp
- **multiply** recent state's cost factor with time difference
- **add** result to energy consumed
- **add** transition's energy cost to energy consumed.

We can see, that one time-stamp and one energy variable (64 bit each) is required per driver. Additionally, an integer (32 bit) is needed to save the current state.

To gain a more practical impression of the costs, we analyzed the memory segments of compiled C++ drivers with and without accounting on an MSP430 architecture. Here, we used the CPU driver and a display driver (Sharp-96 LCD) to evaluate the overheads as shown Table 1 for the text and BSS segments.

Configuration	text	BSS	sum
No accounting	0 B	0 B	0 B
Display accounting	624 B	18 B	642 B
CPU accounting	588 B	18 B	606 B
Both	1160 B	36 B	1196 B

Table 1: Memory segment growth for different energy accounting configurations.

From these values, we estimated the text segment overhead of the accounting implementation as follows:

$$M_{acct}^{text} = n \cdot 470\,B + 22\,B \cdot \sum_{i=1}^{n} E_i \tag{2}$$

Whereas n is the number of drivers and E_i is the number of transitions of a distinct driver. Of course, these numbers can only give an impression, since the actual costs for each operation varies for different architectures, optimization levels and implementation details.

The overhead of the accounting is mainly a result of arithmetic operations issued on large data words (32 and 64 bit words) that are necessary to represent time and energy values with a reasonable accuracy.

4.3 Accuracy

As a measure for the achieved accuracy, we have chosen to model an MSP430FR5969 CPU on the respective TI-Launchpad, as well as a Sharp-96 “Booster-Pack” display and a TI CC1200EMK-868-930 transceiver as periphery.

To gain energy values for our models, the components were driven into the distinct states while transitions were signaled to the measurement equipment (MIMOSA [5]) by using a GPIO[3] pin. By that, we were able to map the measured energy values to the states and transitions of the model.

[3] General Purpose Input/Output

The display driver can be considered as fully functional: All necessary functions like initialization, clearing the screen and writing pixel data to distinct display positions were implemented. The CPU driver was implemented as explained in Section 3.2, and has an active (high-power) CPU state and a low-power mode, which is activated when the scheduler becomes idle. The system clock was set to 16 Mhz. The CC1200 module was modeled, as described in Section 3, with an IDLE and a TX state, reducing its functionality to a minimum.

Now, applications were implemented to utilize the drivers: For the Sharp-96 display, we print text lines and scroll the display if necessary. The used dot-matrix display has no internal character set, and all font mapping must be done in software. This requires a considerable amount of CPU time.

For the CC1200 driver, we created an application that sends different packages with changing idle pauses in a loop.

The energy consumption of the experiments was continuously observed by external energy measurements. In parallel, accounted energy values were continuously output at designated, energy neutral code locations via UART. During UART transfers, the system timers were frozen to keep the accounting consistent.

In the following, we present two experiments:

1. We used the `print` function to repeatedly write three text lines to the display, causing a lot of transfers by the aforementioned scroll functionality. Each write cycle is followed by an idle phase that lets the display hold the image. The idle phase was continuously scaled from 1 ms to 1 s by each iteration, to simulate a set of different usage profiles. The whole experiment took 30 min. For this experiment, Figure 3 and Figure 4 show the results for display and CPU, while Figure 5 depicts the accuracy.
2. Clusters of data packages were sent using the CC1200, each cluster starting from packages with 8 bytes payload and increasing to 48 bytes, then decreasing back to 8 bytes. Every cluster uses a different idle time between each package, starting from 1 ms to 29 ms. Every cluster is sent three times in sequence. The energy consumption of the CC1200 is plotted in Figure 6, the relative accuracy in Figure 7. We left out the CPU energy consumption, as the results were similar to those in the former experiment. The whole experiment took 140 s.

Table 2 shows the minimum and maximum relative error and accuracy for the components in both experiments, considering the relative error calculated as Equation 3 and the accuracy as $1 - \varepsilon_{total}(n)$.

$$\varepsilon_{total}(t) = \left| \frac{p(t)}{m(t)} - 1 \right| \tag{3}$$

4.4 Time and Energy Overhead of the Accounting

To measure the timing overhead imposed by the accounting, we used the internal timers of the CPU and the external energy measurement equipment by signaling events using a GPIO line. Both methods showed an additional CPU utilization of 58 µs on a 16 MHz

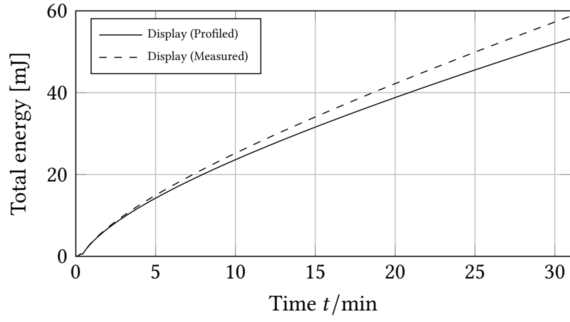

Figure 3: Experiment 1: Display energy consumption at repeated heavy duty write cycles alternated by increasing hold times.

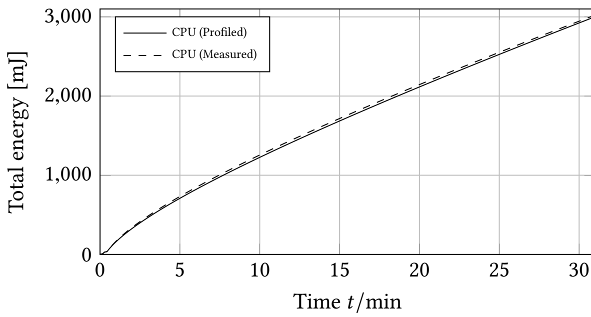

Figure 4: CPU energy consumption for experiment 1.

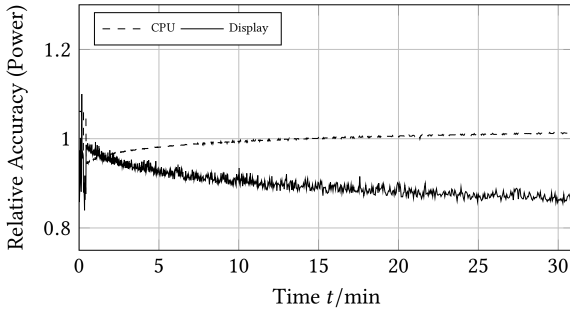

Figure 5: Model accuracy for experiment 1.

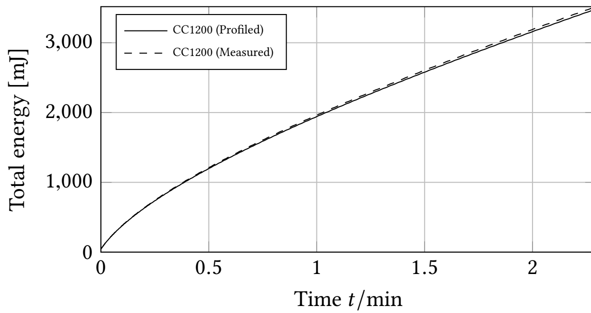

Figure 6: CC1200 energy consumption for experiment 2

	error		accuracy	
Component	min	max	min	max
CPU	0.01%	5.9%	94.1%	99.99%
Display	2.9%	9.8%	90.2%	92.1%
CPU+Display	0.1%	5.6%	94.4%	99.9%
CC1200	0.00%	2.8%	97.2%	100.0%

Table 2: Minimum and maximum cumulative error and accuracy during both experiments.

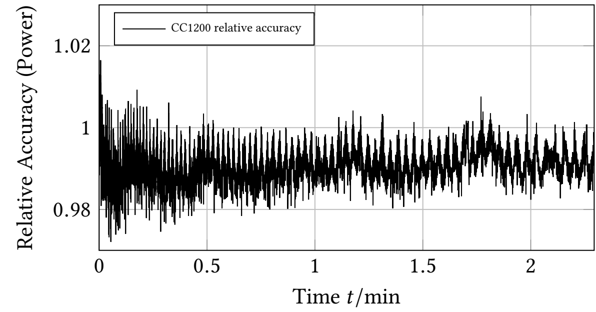

Figure 7: Model accuracy for experiment 2

driven MSP430-FR5969, or 928 CPU cycles, which consumed an average of 138 nJ of energy.

Using the CC1200 application from the previous section, we made 14268 transition within the radio driver to transmit 7134 packages. By that, the overhead of the accounting implementation sums up to 1.9 mJ. This is 0.04% of the overall energy consumption of 4.07 J during that time. In this situation, we can consider the driver as beneficial, assuming that the information gathered by the accounting leads to intelligent run-time decisions on a higher level power-management system.

For the Sharp-96 display, 10 million driver calls were issued to redraw pixel rows on the screen over a 30 minute run. This sums the energy consumption of the accounting to about 1400 mJ. This was approximately 46% of the system's energy consumption of 3059 mJ. Power management cannot benefit here in general due to the almost constant energy consumption of the display. Additionally, the display's energy consumption is well below that of the CPU, so that every additional line of code imposes high impacts. Anyhow, the overhead still can be greatly reduced by bundling driver calls, so that the whole display is updated at once instead of issuing one call per pixel line.

Here we see a clear benefit of energy profiling drivers. When used during the development process, they can quickly deliver valid and accurate energy values for the whole device and for every component. This allows for quick code optimization. After optimization of the code, the energy accounting module can be removed for overhead sensitive drivers.

	transitions	driver overhead	overall consumption	%
CC1200	14268	1.9 mJ	4070 mJ	0.04
Display	$\approx 10^6$	1400 mJ	3059 mJ	45.77

Table 3: Energy consumptions and overheads of both experiments

5 CONCLUSION

In this paper we presented the implementation of energy-aware drivers for peripheral devices and CPUs in deeply embedded system environments. The drivers are based on approaches to model the functional and non-functional properties of these devices as extended priced timed automata. We presented an implementation of our concept on a ultra low-power embedded system. The implementation is modular and can be configured to include an accounting mechanism that tracks the energy consumption of a selected set of devices. This enables algorithms on top of the drivers to make energy-based decisions at runtime.

To prove our concept and point out costs and benefits, we made a practical evaluation on an MSP430 utilizing the CPU, a display and a radio transceiver. We show that the implementation of the driver has a small memory overhead, which was 4 bytes of static/constant memory per state and transition of the underlying model. The implementation of an energy accounting mechanism based on our driver model has an overhead of about 22 bytes of code memory per transition, plus 470 bytes per driver, plus 18 bytes for variables.

To show the accuracy of our approach, the accounted energy consumption within the drivers was compared to actual measurements taken at the same time. We show that the maximum error in 30 minutes of heavy display access is 5.9% for the CPU, 9.8% for the display and 5.6% in sum. The error of the energy accounting for the radio transceiver is less than 2.9%.

Finally, we discussed the energy overhead of the accounting. Here, we pointed out that profiling has an energy overhead of 0.04% for sending data over a typical sensor node transceiver. In contrast, the display experiment had an energy overhead of 46% compared to the overall energy consumption of the system, because the display itself has a mostly constant energy consumption which is far below that of the CPU.

Our concept and its proof implementation clearly show that the implementation of automata based, energy-aware drivers is feasible at low costs. However, there are devices that do not benefit from software driven power management solutions, e.g. if their energy consumption is constant. Especially, when the energy used by a device is less than that of the CPU, the overhead of energy management code can become malicious.

6 ACKNOWLEDGMENTS

This work was supported by the German Research Council (DFG) within the Collaborative Research Center SFB 876, project A4.

REFERENCES

[1] R. Alur, S. La Torre, and G.J. Pappas. 2001. Optimal paths in weighted timed automata. In *Hybrid systems: computation and control*. Springer, 49–62.

[2] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager. 2001. *Minimum-cost reachability for priced time automata*. Springer.

[3] M. Buschhoff. 2014. KRATOS - A Resource Aware, Tailored Operating System. In *Technical report for Collaborative Research Center SFB 876 - Graduate School*, Katharina Morik and Wolfgang Rhode (Eds.). Number 10.

[4] M. Buschhoff, C. Günter, and O. Spinczyk. 2012. A unified approach for online and offline estimation of sensor platform energy consumption. In *8th International-Wireless Communications and Mobile Computing Conference (IWCMC)*. 1154–1158. https://doi.org/10.1109/IWCMC.2012.6314369

[5] M. Buschhoff, C. Günter, and O. Spinczyk. 2014. MIMOSA, a Highly Sensitive and Accurate Power Measurement Technique for Low-Power Systems. In *Real-World Wireless Sensor Networks*, Koen Langendoen, Wen Hu, Federico Ferrari, Marco Zimmerling, and Luca Mottola (Eds.). Lecture Notes in Electrical Engineering, Vol. 281. Springer International Publishing, 139–151. https://doi.org/10.1007/978-3-319-03071-5_16

[6] H. Chen, G. Godet-Bar, F. Rousseau, and F. Petrot. 2011. Me3D: A model-driven methodology expediting embedded device driver development. In *22nd IEEE International Symposium on Rapid System Prototyping (RSP)*. 171–177. https://doi.org/10.1109/RSP.2011.5929992

[7] S.K. Datta, C. Bonnet, and N. Nikaein. 2012. Android power management: Current and future trends. In *Enabling Technologies for Smartphone and Internet of Things (ETSIoT), 2012 First IEEE Workshop on*. IEEE, 48–53.

[8] S. Kellner and F. Bellosa. 2009. Energy accounting support in tinyos. *PIK-Praxis der Informationsverarbeitung und Kommunikation* 32, 2 (2009), 105–109.

[9] S. Kellner, M. Pink, D. Meier, and E.-O. Blass. 2008. Towards a Realistic Energy Model for Wireless Sensor Networks. In *Fifth Annual Conference on Wireless on Demand Network Systems and Services*. 97–100. https://doi.org/10.1109/WONS.2008.4459362

[10] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. 2000. Devil: An IDL for hardware programming. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*. USENIX Association, 2–2.

[11] C. Seceleanu, A. Vulgarakis, and P. Pettersson. 2009. REMES: A Resource Model for Embedded Systems. In *14th IEEE International Conference on Engineering of Complex Computer Systems*. 84–94. https://doi.org/10.1109/ICECCS.2009.49

[12] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. 2001. An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations. In *PATMOS*. Yverdon (Switzerland).

[13] T.K. Tan, A. Raghunathan, and N.K. Jha. 2002. Embedded operating system energy analysis and macro-modeling. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*. 515–522. https://doi.org/10.1109/ICCD.2002.1106822

[14] O. Wang and W. Yang. 2007. Energy Consumption Model for Power Management in Wireless Sensor Networks. In *SECON '07. 4th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*. 142–151. https://doi.org/10.1109/SAHCN.2007.4292826

[15] S. Wang and S. Malik. 2003. Synthesizing operating system based device drivers in embedded systems. In *Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on*. 37–44. https://doi.org/10.1109/CODESS.2003.1275253

[16] A. Weder. 2010. An Energy Model of the Ultra-Low-Power Transceiver nRF24L01 for Wireless Body Sensor Networks. In *Second International Conference on Computational Intelligence, Communication Systems and Networks (CICSyN)*. 118–123. https://doi.org/10.1109/CICSyN.2010.24

[17] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. 2012. AppScope: Application Energy Metering Framework for Android Smartphone Using Kernel Activity Monitoring.. In *USENIX Annual Technical Conference*. 387–400.

[18] H. Zeng, C.S. Ellis, A.R. Lebeck, and A. Vahdat. 2002. ECOSystem: Managing energy as a first class operating system resource. *ACM SIGPLAN Notices* 37, 10 (2002), 123–132.

[19] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R.P. Dick, Z.M. Mao, and L. Yang. 2010. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 105–114.