BSN Simulator: Optimizing Application Using System Level Simulation

Ioana Cutcutache Thi Thanh Nga Dang Wai Kay Leong Shanshan Liu Kathy Dang Nguyen Linh Thi Xuan Phan Edward Sim Zhenxin Sun Teck Bok Tok Lin Xu Francis Eng Hock Tay Weng-Fai Wong
National University of Singapore
kathyngu@comp.nus.edu.sg

***Abstract*—A biomonitoring application running on wireless BAN has stringent timing and energy requirements. Developing such applications therefore presents unique challenges in both hardware and software designs. This paper shows how we successfully apply our full-system simulator to a MEMSWear-Biomonitoring application. The simulation results, together with a set of investigative guidelines, enable us to identify and overcome performance bottlenecks. Our simulator is able to obtain timing and energy measurements for each function in the program as well as for each module in the hardware. Without such detail and accurate information, we would not be able to identify the reason for the low performance in the original application.**

I. Introduction

Developing and tuning an application over a body area wireless sensor network (BAN) is an unique challenge due to conflicting trade-offs in its system requirements: it needs high processing power to meet certain computation goals, and it also needs to consume less energy in order to preserve battery life. Consequently, the developer has to design and optimize both hardware and software in order to build a fast and efficient system.

We have argued that a full system simulator is more suitable in helping the developers achieve their goals, especially when the hardware platform is not yet available at the design stage, or when fine-grained simulation results are needed to tune the system [10]. Particularly, the breakdown in the timing and energy consumption by each function in the application and by each module in the hardware are critical information needed by the developers. It also allows quick design space exploration when there are multiple setup configurations.

We have developed a SystemC-based fast simulator for biomonitoring applications running on a wireless BAN. Our work is part of Embedded and Hybrid System program in Singapore, centered on the development of a Body Sensor Node (BSN) system and of relevant health-care applications. The project is carried out by Singapore Agency of Science Technology and Research (A*STAR) with collaborations from Institute of Infocomm Research (I2R), Institute of Microelectronics (IME), Nanyang Technological University (NTU) and National University of Singapore (NUS).

In this paper we present a case study of how our full system simulator, together with a set of investigative guidelines, help us in identifying and fixing performance issues on a MEMSWear-Biomonitoring application called SpO_2nECG application [20]. We use physical measurements on the timing and energy consumption to validate the results of our simulation. Based on the simulation results, we are able to identify the reason why the messages are sent at a much lower rate than expected, as well as discover a very efficient mechanism to reduce energy consumption and hence improve battery life. Without the full system simulation, the performance bottleneck would be impossible to detect by just merely looking at the application code: the issue lies with how it interacts with the underlying TinyOS code.

The rest of the paper is organized as follows. We briefly review related work in Section II. In Section III we briefly describe our full-system simulator. Section IV presents our case study with the SpO_2nECG application. Section V concludes the paper.

II. Related Work

Numerous simulation tools are available to aid programmers in understanding the performance and behaviour of body sensor networks (BSNs). These tools vary widely in scalability, accuracy and feedback details. Surprisingly, relatively few exist for the purpose of timing and power analysis—the two most essential aspects in the design and optimisation of body sensor applications. In fact, to the best of our knowledge, none of the existing instruction-level simulation tools for MSP430 platform supports timing and power analysis of sensor motes at the functional level.

Well-known discrete event-based simulation environments such as NS-2 [2], TOSSIM [8], and OMNeT++ [23] provide effective ways to validate the behaviour of network protocols. However, they do not capture internal operations of the individual motes that might assist developers in debugging and optimizing applications.

Along the line of our work, there have been a number of instruction-level mote simulators; for example, Atemu [6], Avrora [21], COOJA [12] and their extensions. Atemu simulates the operations of individual motes and communication between them, though it does not supply timing and power consumption information of the motes. Similar to our simulator, Avora and its extension AEON [7] allow for the evaluation of energy consumption and lifetime prediction of sensor network. However, they are both limited to Mica2

978-0-7695-3644-6/09 $25.00 © 2009 IEEE
DOI 10.1109/P3644.21

platform, which is not applicable to our context where our focus is on the MSP430 platform.

Eriksson et al. [1] introduces an instruction level simulator called MSPsim, targeting the MSP430 microcontroller that contains a sensor board simulator as well which simulates hardware peripherals such as sensors, communication ports, LEDs, and sound devices. Although comprehensive in features and easy to be integrated into the cross-level simulation platform COOJA, MSPsim shares the same limitation as Atemu: it supports source-level stepping and run-time variable inspection only, without displaying any timing or power consumption information of the various components of a mote.

Recently, TOSSIM has been extended to estimate the power consumption of the Mica2 sensor mote. The extension, PowerTOSSIM [17], is built on top of TinyOS and is highly scalable. This benefit comes at a price as the tool is coupled with the TinyOS written code. In contrast, our simulator works on machine code level, and it can simulate sensor network applications written in any language with any operating system components, such as both TinyOS and SOS. Further, PowerTOSSIM follows a high-level of abstraction approach, which might not provide enough details necessary for application optimisation and may potentially lead to poor accuracy in the analysis results.

Several other tools that work at the abstract level, for example, SensorSim [13] and SENS [19], unfortunately, assume rather simplistic power usage and battery models which may not be realistic in actual hardware and practical applications.

In the Embedded System domain, several power simulation tools for energy profiling have been proposed (see e.g. [5], [18]). However, most of these tools are limited to profiling of microprocessor energy consumption only as they are designed for general embedded systems,

Despite simulation being the de-facto standard tool for the evaluation of body sensor networks, lately there is a growing interest in the formal method community. Owing to their expressiveness, automata-based techniques have been used to analyse wireless communication and protocols (see e.g. [3], [4]). Recently, Timed-automata has been employed to validate QoS properties of BSNs such as packet end-to-end delay, packet delivery ratio and network connectivity [22].

Another contrasting line of work is the Sensor Network Calculus (SNC) [15], a worst-case analysis framework that uses algebraic techniques. It is developed based on the Network Calculus in Computer Networks domain. SNC has been continuously adapted and extended to effectively model and analyse worst-case behaviour of sensor networks, such as these examples in [9] and [16].

Although comparatively much less explored and limited by their ability to scale up, formal techniques are much faster than simulation and provide formal guarantees. Thus, they can be used in early development, for instance to identify worst-case scenarios of the systems for a given architecture. With further investigation and appropriately incorporated with simulation-based techniques, such formal techniques will certainly benefit system designers and developers to a great extent.

III. System Level Simulation

A. Simulator

We implemented a fast, cycle-accurate simulator for biomonitoring applications. This simulator allows the developer to determine accurately the processing and energy performance of individual modules in the application, under different configurations. The simulator assumes the applications utilize multiple motes which are connected via a wireless BAN. Following is a brief description of our simulator; for a full description please refer to [10].

The mote simulator is implemented in SystemC and it takes in any application code written in NesC. In order to handle a network of sensor motes, multiple instances of these simulators are created as different threads in the same simulation process.

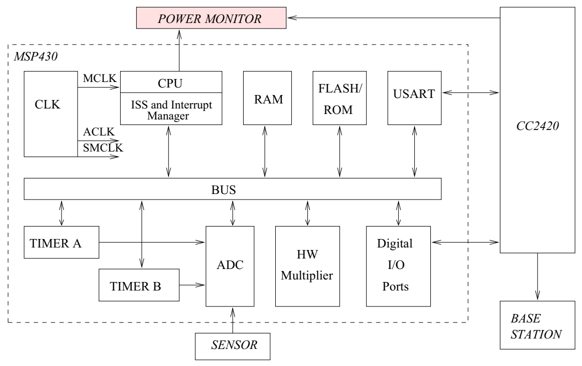


Fig. 1. The structure of the mote simulator.

Figure 1 shows the general structure of our mote simulator. It consists of four main components: a micro-controller module, a ChipCon 2420 module, a sensor module, and a power monitor.

The micro-controller in the BSN IC mote is Texas Instruments MSP430. This module incorporates a CPU module, a clock module, RAM, Flash and other peripherals. The CPU module includes an instruction set simulator and an interrupt manager.

The CC2420 module is an abstraction of the real radio chip. It provides only the basic functionalities, as our focus is to capture information on timing and power consumption.

The power monitor is not part of the MSP430 architecture and it is designated solely to monitor the power consumed by each component of the mote. It is also able to compute the energy consumed by each function of the application being simulated. This information is very useful for determining the functions that need to be optimized in order to reduce the energy consumption.

B. Application

Our simulator provides simulation for the full system of a generic wireless BAN. Typically, a wireless BAN consists of several wearable sensors on a human body. These sensor motes transmit vital body parameters such as ECG or SpO_2 blood level through wireless technology to a gateway mote which in turn is connected to a PDA, whose job is to process data, send commands to the motes, and forward data to a doctors clinic. Additional information on full system simulation can be found in our previous work [11].

The users interact with our simulator through a friendly Graphical User Interface. To simulate an application, the users follow these steps:

- Choose the number of sensor motes.
- Choose the application running on the gateway mote and on each sensor mote.
- Start the simulation. While running, the GUI displays the PDAs screen and a multi-tab window for the motes. For example, the PDA screen can show graphs of data received from the motes. Each tab in the sensor window shows the current status of the corresponding mote.
- Stop the simulation. A summary of total energy and timing for each mote is showed on its corresponding window tab.

C. Functionalities

Our simulator provides the following functionalities to help developers debug or optimize their applications:

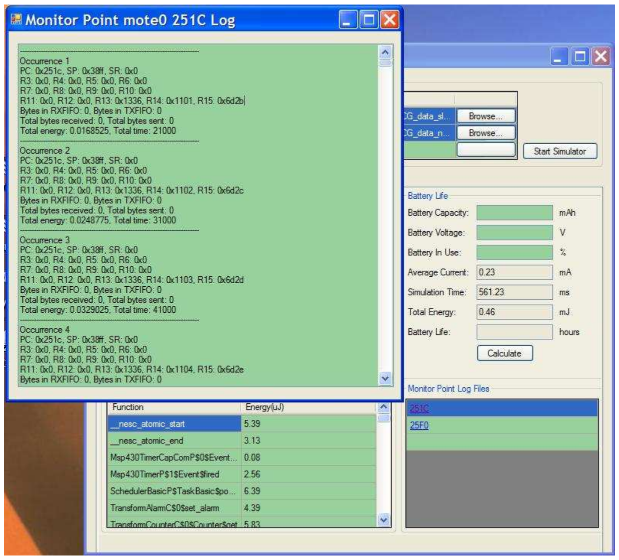
a) Total Energy and Timing: After simulating an application, the total energy and timing are shown. This information is extremely important to evaluate the overall performance of an application.

b) Functions Energy and Timing: A breakdown of timing and energy for each application function running on each mote is also listed. These numbers enable developers to optimize the application by providing deep insights into the behaviour of the application and help them focus effectively on the most time-consuming or energy-consuming functions.

c) Monitoring points: The user can choose several critical points in a program as monitoring points. During simulation, whenever the execution reaches these monitoring points, an instant snapshot of the following information is automatically saved in a log file:

- Energy and timing: This is the energy and time consumed so far by that mote. It gives user a quick look at the rate the code running on the mote is consuming power.
- Register values: The values of CPUs registers are shown for functional debugging. For example, a jump instruction whose destination is indexed mode jumps to different locations depending on the run-time value of the indexing register. Thus, the value of that indexing register could tell which instruction and/or function the simulator is going to execute next. This knowledge is useful for user to validate the control flow.
- Radio buffer: The number of filled bytes in CC2420s transmitting and receiving buffers are captured and displayed. This helps detecting buffer overflow and dropped messages, which enable the developer to adjust the message sizes and the sampling rates.
- Transmission summary: the snapshot provides a count of bytes sent or received so far by the radio chip and the serial port. These data, together with timing information, can be used to compute the average transmission rate.

Figure 2 shows an example of the collected data at various monitor points. This is particularly useful when the user wants to compute time or energy for a fragment of code which could be part of a function or a combination of several continuous functions.



```
----------------------------------------------------
Occurrence 1
PC: 0x251c, SP: 0x38ff, SR: 0x0
R3: 0x0, R4: 0x0, R5: 0x0, R6: 0x0
R7: 0x0, R8: 0x0, R9: 0x0, R10: 0x0
R11: 0x0, R12: 0x0, R13: 0x1336, R14: 0x1101, R15: 0x6d2b
Bytes in RXFIFO: 0, Bytes in TXFIFO: 0
Total bytes received: 0, Total bytes sent: 0
Total energy: 0.0168525, Total time: 21000
----------------------------------------------------
Occurrence 2
PC: 0x251c, SP: 0x38ff, SR: 0x0
R3: 0x0, R4: 0x0, R5: 0x0, R6: 0x0
R7: 0x0, R8: 0x0, R9: 0x0, R10: 0x0
R11: 0x0, R12: 0x0, R13: 0x1336, R14: 0x1102, R15: 0x6d2c
Bytes in RXFIFO: 0, Bytes in TXFIFO: 0
Total bytes received: 0, Total bytes sent: 0
Total energy: 0.0248775, Total time: 31000
----------------------------------------------------
Occurrence 3
PC: 0x251c, SP: 0x38ff, SR: 0x0
R3: 0x0, R4: 0x0, R5: 0x0, R6: 0x0
R7: 0x0, R8: 0x0, R9: 0x0, R10: 0x0
R11: 0x0, R12: 0x0, R13: 0x1336, R14: 0x1103, R15: 0x6d2d
Bytes in RXFIFO: 0, Bytes in TXFIFO: 0
Total bytes received: 0, Total bytes sent: 0
Total energy: 0.0329025, Total time: 41000
----------------------------------------------------
Occurrence 4
PC: 0x251c, SP: 0x38ff, SR: 0x0
R3: 0x0, R4: 0x0, R5: 0x0, R6: 0x0
R7: 0x0, R8: 0x0, R9: 0x0, R10: 0x0
R11: 0x0, R12: 0x0, R13: 0x1336, R14: 0x1104, R15: 0x6d2e
Bytes in RXFIFO: 0, Bytes in TXFIFO: 0
```

Battery Capacity:		mAh
Battery Voltage:		V
Battery In Use:		%
Average Current:	0.23	mA
Simulation Time:	561.23	ms
Total Energy:	0.46	mJ
Battery Life:		hours

Function	Energy(uJ)
__nesc_atomic_start	5.39
__nesc_atomic_end	3.13
Msp430TimerCapComP$0$Event...	0.08
Msp430TimerP$1$Event$fired	2.56
SchedulerBasicP$TaskBasic$po...	6.39
TransformAlarmC0set_alarm	4.39
TransformCounterC$0$Counter$get	5.83

Monitor Point Log Files: 251C, 25F0

Fig. 2. Simulation data collected at monitor points.

d) Monitoring memory addresses: The user can choose several important memory addresses in a program to monitor their evolution. These addresses may refer to variables used by the application or they may correspond to the internal registers of various peripherals. During simulation, whenever the value of such a monitored address is modified the new value and the address of the current instruction are logged to a file.

e) Battery life computation: We employ Rakhmatov's battery model [14] to capture the battery performance and to compute its life time. Since BAN applications are on human body, they require mobility, flexibility and long duration. Thus, the battery life time is a critical factor while developing the applications; the longer the battery can last, the more convenient it is for device-wearing patients.

f) Control flow: Last but not least, our simulator provides users with an option to record the control flow of the application running on a mote in a log file. Since applications written in NesC code are compiled with TinyOS library to generate complete C code and then executable code on a specific platform, details of an applications control flow is only available at C/assembly code level which is significantly different and much more comprehensive than the original NesC code. Understanding the control flow of an application is vital for debugging and optimizing purposes.

D. *Guideline to debug/optimize an application*

We give a general list of guidelines which the developers can adopt in the process of debugging or optimizing applications. These guidelines are created based on our experiences and on the general principle of top-down problem solving. The steps and their order are flexible and could be modified to suit different applications, problems and objectives.

a) First, the user should set up and run the application once to get a summary of the total time and energy consumed by the application for a standard sensor input. Together with a breakdown of timing and energy for each function, this gives a comprehensive overview of the application. Based on this overview, the user might have a general idea if a problem exists and what the problem is.
b) Other information such as battery life and transmission rate could also be a good source to detect the problem.
c) Look at the generated C code or control flow log file to understand the application and how different functions are linked together either as caller-callee relationship or as a sequence of tasks/events.
d) Check the control flow log file if everything is in order. If something goes wrong, the control flow may show some unexpected runtime behaviour.
e) With a good understanding of the application control flow, the user can identify critical points in the program where local information may reveal the cause of the problem. She can run the whole simulation once more to collect data at these monitoring points. Then she may investigate the result to narrow down the problem.
f) If the problem is with application code, fix or improve it. If the problem is with TinyOS code, consult TinyOS developer guide to find if there is any solution or possible optimization. After fixing the code, go back to step a), run the simulation one more time to get the new performance summary. If the result is not satisfying, repeat the steps for further optimization.

Our simulator is very fast, incurring only 5–20 times slower than native execution on real motes yet able to provide critical performance data for developers to tune their applications. This is apparently an advantage compared to other methods. For example, running applications multiple times directly on the real hardware gives very little insight about the applications running on it.

In Section IV, we present an example of how we use our simulator to obtain useful data about the performance of an application and how the developer uses this information to identify and remove a bottleneck in the application.

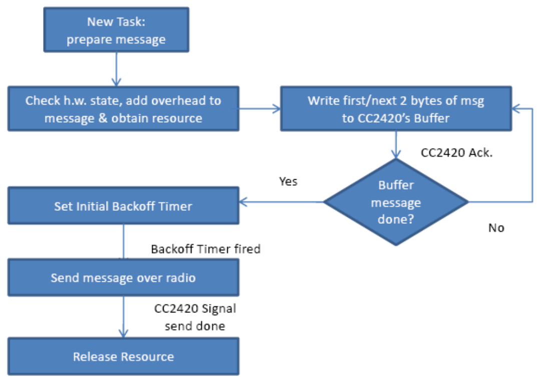

New Task: prepare message
Check h.w. state, add overhead to message & obtain resource
Write first/next 2 bytes of msg to CC2420's Buffer
CC2420 Ack.
Buffer message done?
Yes
No
Set Initial Backoff Timer
Backoff Timer fired
Send message over radio
CC2420 Signal send done
Release Resource

Fig. 3. Control flow of message transmission

IV. Case Study

In this section, we focus on a case study of a MEMSWear-Biomonitoring application, SpO_2nECG application [20]. In this application, a single sensor mote collects data such as RED, IR and ECG from attached sensor at a sampling rate of 250Hz and then sends them all to the gateway station (PDA) for processing. The gateway stattion uses these data as input to compute heart rate, SpO_2 and blood pressure. These outputs might be displayed on the LCD or to be used as input to other programs such as fall detection.

Following the guideline, first we run simulation for this application once to obtain a performance summary. We notice the total energy consumption is dominated by the radio on sensor mote, compared to those by the CPU and other peripheral devices. This leads to battery running out quickly in several hours while the developers expect it to last at least half day.

Next, checking other data in the summary, we find a problem with the transmission rate on sensor motes. Although the transmission rate for radio chip CC2420 is specified to be 250kb/s in the manual, this application achieves a much lower rate of 20kb/s. This causes a serious delay in collecting data, computing important output and detecting critical conditions on the PDA side.

After identifying the two problems, we follow the guideline to fix them.

a) Transmission rate: First, we try to optimize the low transmission rate. By investigating the C code and the generated control flow log file, we come to understand the control flow of the sending process shown in Figure 3. Note that this figure is just one simplified scenario applied for this application for illustration purpose; TinyOS has other scenarios to handle hardware or network congestion as well.
To gain more insight about this process, we perform another tip in the guidelines, which is to set up monitoring points in functions involved in message transmissions, and run the simulation to collect data at these points. From the collected data, we derive, as shown in Figure 4, the timing percentage of different tasks involved in

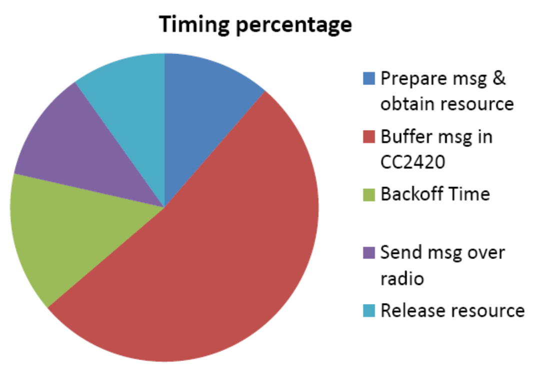


Fig. 4. Timing percentage of different tasks involved in transmission

sending a message. These are average numbers after transmitting over 100 messages.

The simulation result helps us narrow the problem and look into the right place to seek for improvement. Contrast to our initial intuition, actual sending data through radio costs only 11.6% of the total time. Indeed, our simulation brings to our attention that the most time-consuming task is buffering message in CC2420. In this case, the bottleneck lies in TinyOS code rather than the application code, which is impossible to detect by just merely looking at NesC code. From the control flow, we notice that CPU sends only 2 bytes each time to buffer in CC2420, waits for acknowledgment, before continuing with the remaining of a message. This is very time-consuming. Our solution is to change the TinyOS code so that each message is sent as a whole, and thus avoid waiting for (too many) acknowledgment.

After implementing this change and running the simulation once time, the transmission rate improves by 20 percents. However, note that in general this solution is not a good idea because removing synchronous mechanism might cause out-of-synch in hardware state and packet loss.

Another idea that the developers suggested is to move part of data processing from the gateway mote to the sensor mote. Instead of collecting and sending raw data to the gateway mote, the sensor mote can actually extract important information and send only critical data to the gateway mote for further action. This has two advantages: reducing network workload and packet loss, and saving energy for the radio chip CC2420 by sending fewer and shorter messages.

This second suggestion proves itself to be a very good idea. Because the sensor mote now scarcely send critical data to the gateway mote, the slow transmission for a message is tolerable. It also balances the workload between the sensor mote and gateway mote, and reduces the chance of packet loss due to buffer overflow. Energy consumed by the gateway mote drawn from the PDA is also reduced, and hence the PDA could last longer for its other tasks.

b) Radio Energy: Next we analyze the second problem, energy consumed by the SpO_2nECG application. Through summary of simulation result and extrapolation, we know that battery will get exhausted quickly due to high consumption rate of the radio chip. Again, following tip (c) of the guideline, inspecting the radio chip manual and the control flow log file, we find that the reason is that the radio is continuously active in its sending/receiving mode.

At first, since the sensor mote keeps sending data at a rate of 250 messages per second, it is reasonable to keep the radio on all the time to avoid message dropping. However, after the first optimization, the motes no longer need to send messages at such high rate. Because our simulator is able to accurately measure the interval between message sending events, the new simulation result shows that the sensor mote now sends critical messages at peak point of data about every 300ms. Therefore, we suggest to the developers to turn off the radio during inactive periods.

Using the Low-Power Listening Interface that comes with TinyOS, we put the radio to sleep whenever there is no packet to send. The amount of on-time and sleep-time of the radio in each cycle was controlled by the local duty cycle rate, which is the ratio between the radio-on time to the total cycle time:

$$DutyCycleRate = \frac{OnTime}{OnTime + SleepTime}$$

Our objective is to obtain as small duty cycle as possible while ensuring there is no message loss. Toward this, we experimented with a wide range of duty cycle values and simulated each corresponding modified application using our simulator. Based on the simulated analysis, we chose the value that produced the least energy consumption and an acceptable end-to-end delay of the messages.

After implementing this Low Power Listening on radio chip CC2420, which enables us to turn off the radio during inactive periods, the total consumed energy computed by our simulator is reduced significantly by nearly 40%. We verify the result with real measurement, and we found that there is a small gap. This is expected because we have not implemented the LEDs, voltage converter, and battery adaptor in our simulator, and hence we did not estimate the energy consumed by them in our energy computation. Currently we are trying to close the gap by improving the simulator.

These optimizations not only benefit the SpO_2nECG application but several other groups in EHS project as well. They applied these optimizations in their design and gained significant improvement in their applications performance.

V. Conclusion

In this paper we have described a case study of how we successfully use our full-system simulator to dissect the performance of a MEMSWear-Biomonitoring application. Base on the detail results on timing and energy, we are able to identify bottlenecks in the application. The bottlenecks are contrary to our initial suspicious culprits. The results also enable us to come up with solutions that significantly improve the performance, which in turn improve the life time of the batteries.

Acknowledgments: We would like to thank Tulika Mitra, Samarjit Chakraborty, Abhik Roychoudhury, P.S. Thiagarajan for their advice.

References

[1] J. Eriksson, A. Dunkels, N. Finne, F. Osterlind, and T. Voigt. MSPsim - an extensible simulator for msp430-equipped sensor boards. In *Proc. of the European Conference on Wireless Sensor Networks (EWSN)*, 2007.

[2] K. Fall and K. Varadhan. The network simulator NS-2. `http://www.isi.edu/nsnam/ns/`.

[3] A. Fehnker, L.F.W. van Hoesel, and A. H. Mader. Modelling and verification of the lmac protocol for wireless sensor networks., 2007.

[4] M. Fruth. Probabilistic model checking of contention resolution in the IEEE 802.15.4 low-rate wireless personal area network protocol. In *Proc. of the 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, 2006.

[5] T. K. Tan I, A. Raghunathan, and N. K. Jha. Emsim: An energy simulation framework for an embedded operating system, 2002.

[6] M. Karir. Atemu - sensor network emulator / simulator / debugger. `http://www.hynet.umd.edu/research/atemu/`.

[7] O. Landsiedel, K. Wehrle, B. Titzer, and J. Palsberg. Enabling detailed modeling and analysis of sensor networks. *Praxis der Informationsverarbeitung und Kommunikation*, 28(2):101–106, April 2005.

[8] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire tinyos applications. In *Proc. of the 1st ACM Conference on Embedded Networked Sensor Systems (SenSys).*, 2003.

[9] L. Lopes, F. Martins, M.S. Silva, and J. Barros. A formal model for programming wireless sensor networks. In *Proc. of the International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2007.

[10] Kathy Dang Nguyen, Ioana Cutcutache, Saravanan Sinnadurai, Shanshan Liu, Cihat Basol, Edward Sim, Linh Thi Xuan Phan, Teck Bok Tok, Lin Xu, Francis Tay Eng Hock, Tulika Mitra, and Weng-Fai Wong. Fast and accurate simulation of biomonitoring applications on a wireless body area network. In *5th International Workshop on Wearable and Implantable Body Sensor Networks (BSN 2008)*, pages 145–148, June 2008.

[11] Kathy Dang Nguyen, Ioana Cutcutache, Saravanan Sinnadurai, Liu Shanshan, Basol Cihat, Adrian Curic, Tok Teck Bok, Lin Xu, Francis Tay Eng Hock, and Tulika Mitra. A systemc-based fast simulator for biomonitoring applications on wireless ban. In *Workshop on Software and Systems for Medical Devices and Services 2007 (SMDS 2007)*, December 2007.

[12] F. Österlind. The COOJA simulator - user manual. `http://www.sics.se/ fros/cooja`.

[13] S. Park, A. Savvides, and M. B. Srivastava. SensorSim: A simulation framework for sensor networks. In *Proc. of the 3th ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWIM)*, 2000.

[14] D. Rakhmatov, S. Vrudhula, and D.A. Wallach. A model for battery lifetime analysis for organizing applications on a pocket computer. In *Proc. IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2003.

[15] J.B. Schmitt and U. Roedig. Sensor network calculus - a framework for worst case analysis. *Distributed Computing in Sensor Systems*, 3560:141–154, 2005.

[16] Jens B. Schmitt, Frank A. Zdarsky, and Lothar Thiele. A comprehensive worst-case calculus for wireless sensor networks with in-network processing. In *RTSS '07: Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 193–202, Washington, DC, USA, 2007. IEEE Computer Society.

[17] V. Shnayder, M. Hempstead, B. Chen, G.W. Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proc. of the 2st ACM Conference on Embedded Networked Sensor Systems (SenSys).*, 2004.

[18] A. Sinha and A.P. Chandrakasan. JouleTrack: a web based tool for software energy profiling. In *Proc. of the 38th Conference on Design automation (DAC)*, pages 220–225, New York, NY, USA, 2001. ACM.

[19] S. Sundresh, W. Kim, and G. Agha. SENS: A sensor, environment and network simulator. In *Proc. of the 37th Annual Symposium on Simulation (ANSS)*, page 221, Washington, DC, USA, 2004. IEEE Computer Society.

[20] F. E. H. Tay, D. G. Guo, L. Xu, M. N. Nyan, and Yap K. L. Memswear-Biomonitoring System for Remote Vital Signs Monitoring. In *Procs. 4th International Symposium on Mechatronics and Its Applications (ISMA07)*, 2007.

[21] B.L. Titzer, D.K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Proc. of the 4th International Symposium on information Processing in Sensor Networks (IPSN)*, page 67, Piscataway, NJ, USA, 2005. IEEE Press.

[22] S. Tschirner, L. Xuedong, and W. Yi. Model-based validation of QoS properties of biomedical sensor networks. In *Proc. of the 7th ACM international conference on Embedded software (EMSOFT)*, pages 69–78, New York, NY, USA, 2008. ACM.

[23] A. Varga. OMNet++: Discrete event simulation system. `http://www.omnetpp.org/`.