Simulating the Power Consumption of Large-Scale Sensor Network Applications

Victor Shnayder, Mark Hempstead, Bor-rong Chen, Geoff Werner Allen, and Matt Welsh
Division of Engineering and Applied Sciences
Harvard University
{shnayder,mhempste,brchen,werner,mdw}@eecs.harvard.edu

ABSTRACT

Developing sensor network applications demands a new set of tools to aid programmers. A number of simulation environments have been developed that provide varying degrees of scalability, realism, and detail for understanding the behavior of sensor networks. To date, however, none of these tools have addressed one of the most important aspects of sensor application design: that of *power consumption.* While simple approximations of overall power usage can be derived from estimates of node duty cycle and communication rates, these techniques often fail to capture the detailed, low-level energy requirements of the CPU, radio, sensors, and other peripherals.

In this paper, we present PowerTOSSIM, a scalable simulation environment for wireless sensor networks that provides an accurate, per-node estimate of power consumption. PowerTOSSIM is an extension to TOSSIM, an event-driven simulation environment for TinyOS applications. In PowerTOSSIM, TinyOS components corresponding to specific hardware peripherals (such as the radio, EEPROM, LEDs, and so forth) are instrumented to obtain a trace of each device's activity during the simulation run. PowerTOSSIM employs a novel code-transformation technique to estimate the number of CPU cycles executed by each node, eliminating the need for expensive instruction-level simulation of sensor nodes. PowerTOSSIM includes a detailed model of hardware energy consumption based on the Mica2 sensor node platform. Through instrumentation of actual sensor nodes, we demonstrate that PowerTOSSIM provides accurate estimation of power consumption for a range of applications and scales to support very large simulations.

Categories and Subject Descriptors

I.6 [**Simulation and Modeling**]: General; B.8.0 [**Performance and Reliability**]: General; C.4 [**Performance of Systems**]: Modeling techniques

SenSys'04, November 3–5, 2004, Baltimore, Maryland, USA.

General Terms

Measurement, Performance, Experimentation, Verification.

Keywords

TOSSIM, power simulation, sensor networks, mica2 energy model.

1. INTRODUCTION

The challenges of developing, debugging, and evaluating complex sensor network applications demands a new set of tools to aid programmers. Simulation environments, such as ns2 [6], TOSSIM [13], and Atemu [11], provide varying degrees of scalability, realism, and detail for understanding the behavior of sensor networks. To date, however, none of these tools have considered one of the most important aspects of sensor application design: that of *power consumption.* While simple approximations of overall power usage can be derived from estimates of node duty cycle and communication rates, these techniques often fail to capture the detailed, low-level energy requirements of the CPU, radio, sensors, and other peripherals.

Understanding power consumption is especially critical for sensor networks operating on limited power reserves, such as batteries or solar cells. Sensor network designers need the ability to obtain accurate and dependable power consumption figures to tune their applications before deployment in real environments. Apart from aggregate power consumption over time, the *pattern* of power load is important to consider, as this affects the ability of the power source to deliver adequate energy over time. These challenges were apparent in the Great Duck Island sensor network deployment [24], in which nodes significantly underperformed their expected lifetimes.

Existing simulation environments allow researchers to study dynamics such as communication overheads within the network, but no comparable tools exist for power consumption. Moreover, simulating the behavior of each sensor node at the level of CPU cycles or radio bits does not scale for networks of considerable size (e.g., thousands of nodes). *In situ* measurement of power consumption is possible at small scales, for instance, by instrumenting specific nodes in a lab setup. However, such an approach does not provide a network-wide picture of power consumption which is necessary for analysis of hot spots, or of variations in power usage under changes in the environment or network topology.

In this paper, we present PowerTOSSIM, a scalable simulation environment for wireless sensor networks that pro-

vides an accurate, per-node estimate of power consumption. PowerTOSSIM is based on TOSSIM [13], an event-driven simulation environment for TinyOS [10] applications, and derives much of its value from the tools built up around the TinyOS environment. In PowerTOSSIM, TinyOS components corresponding to specific hardware peripherals (such as the radio, EEPROM, LEDs, and so forth) are instrumented to obtain a trace of each device's activity during the simulation run. To scale up to large numbers of sensor nodes, PowerTOSSIM runs applications as a native executable and does not directly simulate each node's CPU at the instruction level. Instead, PowerTOSSIM employs a code-transformation technique to estimate the number of CPU cycles executed by each node. Finally, the trace of each node's activity is fed into a detailed model of hardware energy consumption, yielding per-node energy consumption data. This energy model can be readily modified for different hardware platforms.

The contributions of this paper are as follows. We describe the design and implementation of PowerTOSSIM, as well as present a detailed energy profile of the Mica2 sensor node obtained via extensive application-level benchmarking. We validate the accuracy of PowerTOSSIM's power estimates against 15 TinyOS applications, demonstrating that PowerTOSSIM achieves within 0.45–13% of the true power consumption of nodes running an identical program. PowerTOSSIM is designed to scale to very large networks, and runs 20 times faster than Atemu [11], an instruction-level simulator for the AVR platform that runs TinyOS programs. PowerTOSSIM is the first scalable simulation environment for sensor networks that provides accurate power consumption data.

The rest of this paper is organized as follows. We first discuss motivation and related work in Section 2. In Section 3 we describe our hardware measurement configuration, benchmarking, and the resulting Mica2 energy model. In Section 4 we describe the design of TOSSIM and our modifications to it. Section 5 describes the implementation of PowerTOSSIM in detail, and Section 6 presents experiments to validate its accuracy and scalability. Section 7 describes future work and concludes.

2. BACKGROUND AND RELATED WORK

Understanding the performance and behavior of sensor networks requires simulation tools that can scale to very large numbers of nodes. Traditional network simulation environments, such as ns2 [6], are effective for understanding the behavior of network protocols, but generally do not capture the operation of endpoint nodes in detail. Also, while ns2 provides implementations of the 802.11 MAC/PHY layers, many sensor networks employ nonstandard wireless protocols that are not implemented in ns2.

A number of instruction-level simulators for sensor network nodes have been developed, including Atemu [11] and Simulavr [20]. These systems provide a very detailed trace of node execution, although only Atemu provides a simulation of multiple nodes in a networked environment. The overhead required to simulate sensor nodes at the instruction level considerably limits scalability. Other sensor network simulation environments include PROWLER [21], TOSSF [19] (based on SWAN [14]), SensorSim [17], and SENS [23]. Each of these systems provides differing levels of scalability and realism, depending on the goals for the simulation environment. In some cases, the goal is to work at a very abstract level, while in others, the goal is to accurately simulate the behavior of sensor nodes. Few of these simulation environments have considered power consumption. SensorSim [17] and SENS [23] incorporate simple power usage and battery models, but they do not appear to have been validated against actual hardware and real applications [18]. Also, SensorSim does not appear to be publically available.

Several power simulation tools have also been developed for energy profiling in the embedded systems community. EMSIM [25] is a simulator for an embedded OS based on the StrongARM microprocessor. It consists of a StrongARM instruction-set simulator and simulation models for memory, UART and other peripherals connected to the processor. JouleTrack [2] is another tool built for general embedded software energy profiling but it estimates only the microprocessor energy consumption. Although these tools show high accuracy in energy profiling, they are designed for simulating a single host's energy use only. From a sensor network perspective, a tool for efficient large scale profiling is desired.

EmStar [9] provides an emulation environment for sensor networks using either a simulated radio channel or real messaging on a ceiling array of nodes connected via a serial port multiplexer. EmStar allows sensor network applications to be developed in a friendly Linux-based environment, allowing one to leverage debugging and tracing features in standard operating systems. EmTOS [22] is a TinyOS emulation environment for EmStar. While not a simulation environment *per se*, EmStar and EmTOS provide a valuable framework for testing sensor network applications.

TOSSIM [13] provides a scalable simulation environment for sensor networks based on TinyOS [10]. Unlike machine-level simulators, TOSSIM compiles a TinyOS application into a native executable that runs on the simulation host This design allows TOSSIM to be extremely scalable, supporting thousands of simulated nodes. Deriving the simulation from the same code that runs on real hardware greatly simplifies the development process. TOSSIM supports several realistic radio-propagation models and has been validated against real deployments for several applications. However, TOSSIM does not provide any information on the power consumption, which is the key goal of this paper.

Most of the individual ideas in this work have been used before. Our CPU power estimation technique is similar to the basic block profile based macro modeling in Tan, et al.[26]. Trace based modeling and offline processing are also extremely common ideas in systems research [7]. Modeling power usage by keeping track of the state of various system components is also an established technique [5]. Our main contribution is the integration of many disjoint ideas into a single tool that enables accurate power simulation of sensor networks at a scale that was previously impossible to attain.

3. HARDWARE CHARACTERIZATION

In this paper, we focus on sensor networks based on small, low-power sensors such as the Mica2 sensor node, developed by UC Berkeley. This device consists of a 7.3 MHz ATmega128L processor, 128KB of code memory, 512KB EEPROM, 4KB of data memory, and a ChipCon CC1000 radio capable of transmitting at 38.4 Kbps with an outdoor transmission range of approximately 300m. The device measures 5.7cm × 3.1cm × 1.8cm and is typically powered by

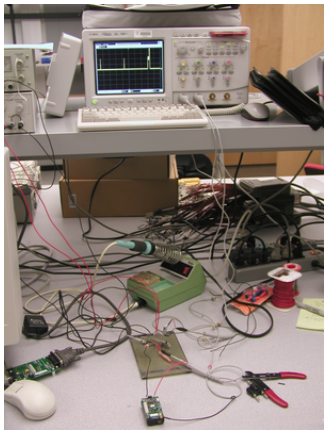


Figure 1: Our hardware measurement configuration. ***Our setup includes an oscilloscope, instrumentation amp, power supply, mote, PC, and programming board. The instrumented mote is at the bottom center of the picture.***

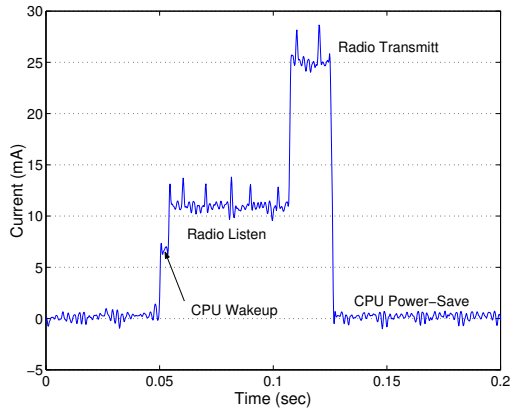


Figure 2: Measured current consumption for transmitting a single radio message at maximum transmit power on the Mica2 node.

2 AA batteries. The expected lifetime varies from days to months, depending on application duty cycle. The Mica2 can be interfaced to a number of sensors such as photoresistors, thermistors, passive infrared, magnetometers, and accelerometers.

The first challenge in simulating the power consumption of sensor network devices is obtaining an accurate, detailed model of a typical node's power consumption. We have conducted extensive power profiling of the Mica2 platform using the standard Mica2 sensor board equipped with a photoresistor, temperature sensor, microphone, sounder, and tone detection circuit. In this section we describe our measurement setup and methodology, and present the resulting power model.

3.1 Measurement setup

Our measurement setup is depicted in Figure 1. An oscilloscope was chosen to measure current with the highest possible resolution in order to reveal all power transitions. The oscilloscope used was an Agilent Infiniium 54832B[1] with 10:1 passive probes. The oscilloscope provides a 1 GHz sampling resolution. All data was logged digitally for later analysis. Because the Mica2 mote exhibits drastic current swings (from 30 μA to 30mA) the measurement circuit was designed to respond to the swings but not drastically change the voltage supplied to the mote. A 1.03 Ω resistor was placed in series with the mote and an instrumentation amplifier (Analog Devices AD620[3]) with a gain of 106 was used to amplify the voltage across the resistor. This amplified voltage was measured by the scope. Input offsets induced by the amplifier were measured and accounted for during data analysis.

Several benchmarks required synchronization between the measured signal and the application. In these cases, data capture was initiated by raising an I/O pin on the mote's microcontroller. Digital multi-meters which are capable of accurately capturing extremely small currents were used to measure the low power modes.

Figure 2 illustrates a high-resolution data capture of the current consumption for transmitting a radio message. In this example the mote starts in a low power state (consuming less than 100 μA), wakes up, and transmits the message. The TinyOS radio stack uses the Carrier Sense Multiple Access (CSMA) collision avoidance protocol. When using CSMA, sending a message requires the mote to listen to the radio channel to detect potential collisions before beginning transmission. The figure clearly shows the discrete power levels for each of these operations.

3.2 Micro-benchmarks

Isolating the hardware consumption of each of the mote's devices (CPU, radio, EEPROM, etc.) required developing a set of micro-benchmarks that exercise each component independently. Two different micro-benchmarks were written to capture the current drawn by the CPU. The first spun in a series of loops with known instruction counts in order to capture the per-instruction current consumption overhead. The second exercised each of the seven CPU power states supported by the ATmega128L. Measuring the energy required to transmit and receive messages is possible with a standalone benchmark that periodically broadcasts a message; as shown in Figure 2, message transmission requires an initial listen period to avoid channel contention, which is straightforward to isolate in the trace. The CC1000 radio supports programmable transmission power levels ranging from -20 dBm to $+10$ dBm. A micro-benchmark was written to measure energy used at different transmit powers. Likewise, another micro-benchmark measured the EEPROM by performing a series of read and write operations, raising an I/O pin to trigger data collection at the appropriate point.

In order to characterize the analog-to-digital converter and the sensors a micro-benchmark was written that enabled the ADC and then periodically sampled the sensors. A general purpose I/O pin was raised when a sensor reading was requested and then cleared when the value was returned. Both the light and the temperature sensor were tested.

3.3 Hardware power model

Table 1 presents the resulting power model for the Mica2 hardware platform. As the table shows, the different CPU power modes cover a wide range of current levels, from 103μA in the "power down" state up to 8mA when ac-

Mode	Current	Mode	Current
CPU		**Radio**	
Active	8.0 mA	Rx	7.0 mA
Idle	3.2 mA	Tx (-20 dBm)	3.7 mA
ADC Noise Reduce	1.0 mA	Tx (-19 dBm)	5.2 mA
Power-down	103 μA	Tx (-15 dBm)	5.4 mA
Power-save	110 μA	Tx (-8 dBm)	6.5 mA
Standby	216 μA	Tx (-5 dBm)	7.1 mA
Extended Standby	223 μA	Tx (0 dBm)	8.5 mA
Internal Oscillator	0.93 mA	Tx (+4 dBm)	11.6 mA
LEDs	2.2 mA	Tx (+6 dBm)	13.8 mA
Sensor board	0.7 mA	Tx (+8 dBm)	17.4 mA
EEPROM access		Tx (+10 dBm)	21.5 mA
Read	6.2 mA		
Read Time	565 μs		
Write	18.4 mA		
Write Time	12.9 ms		

Table 1: Power model for the Mica2. ***The mote was measured with the micasb sensor board and a 3V power supply.***

tively executing instructions. Likewise, the choice of radio transmission power affects current consumption considerably, from 3.7mA at -20dBm to 21.5mA at +10dBm. However, in many of our applications the radio is almost always listening for incoming messages, which consumes 7mA regardless of transmission activity.

Note that it is possible to achieve even lower energy consumption (down to 30μA or so) by disabling the external oscillator, JTAG interface, and making various hardware modifications to the Mica2 platform. These features have been exploited to achieve very long lifetimes in certain applications [24], but are not used by the bulk of applications provided in the TinyOS codebase. In any case, the power model can be readily modified to account for decreased current consumption in "snooze" states if one wishes to evaluate the effect of these changes.

Another component of power consumption is the current drawn by the sensors themselves. On the Mica2, analog-to-digital conversion (ADC) does not consume a measurable amount of energy, given that the ADC logic is incorporated into the CPU. However enabling or disabling particular sensors may cause additional energy consumption while those devices are active. The simple Mica2 sensor board does not specifically gate the supply voltage for each sensor, and thus consumes a constant current of 0.7 mA when it is attached to the mote, regardless of whether the particular sensors are being sampled. If a more sophisticated sensor board is used, our power model can easily incorporate the current draw of individual sensors.

3.4 *In situ* power monitoring with MoteLab

Apart from the rather elaborate measurement setup shown in Figure 1, we have found it valuable to obtain real-time power consumption data from sensor nodes in a larger-scale testbed. We have developed a web-based interface to a wired network of 30 Mica2 nodes distributed throughout our building, called MoteLab. Each node is connected to an Ethernet-based backchannel board that provides TCP/IP access to each mote's serial port for data capture and reprogramming. The web interface allows a user to remotely reprogram the entire testbed with specified TinyOS binaries, and logs messages sent to each mote's serial port to a database. The data can be accessed while the job is running or used for later analysis.

As part of this environment, we have instrumented one node with a network-attached digital multimeter, and implemented a server component that logs the power consumption of this node to the database as well. This setup provides *in situ* power measurements of the sensor node as it participates in the sensor network application provided by the user, requires no additional configuration (other than clicking a checkbox in the web interface to enable the power monitoring) and saves considerable effort on the part of sensor network developers. We have found it to be extremely valuable for validating experiments conducted in the PowerTOSSIM environment.

4. DESIGN OF POWERTOSSIM

In this section we describe the overall design of PowerTOSSIM, which consists of instrumentation of the TinyOS codebase to track hardware power state transitions, an accurate CPU cycle counting mechanism based on basic-block-level profiling, and analysis tools to visualize and analyze power consumption results on a per-mote basis.

4.1 TinyOS and TOSSIM

PowerTOSSIM is based on the TinyOS [10] operating system and the TOSSIM [13] simulation environment. TinyOS is a component-oriented, event-driven operating system for sensor networks, consisting of a simple FIFO task scheduler and numerous software components for sensing, radio communication, EEPROM access, and other devices. TinyOS is not a binary kernel *per se*; rather, one assembles a TinyOS application by linking multiple software components into an optimized binary. TinyOS is written in NesC [8], a C-based language that provides support for the TinyOS component and concurrency model.

TinyOS has become a popular environment for experimenting with sensor network applications, due to its modular nature and support for several common sensor node platforms. As described in Section 2, TinyOS supports a simulation environment called TOSSIM [13]. In TOSSIM, the TinyOS application is compiled directly into an event-driven simulator that runs on the simulation host. This design exploits the component-oriented nature of TinyOS by effectively providing drop-in replacements for the TinyOS components that access hardware; TOSSIM provides simulated hardware components such as a simple radio stack, sensors, and other peripherals. This design allows the same code that is run on real hardware to be tested in simulation at scale. PowerTOSSIM makes use of the TinyOS and TOSSIM component model to instrument hardware state transitions for the purpose of tracking power consumption. This is described in detail below.

4.2 PowerTOSSIM architecture

Figure 3 illustrates the architecture of PowerTOSSIM. PowerTOSSIM tracks the power state of each hardware component of the simulated motes by generating specific *power state transition messages* that are logged during the simulation run. This is accomplished by instrumenting the TOSSIM simulated hardware components with calls to a new component, *PowerState*, which tracks hardware power states for each mote and logs them to a file during the run. Estimating CPU usage is somewhat more involved: since PowerTOSSIM runs the mote software as a native binary on the host machine, it has no information on the length of

time that a given mote spends using the CPU. CPU profiling is accomplished by mapping the basic blocks executed by the simulation code to cycle counts in the corresponding mote binary; this is described further in Section 4.4.

The power state data generated by PowerTOSSIM can be combined with a *power model*, such as that described in Section 3, to determine per-mote and per-component energy usage. This processing can be performed using offline tools to obtain detailed traces of power consumption for each hardware component of each mote, or fed into the TinyViz visualization tool to display power consumption data in realtime.

We chose to decouple the generation and processing of power state transition data for two reasons: efficiency and flexibility.

Efficiency:. One of the important characteristics that PowerTOSSIM inherits from TOSSIM is its very high efficiency in simulating large networks that scale to thousands of nodes. To preserve this efficiency, it is important to avoid introducing high overheads into the simulation itself. Logging hardware state transition messages at runtime introduces very low overhead. Likewise, allowing the simulation to run as a native binary avoids the overhead of instruction-level simulation.

Flexibility:. It is also important to provide a high degree of flexibility for capturing and modeling the power state of the mote; we did not want to assume a particular hardware platform, as new designs are constantly being considered. With the decoupled design it is possible to evaluate the power efficiency of potential hardware designs only by plugging a new power model into the analysis tools; the simulation itself need not be re-executed.

The following subsections present the design of each part of the PowerTOSSIM architecture in detail.

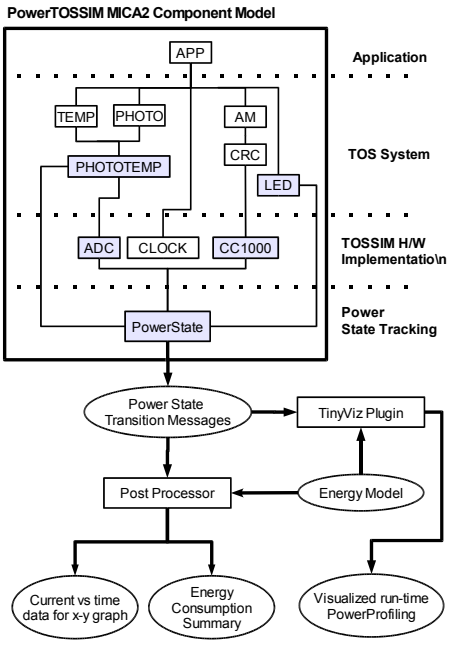


Figure 3: PowerTOSSIM architecture. ***Simulated hardware components (radio, sensors, LEDs, etc.) make calls to the* PowerState *module, which emits power state transition messages for each component. These messages can be combined with a power model to generate detailed power consumption data or visualizations. (AM is the active messaging component that makes up the upper level of the radio stack)***

4.3 Component instrumentation

In TinyOS, each of the mote's hardware components is driven by a specific software module that is responsible for controlling its operation. For example, the *CC1000RadioIntM* module handles most aspects of radio communication using the ChipCon CC1000 radio. TOSSIM replaces many of these hardware drivers with its own simulated versions, making it possible to link a TinyOS application to the simulated hardware with very few code changes. PowerTOSSIM leverages this design by instrumenting each of the simulated hardware drivers with power state transition messages that are logged during the simulation. This is accomplished by adding calls from each hardware driver to a new module, called *PowerState*, that emits log messages when the power state of each hardware device changes. Abstracting power state transitions in a separate module allows the interface to be readily extended to support new hardware components, such as new sensor platforms.

4.4 CPU profiling

As described above, TOSSIM achieves scalability by compiling the TinyOS application code into a binary that runs directly on the simulation host. While very efficient, this design makes it difficult to determine how much time each simulated mote's CPU spends in the *active* state (actively executing instructions) versus the *idle* state, or any of the other low-power states. Tracking the amount of time the CPU spends active is important for accurately capturing power consumption, especially for applications that make use of CPU-intensive operations (such as encryption) or that spend much time in low-power sleep modes, only to wake up and perform computation infrequently.

The simple microcontrollers used on current sensor node designs consume approximately constant power while executing instuctions. This is due to the fact that they do not use the sophisticated chip level power management strategies present in more advanced processors, so the instuction core, SRAM, ADC, oscillator, timer, and other peripherals are always on when the controller is in active mode.

Recalling Table 1, the ATMEL Atmega128L CPU consumes about 8 mA while executing instructions, and 3.2 mA while idle. Likewise, the cycle time for each instruction is well-documented and usually deterministic, or at least predictable. Therefore, to compute CPU energy usage it is adequate to track the amount of time the CPU spends in each power state. The amount of time that a node spends in idle mode depends on external factors, such as the timing of clock interrupts, that are already modeled by TOSSIM. However, the amount of time spent executing CPU instructions is not captured by TOSSIM.

Determining CPU execution time can be accomplished by simulating the execution of each instruction, although doing so for large sensor networks would incur a tremendous performance penalty. Our approach is to (1) instrument

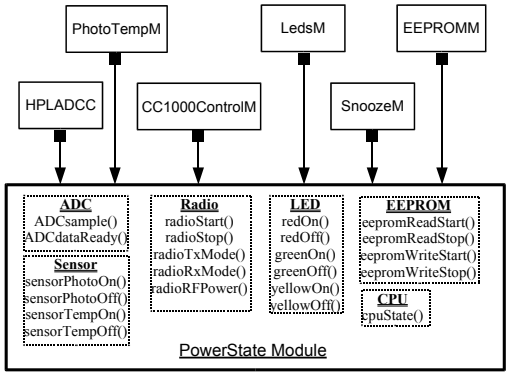


Figure 4: Wiring of the PowerState module. ***Each hardware component is wired to the PowerState module as shown in the figure. Whenever a power state change occurs the corresponding command in PowerState is invoked to generate a state transition message.***

the PowerTOSSIM binary to obtain an execution count for each *basic block* (run of instructions with no branches) executed by the simulated CPU; (2) map each basic block to its corresponding assembly instructions in the AVR binary; (3) determine the number of CPU cycles for each basic block using simple instruction analysis; and (4) combine the simulation basic block execution counts with their corresponding cycle counts to obtain the total CPU cycle count for each simulated mote.

At the end of the simulation run, PowerTOSSIM logs basic block execution counters that are processed offline in the manner described above to obtain CPU cycle count totals. As we show below, this process is fairly accurate and incurs very little overhead at simulation time. There are several potential sources of inaccuracy, however, due to the presence of code in the simulation binary with no corresponding code in the AVR binary, and vice versa. In addition, some difficulties arise with mapping basic blocks to AVR instructions, as described in Section 5.3.

4.5 Analysis tools

Once the power and CPU state data has been obtained by the simulation run, several tools are available to permit analysis and visualization. For example, an offline postprocessor generates power consumption time-series data as well as tallies of the per-mote and per-component power consumption for the entire simulated sensor network. Another tool makes use of the basic block execution counters to perform hotspot analysis on the application code. We have also implemented several plugins for TinyViz, the TOSSIM visualization tool, that allow the user to monitor the power consumption of simulated motes in realtime.

5. IMPLEMENTATION

In this section we detail the implementation of PowerTOSSIM.

5.1 PowerState Module

To avoid scattering power state tracking code throughout the simulator, we created a single TinyOS module, called *PowerState*, that other TinyOS components make calls to in order to register hardware power state transitions. Figure 4 depicts the *PowerState* module and its connections to other TinyOS components; *PowerState* consists of a single interface with one command for each possible state transition. Each function tests if power profiling is enabled, and if so, emits a log message detailing the mote number, the specific power state transition, and the current simulation time. An excerpt from this log is shown below:

```
0: POWER: Mote 0 LED_STATE RED_OFF at 18677335
0: POWER: Mote 0 LED_STATE YELLOW_OFF at 18677335
0: POWER: Mote 0 ADC SAMPLE RSSI_PORT at 18990479
0: POWER: Mote 0 ADC DATA_READY at 18990679
0: POWER: Mote 0 RADIO_STATE TX at 18993551
0: POWER: Mote 0 RADIO_STATE RX at 19199375
```

5.2 Component instrumentation

A number of TOSSIM components representing hardware drivers were instrumented with calls into the *PowerState* module. These include components representing the radio, CPU power state control, ADC, LEDs, EEPROM, and photo and temperature sensors.

Very few modifications were required to the original TOSSIM code to track the power state of these various devices. A total of only 46 lines of code were changed or added into existing TinyOS components for power state tracking. These lines are merely wiring and function calls to commands in *PowerState* module. A summary of the lines affected in each component is shown in Table 2. The *PowerState* module itself consists of about 400 lines for formatting and printing state messages. We discuss the interesting points of this instrumentation below.

The radio is a major power consumer on the motes and tracking its state accurately is essential to the success of any power profiling tool for sensor networks. Unfortunately, TOSSIM does not include a Mica2 CC1000 radio stack because it was designed to model the original Mica platform, which used a different radio chip. We ported the Mica2 CC1000 radio stack to TOSSIM to accurately capture the radio chip's power state.

The Atmega128L CPU used by the Mica2 has seven distinct power states: *active*, *idle*, *ADC noise reduction* (used to reduce noise while taking ADC readings), *power down*, *power save*, *standby*, and *extended standby* [4]. While few (if any) TinyOS applications exploit all of these states, capturing them is relatively straightforward in PowerTOSSIM. By default, the CPU is in the *active* state while executing instructions, and in the *idle* state while waiting for an incoming interrupt. As described in Section 5.3, below, we capture the amount of time in the *active* state through CPU cycle estimation, and assume the CPU is in *idle* unless it has been explicitly put into one of the low power sleep modes.

Analog-to-digital conversion is used to sample analog sensor data from the physical environment. On the Mica2, ADC is built-in to the CPU and does not consume a discernible amount of power when active, although alternative sensor node platforms may incorporate an external ADC that can be independently enabled or disabled. Therefore, we have instrumented the TinyOS *ADCC* component to track the amount of time that the ADC is active, although in our Mica2 power model it does not contribute to overall energy consumption.

5.3 CPU profiling

Our approach for determining the amount of time that each simulated mote's CPU is active is to instrument the

Category	TinyOS Component	Message generated	Files involved	Number of lines added for state tracking
System	LedsC	LED_STATE	LedsC.nc, LedsM.nc	9
	Main	CPU_CYCLES	Main.nc	2
Platform-specific	ADCC	ADC	ADCC.nc, HPLADCC.nc	6
	CC1000RadioC	RADIO_STATE	CC1000RadioC, CC1000ControlM	8
	EEPROM	EEPROM	EEPROM.nc, EEPROMM.nc	7
	SnoozeC	CPU_STATE	SnoozeC.nc, SnoozeM.nc	8
Sensorboard	PhotoTemp	SENSOR_STATE	PhotoTemp.nc, PhotoTempM.nc	6
Total				46

Table 2: Summary of changes to TinyOS for power state instrumentation. ***Only 46 lines were added to existing TinyOS components.***

Task	PowerTOSSIM	atemu	error (%)
hash(char[27])	3002	2956	1.5
hash(char[404])	45415	40279	11.3
qsort(int[100])	69581	92598	-33.0
encrypt(char[100])	103756	106096	-2.3
decrypt(char[100])	105075	107890	-2.7

Table 3: Accuracy of the cycle count estimator. ***This table shows the number of CPU cycles executed by several CPU-intensive operations as measured by our basic-block estimation technique and Atemu, an instruction-level simulator for the AVR platform.***

TOSSIM application to report basic block execution counts, and then map each basic block to the appropriate number of instructions as executed by the mote. This avoids the overhead of instruction-level simulation while maintaining relatively accurate CPU cycle counts.

The NesC compiler parses an entire TinyOS component graph and emits a single C source code file that is then compiled with a back-end compiler appropriate for the target platform (e.g., `avr-gcc` in the case of the Mica2 motes). We exploit this design by instrumenting this intermediate C source file with basic block execution counters. This is accomplished using C Intermediate Language (CIL) [16], a toolkit for programmatically manipulating C source code files. CIL builds a high-level representation of the structure of the C program and permitting easy analysis and source-to-source transformations.

Our use of CIL involved writing a set of source transformation rules that inserted an execution counter into each basic block of the program. This transformation requires less than 100 lines of OCaml code. In addition to modifying the NesC-generated source code, the transformation code records the original NesC source file and line number corresponding to each basic block, allowing us to map basic blocks back to their original source code line number.

The next step is to map each basic block to the appropriate number of CPU instructions as executed by the Atmega128L on the Mica2. This is accomplished by compiling the TinyOS application to a Mica2 binary and using a disassembler (*avr-objdump*) to determine the set of AVR instructions for each source code line. This is then correlated with the basic block line number information to obtain the set of assembly instructions corresponding to each basic block. To obtain CPU cycle counts, the number of cycles for each AVR instruction in the basic block (obtained from the ATmega128L data sheet [4]) is totaled. In cases where an instruction can consume a variable number of cycles, the expected number of cycles for each instruction is used.

After the simulation executes, the basic block execution counts are recorded in the PowerTOSSIM log. Postmortem analysis is then used to determine the CPU cycle counts for each mote using the technique described above.

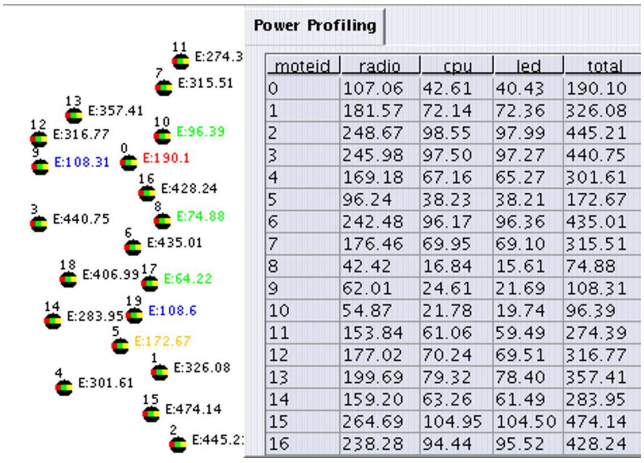
There are several complications with this scheme. One is that a single C source code line may get compiled into several basic blocks, each of which is counted separately in the simulated code. Since the C-to-assembly mapping contains all of the assembly instructions for each source line, whenever *any* basic block contained in the line gets executed, it counts as if the entire line was executed. This leads to overcounting.

Another problem is that not all of the code executed by the simulation is present in the AVR binary, since TOSSIM provides its own replacements for a number of low-level hardware modules. These basic blocks have no mapping to AVR assembly instructions, and are simply ignored in the CPU cycle counts, which leads to undercounting cycles. However, nearly all of this code corresponds to low-level hardware drivers, which are primarily accounted for by the other components of the hardware model. The bulk of the TinyOS codebase (system-level services, libraries, and application code) is shared between the TOSSIM and mote environments, and accounts for a significant proportion of the CPU execution time of typical applications.

The accuracy of our cycle count estimator is shown in Table 3. For comparison we obtained cycle count numbers from Atemu [11], an instruction-level simulator for the AVR platform. Because Atemu simulates the individual instructions, we assume that the cycle counts it produced are extremely accurate. Encryption and decryption operations were performed using the Skipjack encryption module from TinySec [12]. As the table shows, our method is fairly accurate for application level code, although some discrepancies do arise, for example, in *qsort()*. We believe this to be due to cases where a small number of basic blocks that happen to have slightly inaccurate cycle mappings account for the majority of CPU time.

5.4 Analysis tools

We have developed several tools for analyzing and visualizing the power consumption data generated by PowerTOSSIM. These tools take as input the log files generated by *PowerState*, the CPU profiling information, and a hardware power model. The first tool is a *postprocessor* that computes energy totals for the various hardware components for each mote as well as outputting a time series trace of power con-



moteid	radio	cpu	led	total
0	107.06	42.61	40.43	190.10
1	181.57	72.14	72.36	326.08
2	248.67	98.55	97.99	445.21
3	245.98	97.50	97.27	440.75
4	169.18	67.16	65.27	301.61
5	96.24	38.23	38.21	172.67
6	242.48	96.17	96.36	435.01
7	176.46	69.95	69.10	315.51
8	42.42	16.84	15.61	74.88
9	62.01	24.61	21.69	108.31
10	54.87	21.78	19.74	96.39
11	153.84	61.06	59.49	274.39
12	177.02	70.24	69.51	316.77
13	199.69	79.32	78.40	357.41
14	159.20	63.26	61.49	283.95
15	264.69	104.95	104.50	474.14
16	238.28	94.44	95.52	428.24

Figure 5: Screenshot of the PowerProfiling plugin for TinyViz. ***The table on the right reports a run-time summary of energy consumed by each component of the simulated network. Each mote is also assigned a color based on the total amount of energy that it has consumed since the start of the run.***

sumed by each mote (example graphs of these traces are shown in Section 6).

We have also implemented a plugin for TinyViz, the Java-based visualization environment for TOSSIM, that reports per-mote power consumption as the simulation runs. The plugin also assigns a color to each mote based on how much power it has consumed during the course of the run, making it possible to visualize power hotspots in the network. Figure 5 shows a typical screenshot of the visualization.

Another tool uses the basic block execution counters produced by the CPU profiler and generates an ordered list of the TinyOS components that consume the most CPU cycles during the run. This list can also be broken down by basic block. This analysis is extremely valuable for understanding the overhead of various components in a TinyOS application. Note that analysis of the TOSSIM binary itself is generally not adequate as there is little correspondence between the number of instructions executed on, e.g., an x86 machine and that on an 8-bit microcontroller lacking hardware floating-point support. Table 4 shows an example of this tool run on the TinyDB [15] sensor network query processor.

6. EVALUATION

This section presents the evaluation of PowerTOSSIM along two axes: validation of the simulated power consumption data against real hardware for a range of applications, and PowerTOSSIM's ability to scale to very large simulations.

6.1 Total Energy Consumption

To validate PowerTOSSIM's energy consumption estimates, we ran a range of real TinyOS applications both on PowerTOSSIM and on an actual mote that was instrumented to obtain power traces. Most of these applications were drawn from the standard TinyOS distribution and were chosen to represent a broad range of typical sensor network programs. Many of the applications, such as *Blink* and *Sense*, are sim-

Rank	Cycles	Component
1	2191584.5	TimerM
2	661187.5	TupleRouterM
3	577066.5	SimpleTimeM
4	383780.0	sched.c
5	371004.5	TimeUtil
6	153594.0	AbsoluteTimer
7	134514.0	Time
8	95567.5	TimeUtilC
9	16119.0	ADCM
10	12029.0	tos.h (memcpy, memset, rcombine)
11	9780.5	TinyAlloc
12	7579.5	AMPromiscuous
13	7186.0	MultiHopLEPSM
14	5868.0	Attr
15	3916.0	HPLADC
16	2880.0	SendMsg
17	2187.5	QueuedSendM
18	1743.5	Leds
19	1505.0	Command
20	1180.0	ADC

Table 4: Hotspots in the TinyDB query-processing engine. ***This table shows the top 20 most CPU intensive components in the TinyDB application. Results are for a 60 second run with the query "SELECT light FROM sensors SAMPLE PERIOD 1024."***

Benchmark	Simulated	Measured	Error (%)
Beacon	92.93	106.73	-12.9
Blink	940.26	931.72	0.85
BlinkTask	940.28	917.90	2.5
CntToLeds	1336.49	1330.00	0.45
CntToLedsAndRfm	2620.37	2562.00	2.3
CntToRfm	2028.09	1985.00	2.1
Oscilloscope	867.94	801.60	8.3
OscilloscopeRF	2136.45	2021.90	5.7
Sense	865.59	900.72	-3.8
SenseLightToLog	2133.89	2005.26	6.4
SenseTask	865.62	944.74	-8.3
SenseToLeds	868.70	977.73	-11.1
SenseToRfm	2152.27	2059.16	4.5
Average			4.7
TinyDB (idle)	2001.31	2275.55	-12.1
TinyDB (select light)	2144.86	2465.30	-13.0
Surge	2089.09	2028.40	3.0
Average			9.5

Table 5: Measured versus simulated energy for various TinyOS applications. ***All applications were executed for 60 real or simulated seconds and all values are in millijoules (mJ).***

ple demonstration programs that exercise particular features of the motes. Most of these programs perform some combination of sensing, blinking LEDs, transmitting radio messages, and recording data in the EEPROM; their function is mostly self-explanatory from the program name.

Beacon is the only program in our benchmark suite that uses the low-power sleep states of the Mica2. It transmits a radio beacon message every two seconds but drops the CPU to a low power mode between each beacon; the other programs consume considerably more CPU energy since they sit in the (relatively higher-power) idle mode when not executing instructions.

In addition to these simple benchmarks, we examined two more involved applications: *TinyDB* and *Surge*. TinyDB [15] is a sophisticated program that provides a query interface to sensor network data. It involves sensing, aggregation, communication, and computation. Surge is an adaptive multihop routing protocol that forms a spanning tree rooted at the wired base station node. Each node periodically takes a sensor reading and routes it to the root. Surge performs complex radio link quality estimation for parent selection in the routing tree. These applications were run in a small network of several motes, as described in Section 6.4, below.

Table 5 presents total simulated and measured energy numbers for a 60 second run of each application. As the table shows, PowerTOSSIM achieves excellent accuracy, with an average error of just 4.7% compared to the actual mote, and a maximum error of about 13%. Some of this difference between simulated energy and measured energy can be attributed to voltage fluctuations, noise and rounding error in the experimental setup. Other differences may be due to inaccuracies in the power model or in CPU cycle counting, though we believe that we are accurately capturing nearly all of the hardware power transition states.

6.2 Energy Breakdown By Component

Table 6 shows the simulated energy consumption for each application broken down by hardware component. PowerTOSSIM makes it extremely easy to perform this kind of analysis, which is otherwise extremely difficult for real sensor nodes.

The behavior of several applications depends on the particular sensor readings that they obtain. For example, *SenseToLeds* takes a light sensor reading and displays the high-order bits of the reading using the 3 LEDs. Consequently, the total energy consumed depends on the light intensity of the environment, which is difficult to match in the simulation. To avoid this, we simply disabled the LEDs in these applications, which explains why the LEDs are shown as not consuming energy for several applications that would otherwise have done so.

Some interesting observations can be made using this data. While active CPU energy (when executing actual instructions) is very small, the total CPU energy consumes a significant fraction of the energy for the entire run. This is because most applications leave the CPU in the 3.2 mA idle state rather than using the lower-power sleep modes. However, even for the Beacon application, which uses low-power sleep, the CPU consumes a considerable fraction of the overall energy. In this case, the CPU must be active to transmit outgoing beacon messages a byte at a time. Packet-based radio interfaces (such as the Chipcon CC2420) may provide significant savings in this regard. Another lesson to take away is that the LEDs draw a significant amount of current, and should be used for debugging but disabled when the application is actually deployed.

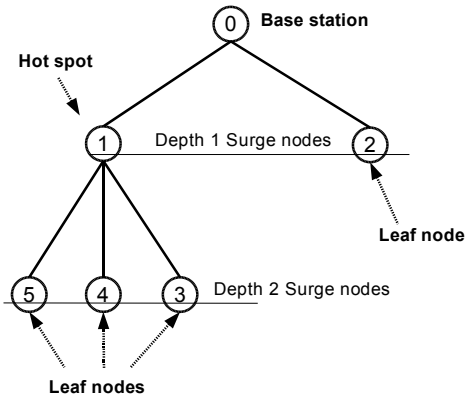


Figure 8: Six-node topology used to evaluate Surge. ***Mote 1 is the "hotspot" since Motes 3, 4, 5 all rely on it to relay their messages to the base station Mote 0.***

6.3 Current traces

In addition to total energy consumption data, PowerTOSSIM outputs realtime traces of simulated current draw. We verified that these traces correspond well with the measured oscilloscope trace for the same application.

Figure 6 shows the measured and simulated current for the Beacon application. Note the two small peaks between every pair of large peaks. The large peaks every 2 sec are due to message transmissions, while the smaller peaks are due to the CPU waking up briefly to increment a counter. PowerTOSSIM successfully captures all the details of the real trace, including the spurious CPU wakeups.

Many of the sensor applications that we tested had rather dry behaviors, that is, they spend most of their time in idle mode and transmit a message or blink an LED periodically. To demonstrate a more interesting application, we measured *CntToLedsAndRfm*, which increments an internal counter each time a 4 Hz timer fires. The lower 3 bits of the counter are displayed on the 3 LEDs, resulting in an interesting pattern of current consumption as shown in Figure 7. The program also transmits a message with the counter value on each timer interrupt. The graph clearly shows the radio messages and the LEDs cycling through the sequence from 0 to 7.

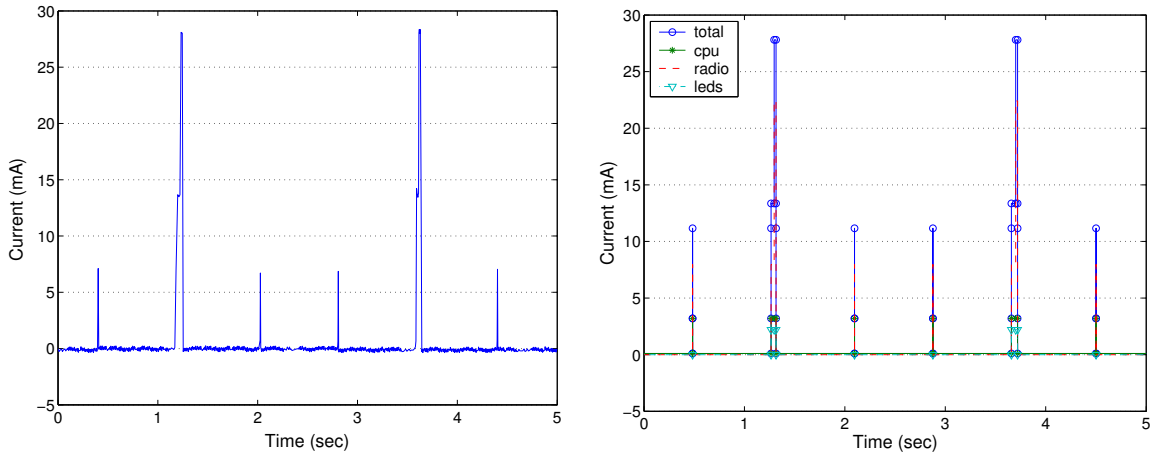
6.4 Networked applications: TinyDB and Surge

In addition to the simple benchmarks presented earlier, we evaluated *TinyDB* and *Surge*, which represent more complex sensor network applications. We find that PowerTOSSIM does a very good job at estimating the power consumption of these applications as well.

The last 3 lines Table 5 present the results for TinyDB and Surge. There are two TinyDB tests. The first is TinyDB in idle mode, with no running queries. In this mode, TinyDB simply performs periodic network topology maintenance and listens for new queries. The second measurement was taken

Application	CPU idle	CPU active	Radio	Leds	Sensor Board	EEPROM	Total
Beacon	35.86	0.58	47.68	8.81	0.00	0.00	92.93
Blink	742.50	0.25	0.00	197.52	0.00	0.00	940.26
BlinkTask	742.50	0.27	0.00	197.52	0.00	0.00	940.28
CntToLeds	743.72	0.57	0.00	592.20	0.00	0.00	1336.49
CntToLedsAndRfm	741.90	1.61	1284.65	592.20	0.00	0.00	2620.37
CntToRfm	741.90	1.54	1284.65	0.00	0.00	0.00	2028.09
Oscilloscope	742.65	1.46	0.00	0.00	123.82	0.00	867.94
OscilloscopeRF	741.90	1.85	1268.76	0.00	123.95	0.00	2136.45
Sense	742.21	0.38	0.00	0.00	123.00	0.00	865.59
SenseLightToLog	741.90	0.81	1262.95	0.00	123.95	4.28	2133.89
SenseTask	742.21	0.42	0.00	0.00	123.00	0.00	865.62
SenseToLeds	743.72	0.73	0.00	0.00	124.25	0.00	868.70
SenseToRfm	741.90	1.77	1284.65	0.00	123.95	0.00	2152.27
TinyDB (idle)	693.29	10.41	1181.78	0.00	115.83	0.00	2001.31
TinyDBApp (select)	742.85	11.20	1266.70	0.00	124.11	0.00	2144.86
Surge	727.28	1.50	1239.02	0.00	121.30	0.00	2089.09

Table 6: Simulated component energy breakdown for various applications. ***All values are in millijoules (mJ).***



(a) Measured **(b) Simulated**

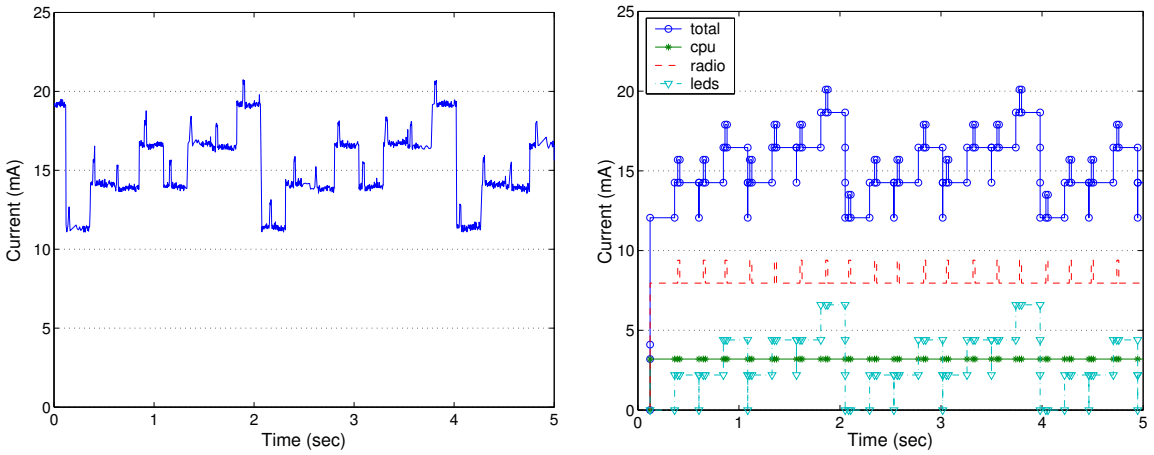
Figure 6: Measured and simulated current consumption for the *Beacon* application. ***The simulated version includes a breakdown according to radio, LEDs, and CPU current. A lower resolution digital multi-meter was used for the above measurement, which did not capture the very short duration peak power spikes during the wakeups.***

running the query "SELECT light FROM sensors SAMPLE PERIOD 1024." This query reports the value of the light sensor on each mote once every 1024 ms. The error for TinyDB is higher than for most of the benchmark applications, though we believe this is partly due to the fact that TinyDB exhibits somewhat different behavior in simulation than it does on actual hardware.

Surge forms a spanning-tree-based multihop route from each node to a wired basestation. Each node performs periodic estimations of the link quality to its neighbors and attempts to select a parent node that will maximize the probability of its messages reaching the base station. Each node samples its light sensor and generates a message once every 2 sec. Therefore, nodes that have children in the routing tree will relay both the messages of their children as well as their own messages. With this approach, it is possible that some nodes become "hotspots" that forward many more packets than other nodes.

As shown in Table 5, PowerTOSSIM captures the overall power consumption of Surge with very high accuracy (3% error). In this measurement, we instrumented a single Surge node with several other network nodes relaying data to it. Using PowerTOSSIM, we were interested to verify whether hotspots in the routing tree really lead to increased power consumption on those nodes. To induce a specific hotspot in the routing tree, we modified the Surge code to always form the topology depicted in Figure 8, and ran the resulting network in PowerTOSSIM. In this arrangement, mote 1 has three children and will have four times more radio packets to send out than other nodes in the network.

Figure 9 shows the simulated energy consumption for each node as well as the total number of transmitted messages. The hotspot node sends out about three times more messages than the other motes in the network, as expected. However, the energy consumption of the hotspot is not very different from the other nodes. This is because Surge does not use the low power CPU and radio modes between transmitting messages. Because the relative rate of message forwarding is quite low (no more than 2 messages/sec on the hotspot node versus 0.5 messages/sec for the other nodes),



(a) Measured (b) Simulated

Figure 7: Measured and simulated current consumption for the *CntToLedsAndRfm* application. *The simulated version includes a breakdown according to radio, LEDs, and CPU current.*

there is very little difference in the simulated energy consumption for each node.

6.5 Scalability

Finally, we evaluate the overhead of power profiling in PowerTOSSIM with respect to both the original TOSSIM environment upon which it is based, as well as Atemu [11], an AVR instruction-level simulator that is capable of simulating multiple motes. Performance tests were run on a 2.4GHz Pentium 4 machine running Linux with 512MB of main memory and 512KB L2 cache. The results are shown in Figure 10. In all cases we ran the *OscilloscopeRF* application which takes periodic sensor readings and broadcasts them as radio messages. Each run was for 300 virtual seconds.

As can be seen, PowerTOSSIM without CPU cycle counters (that is, only instrumenting the hardware state transitions using *PowerState*) has little measurable overhead over TOSSIM. In the 1000-mote case, the time for PowerTOSSIM was within 1% of the time for TOSSIM. Adding basic block counters increases the amount of memory used in the simulator, which causes a slowdown when simulating a large number of motes. For OscilloscopeRF, there are approximately 4KB of counters per mote, which overflows the 512KB L2 cache at about 125 simulated motes. This explains the inflection point above 100 simulated motes in the figure.

We also compare PowerTOSSIM's performance to Atemu, which performs instruction-level simulation and executes each simulated mote as a separate thread. We show data for up to 50 simulated motes in Atemu. For this application, Atemu is approximately 20 times slower than PowerTOSSIM due to the high overhead of instruction-level simulation, which limits its value for simulating large networks. However, Atemu does provide very accurate instruction-level analysis of the simulated mote, which can be valuable when this level of detail is warranted.

6.6 Environmental factors

We were able to achieve excellent correspondence between simulation and our experiments. However, a real world deployment would likely see more variance in actual energy usage due to environmental variation that can be carefully controlled in the lab. The main factors are:

- *Supply voltage* – This is the most important factor that can contribute to energy use variability. Voltage has a linear effect on current and a quadratic effect on power and energy. As motes such as the Mica2 commonly run on regular batteries with a voltage range from 1.8 to 3.0 volts, this can lead to very large fluctuations in energy use.

- *Event response to environment* – As discussed above, sensor network applications often have behavior that depends on the particular readings obtained. Thus, the energy used by a network may be vastly different if it detects a lot of events that require some sort of processing (sending radio messages, blinking LEDs, etc). This is fairly obvious, but is hard to quantify in a simulator.

- *Temperature* – Temperature has a fairly small effect on active current (only a few percent). However, if the device is in an extremely low power sleep mode temperature variations can have a more drastic affect on leakage current. Since nodes in many sensor network applications spend much of their time in sleep mode, it is important to be aware of this. Unfortunately, it is difficult to quantify in general, and we were unable to obtain any datasheets on it, as it depends on manufacturing tolerances.

Environmental factors like temperature and humidity can also affect the reliability of the motes [24]. As a result, it is difficult to predict the overall failure behavior of the network.

7. FUTURE WORK AND CONCLUSIONS

As the sensor network community expands in terms of size and scope of interest, we believe it is critical that effective tools be produced to understand the behavior of sensor

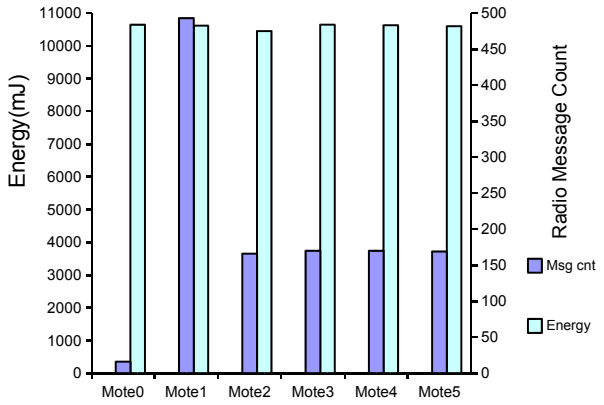


Figure 9: Simulated energy consumption vs. radio message counts for Surge run. ***Simulations were run for 300 virtual seconds with a 0.5 Hz message rate over the 6-node topology shown in Figure 8. The data shows that each Surge mote consumes roughly the same energy regardless of its position in the topology, although the routing node (node 1) sends out 3 times more messages than the leaf nodes (nodes 2 through 5).***

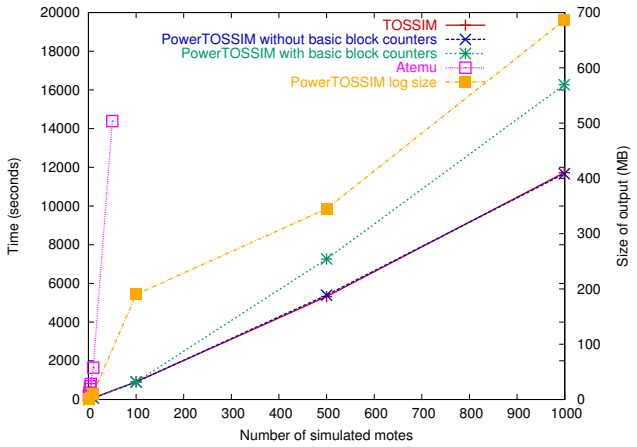


Figure 10: Scalability of PowerTOSSIM as the number of simulated motes is increased. ***The OscilloscopeRF application was simulated for 300 virtual seconds for an increasing number of motes. PowerTOSSIM introduces very low overhead on top of the basic TOSSIM infrastructure, and scales well with increasing numbers of motes. Adding basic block counting increases overhead somewhat. Also shown is the total size of the PowerTOSSIM log file and the time to simulate up to 50 motes using Atemu, an instruction-level simulator for motes.***

network applications. While real-world deployments are ultimately the ideal evaluation environment, simulation tools have a great deal of value for debugging, testing, and experimenting with applications in a controlled setting.

This paper addresses a pressing need for this community: that of modeling the *power consumption* of sensor network applications. Rather than sacrificing scalability and efficiency by using a low-level hardware simulation, we propose the use of efficient emulation of the sensor node hardware platform coupled with careful instrumentation of the power states. PowerTOSSIM generates an event-driven simulator directly from TinyOS code and emits power state transitions for multiple hardware peripherals (radio, sensors, LEDs, etc.). In addition, PowerTOSSIM obtains an accurate estimate of CPU cycle counts for each mote by measuring basic block execution counts and mapping each basic block to microcontroller instructions. We have demonstrated that PowerTOSSIM obtains very accurate power consumption results for a wide range of TinyOS applications and exhibits very little overhead above that of the TOSSIM environment upon which it is based.

We envision a number of future directions for PowerTOSSIM itself as well as for studies that this tool enables. PowerTOSSIM's flexible power model allows both current and future sensor node designs to be evaluated with respect to power efficiency. We are currently designing a novel sensor node architecture and intend to use the PowerTOSSIM environment to model its behavior in large-scale deployments in advance of taping out actual hardware. As part of this effort we are building a sensor network "benchmark suite" that can be used to understand the tradeoffs between generality, performance, and energy requirements across a range of applications. Finally, PowerTOSSIM makes it possible to understand the energy consumption of a sensor network *as a whole*, rather than simply at the per-node level. Many techniques for sensor data aggregation, routing, filtering, and compression have been proposed in the literature, though few have been evaluated in large-scale settings using a realistic power model. PowerTOSSIM should open up broad avenues for exploration of energy-efficient sensor network algorithms.

8. REFERENCES

[1] Agilent 54832B Infiniium Oscilloscope. `http://www.agilent.com`.

[2] A. C. Amit Sinha. Jouletrack - a web based tool for software energy profiling. In *Proceedings of the 38th Design Automation Conference*, 2001.

[3] Analog Devices AD620 Instrumentation Ampifier. `http://www.analog.com`.

[4] Atmel Corp. ATmega128(L) Datasheet. `http://www.atmel.com/dyn/resources/prod_documents/2467S.pdf`.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA*, pages 83–94, 2000.

[6] K. Fall and K. Varadhan. The *ns* manual. `http://www.isi.edu/nsnam/ns/doc/index.html`.

[7] J. Flinn and M. Satyanarayanan. Powerscope: a tool for profiling the energy usage of mobile applications. In *Second IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–10, Feb. 1999.

[8] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. Programming Language Design and Implementation (PLDI)*, June 2003.

[9] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin. EmStar: A software environment for developing and deploying wireless sensor networks. In *Proc. USENIX'04*, 2004.

[10] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[11] M. Karir. atemu - Sensor Network Emulator / Simulator / Debugger. `http://www.isr.umd.edu/CSHCN/research/atemu/`.

[12] C. Karlof, N. Sastry, and D. Wagner. Tinysec. `http://www.cs.berkeley.edu/~nks/tinysec/`.

[13] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys) 2003*, Nov. 2003.

[14] J. Liu, D. Nicol, F. Perrone, M. Liljenstam, C. Elliot, and D. Pearson. Simulation modeling of large-scale ad-hoc sensor networks. In *Proc. European Interoperability Workshop 2001*, London, England, June 2001.

[15] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proc. the 5th OSDI*, December 2002.

[16] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of Conference on Compilier Construction*, 2002.

[17] S. Park, A. Savvides, and M. B. Srivastava. SensorSim: A simulation framework for sensor networks. In *Proc. MSWIM 2000*, Boston, MA, August 2000.

[18] S. Park, A. Savvides, and M. B. Srivastava. Simulating networks of wireless sensors. In *Proc. the 2001 Winter Simulation Conference*, Arlington, VA, December 2001.

[19] L. F. Perrone and D. M. Nicol. A scalable simulator for TinyOS applications. In *Proc. the 2002 Winter Simulation Conference*, 2002.

[20] T. A. Roth. Simulavr: an AVR simulator. `http://www.nongnu.org/simulavr/`.

[21] G. Simon, P. Völgyesi, M. Maróti, and A. Lédeczi. Simulation-based optimization of communication protocols for large-scale wireless sensor networks. In *Proc. 2003 IEEE Aerospace Conference*, Big Sky, MT, March 2003.

[22] T. Stathopoulos. EmTOS: TinyOS/NesC Emulation for EmStar. `http://cvs.cens.ucla.edu/emstar/ref/emtos.html`.

[23] S. Sundresh, W.-Y. Kim, and G. Agha. SENS: A sensor, environment and network simulator. In *Proc. 37th Annual Simulation Symposium (ANSS '04)*, 2004.

[24] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. Lessons from a sensor network expedition. In *Proc. the First European Workshop on Wireless Sensor Networks (EWSN)*, January 2004.

[25] A. R. T. K. Tan and N. Jha. Emsim: An energy simulation framework for an embedded operating system. In *Proceedings of the International Conference on Circuits and Systems*, 2002.

[26] T. K. Tan, A. Raghunathan, G. Lakshminarayana, and N. K. Jha. High-level software energy macro-modeling. In *Design Automation Conference*, pages 605–610, 2001.